gchii.org

Essential Training Series

Essential EGL+Batch

Overview

About this presentation

This presentation is a companion for the *Essential EGL+Batch* video series. It provides a solid introduction for Enterprise Generation Language (EGL). It features EGL Development Tools (EDT), with emphasis on the EGL+Batch programming model.

Edition

Date Revised: 23 May 2013

About the author

Mr. Herschberger is a software architect and technical writer. He is an advocate of a Java-based operating system and is fluent in EGL, Java, C/C++ and COBOL.

See also

- Website. http://www.mindspring.com/~gchii/
- YouTube channel. http://www.youtube.com/user/gchii3

Thank you

I thank

- Russ G. for his help and motivation to write this presentation;
- The community at the EGL Development Tools Project for building a free-license and open-source product for EGL;
- IBM for DB2 Express-C, contributing to EGL Development Tools Project;
- Literature & Latte for Scrivener;
- Google for YouTube, Blogger and Sites;
- VMware for VMware Fusion; and
- Apple for OS X, iMovie, Garage Band and Podcast Publisher.

Elegant simplicity

To evolve a large library of production quality code, start with the elegant simplicity of EGL+Batch. In the batch environment, you run an EGL program from the command line. Focus on your business needs instead of technology.

There are many reasons to start with EGL+Batch.

- ✓ Reduce your frustration. Don't get overwhelmed with a long list of models and modes supported by EGL.
- ✓ Be frugal. Learn a lot about EGL on a low- or no-cost operating system, such as Linux, OS X and Windows.
- ✓ Be methodical. Build your confidence. Build a personal library of coding examples, tricks and traps.
- ✓ Be productive. Write production-quality libraries. Write simple unit tests for your production library. Write programs to create tables and views, update a production database, validate data, import from and export to files.
- ✓ Be pragmatic. Put off the complexity of multiple language generation, such as EGL Rich UI. Put off the complexity and hardware requirements of a Java EE application server. Put off the complexity of multiple operating system programming on Power Systems, System i, and System z.

The batch environment should be your first step in your journey even if your long-term goal is to build a EGL Rich UI and web service providers. It demonstrates the essential language features, such as defining a variable, implementing a library, debugging a program, designing a record and connecting to a Java Database Connectivity (JDBC)-compatible database.

Why EGL Development Tools?

This presentation features EGL Development Tools (EDT) version 0.8. We feature this product for at least the following reasons.

- ✓ It is an free license product so it does not require a large up-front personal investment or permission from management to write an expense check.
- ✓ It is open source so you can read every line of source code and share your experience and questions with the EDT community.
- ✓ It is a product the well-respected Eclipse Foundation, with an initial contribution from IBM.

Are you new to Eclipse? This presentation includes a brief introduction to the Eclipse workbench. We create an EGL project, modify a Run Configuration and create a runnable Java Archive (JAR). We export a archive file to backup our work. We import an archive file to a new workspace.

Starting at the beginning

This presentation does not assume that you are familiar with a commercial product, such as IBM Rational Business Developer. It does not assume you are an expert with the Java technologies.

The essentials of EGL programming are fully explained, with working source code examples.

Essential EGL Developer

Overview

In this presentation, I will explain the relationship between the Enterprise Generation Language (EGL) and its dependencies:

1. Computer Hardware,
2. Operating System,
3. EGL Development Tools, and
4. Remote Systems Explorer.

Computer Hardware

The first limitation is computer hardware. Your impression of EGL and its tooling depends a lot on your hardware.

The faster your hardware the less you'll wait. Remember you needs enough computer power to run both the tools and the application you're developing.

Underpowered hardware can lead to frustration. Although the EGL generator and Java compiler are optimized, remember that you may eventually work with multiple or large (100,000+ lines of code) projects. The generator and compiler can become a bottleneck in the development process.

Caution: When working with EGL Development Tools (EDT), remember that you may eventually need more powerful hardware to start working with a Java Extended Edition (JEE)-compatible application server, such as IBM WebSphere Application Server, and a JDBC-compatible database, such as IBM DB2.

Hardware Specifications

Here are our recommendations for choosing a computer.

Hardware	**Measure**
Minimum Memory	2GB+ for Linux 32-bit
	3GB+ for Windows 32-bit
Minimum Disk Space	350GB+ for Linux/Windows
Minimum Processor	Dual Core

Minimum Monitor	13” flat screen
Recommended Monitor	27” flat screen (1080p)

Virtual Hardware

Software, such as VMWare Fusion, provides virtual hardware. You can create a virtual disk, for example, without going out and buying another hard drive.

What is a *virtual machine*? A virtual machine is virtualized computer hardware. You can start and stop a virtual machine and its operating system after you start your host operating system on a real machine.

Unfortunately, the Java Virtual Machine (JVM) is misnamed. It is not a virtual machine at all, as in virtualization of hardware, but an *abstract* machine. An official JVM from Oracle requires a real or virtual machine and its operating system to run. It is just another application, like a text editor or web browser.

Operating System

While the first limitation is computer hardware, the second limitation is an operating system. An *operating system* is a collection of fundamental utility programs for your computer. From an application programming point of view, an operating system is also known as a *platform*, or a hardware/software foundation to build upon.

1. The Java Virtual Machine (JVM) is operating system-independent (a.k.a. platform-independent). In other words, it runs on Linux, OS X, Windows and many other operating systems.
2. Eclipse is based upon JVM technology and is also somewhat platform-independent. In other words, the tools run on Linux, OS X and Windows.
3. EGL Development Toolkit (EDT) is based upon Eclipse. Just like Eclipse, it runs on Linux, OS X and Windows.

Your choice of operating system determines what you can and cannot do with your computer hardware. In other words, it contributes to your impression of EGL Development Tools.

Note: A platform-independent application is possible, but not automatic. EGL does not prevent you from writing a platform-specific application. If your goal is to create a platform-independent application, you must be prepared to perform additional work.

In summary, your operating system choices include:

1. Linux
2. OS X
3. Windows

What's Next

Installing software for EGL Development Tools.

- Installing EDT on Linux
- Installing EDT on Mac OS X
- Installing EDT on Windows
- Installing Remote Systems Explorer
- Java Console

EDT on Linux

Overview

Download and install EGL Development Tools version 0.8.2 on Linux 32-bit. While I am featuring OpenSUSE Linux version 11.1 in this presentation, it applies to Linux in general.

Linux

This provides a low- or no-fee license for the operating system. It provides a cost-effective, robust, production-quality operating system in a virtualization solution, such as VMware ESX.

While there are many flavors and versions of Linux available, only two flavors are officially supported by IBM for Rational Business Developer.

1. RedHat Enterprise Linux (RHEL) versions 6 or 5
2. SUSE Enterprise Linux (SEL) versions 12 or 11.

Note: OpenSUSE 11.1 is equivalent to SEL 11, and OpenSUSE 12.1 is equivalent to SEL 12.

For the purpose of learning EGL without expending a lot of time and money, we recommend OpenSUSE 11.1 at the time of this writing.

Essential Linux commands

Some of the essential Linux commands are:

1. `ls`. This command provides a list of files and directories.

Note: It is similar to the `dir` command in MS-DOS.

2. `cd`. This command changes the current working directory. The `pwd` command displays the current working directory.

3. `chown`, `chgrp`, `chmod`. These commands change owner, change group and change mode, respectfully.

4. `man` *`command`*. This command displays a quick start manual for a command.

4. Starting a command in the current directory. For the best possible security, it is essential to never put the dot (refering to the current working directory) in a

command path. Therefore, to run a command in the current directory, use the dot-slash prefix. For example, the `./eclipse` command works; `eclipse` doesn't.

5. root user. The most powerful account on Linux is the superuser account called "`root`".

Recommended: Do not use the superuser account for daily tasks, such as developing an EGL application.

6. Additional accounts. The `useradd` command enables you to add accounts for additional users. An account has a name, and likely, a corresponding password. The `groupadd` command enables you to create a group. A group has zero or more user accounts. Example: `useradd gchii`

7. User, Group and Other. The u (User), g (Group) and o (Other) privileges are associated with every file and directory in a Linux file system. The user privilege grants access based on an individual user account. The group privilege grants access based upon a user account belonging to a group. The other privilege grants access without regard to user or group. Privileges are resolved to the highest privilege using three methods.

8. Read, Write and Execute. The r (Read), w (Write) and x (Execute) privileges are associated with every file and directory in a Linux file system. The read privilege enables you to read a file. The write privilege enables you to write (or overwrite) a file. The execute privilege enables you to invoke a file from the command line, such as a native compiled program or a shell script. And there is an exception for directories, which cannot be invoked. The execute privilege enable you to read the contents of a directory.

9. The root (/) file system. A root file system is established at boot-time. One of the partitions on a disk becomes the root file system. All other file systems must be integrated into the root file system as a directory. The root file system is unrelated to the "`root`" user account.

Procedure 1A: Installing EGL Development Tools on Linux

This procedure illustrates how to download and install EGL Development Tools (EDT) on Linux, featuring version 0.8.

For more information on installing EGL Development Tools on Linux, see also *Essential EGL+Batch - Installing EGL Development Tools on Linux* (video).

Transcript: Installing EGL Development Tools on Linux

Welcome to *Essential EGL+Batch*. This series features EGL Development Tools version 0.8.

In this lesson, we're going to (1) download and (2) install the EGL Development Tools on Linux. We're going to assume that you have met the system requirements for EDT, such as a Java Runtime Environment version 6 or greater.

What you want to do is to open your favorite web browser and in the location field you want to type in "`www.eclipse.org`", which brings you to the main page. And then, you want to select projects, And then, you want to select the list of all projects. Now, there's a lot of projects at the "`eclipse.org`" web site. We want to go to the section on tools, we want to find the section on tools and then select the EGL Development Tools. EGL Development Tools enables you to bring up an EGL+Batch environment.

Now, on the EGL Development Tools page, we want to go to Downloads. And then, on the Downloads page, we'll scroll down and we're looking for the all-in-one package for our platform. And in this case, we'll be installing EGL Development Tools on Linux (32-bit).

We click on that link and it brings us to a page where we want to find a mirror that's close to our location on the Internet. And my favorite mirror is the Georgia Tech Software Library because I'm located near Georgia Tech. We want to save the file.

The file downloads in about six minutes on my computer. I sped that up so you don't have to wait for six minutes of video.

So, the download is complete. The name of the file is "`eglweb-082-linux-gtk.tar.gz`."

Now, to be able to use a GZ-compressed TAR-compatible archive, we want to uncompress and unarchive it.

From a command line, I'll create a new directory called "`eglweb`". I'll use the tar command with the x, v, z and f options to uncompress and unarchive the file.

There! With the file uncompressed and unarchived, I need to switch to the superuser to make the EGL Development Tools available for all users. So, I'm going to change directories to "`/opt/eclipse.org`". I'm going to use the move command to change the name of the "`eclipse`" directory to "`eglweb`". Now, if I go into "`eglweb`" what I'm looking for is the "`eclipse`" command and there it is.

So, now, I can go back to my account. And, I'll switch to "`/opt/eclipse.org/eglweb`". And, I'll start Eclipse with "`./eclipse`". And, that starts the EGL Development Tools.

I'll accept the default directory by pressing the **OK** button.

This has been another lesson in the series Essential EGL Batch. Thanks for watching!

Virtualization

More extreme virtual machine plus operating system choices are available, such as:

1. Linux on Power System
2. Linux on System z
3. Linux on VMware ESX
4. Windows on Linux plus VMware Workstation

EDT on OS X

Overview

Download and install Eclipse Indigo (3.7) on Mac OS X (Lion). Install EGL Development Tools (EDT) with the Install New Software feature of Eclipse.

Procedure 1B: Installing EGL Development Tools on Mac OS X

This procedure illustrates how to download and install EGL Development Tools (EDT) v0.8 on Mac OS X (Lion). Unlike the procedures for Linux and Windows, we install the Eclipse JEE Developer Tools and then install new software.

For more information on installing EGL Development Tools on Mac OS X, see also *Essential EGL+Batch - Installing on Mac OS X* (video).

Transcript: Installing EGL Development Tools on OS X

Welcome to *Essential EGL+Batch*. This series is based upon EGL Development Tools version 0.8.

In this episode, we're going to go to "`www.eclipse.org`" and download the Eclipse IDE for Java EE Developers. And then, we're going to install the EGL Development Tools.

So, we selected Download and then we go over to find "Indigo". And then, on the Indigo page, we look for Downloads for Indigo. We want to make sure that we're downloading Indigo, which is Eclipse 3.7. We want Eclipse IDE for Java EE Developers for Mac OS X (64-bit).

We go to a page that we need to select a mirror, and of course, my favorite mirror is Georgia Tech because I am located near Georgia Tech. The name of the download file is "`eclipse-jee-indigo-SR2-macosx-cocoa-x86_64.tar.gz`." We're going to **Show in Finder**. We're going to right-click and open with the Archive Utility so that it will expand the archive. We'll end up with an "`eclipse`" folder.

I want to rename the "`eclipse`" folder to "`eclipse-indigo`". I want to rename the eclipse executable from "`Eclipse`" to "`Eclipse-indigo`". Now, I'll drag the "`eclipse-indigo`" folder to applications.

With the Launch Pad, I'll be able to start Eclipse Indigo. When I first download and install it, it asks me am I sure that I want to run it. I say **OK**.

It asks me for a workspace. I'll use the default in this demonstration. And, we're loading Indigo.

Indigo does not come with the EGL Development Tools. It comes with a lot of stuff, but not the EGL Development Tools. So, what we want to do is add EGL Development Tools to our Eclipse Indigo.

It's Help. Pull down the Help menu and select Install New Software.

We click the **Add** button to add a site. This ("`http://download.eclipse.org/edt/releases/1.0`") is the update site for EGL Development Tools.

We want to hide items that are already installed; check that box. We want to *not* group items by category; we uncheck that box. And then, we'll install EGL IDE for Web Developers, EGL Web Developer IDE Feature, and EGL Web Developer Tools. And, I'm demonstrating this on 0.8.2.

Click the **Next** button. We confirm, yes, this is what we want to install. We're given a license page. We accept the terms of the license and click **Finish**.

It goes about installing the software. I get this warning about an unsigned package. (Click the **OK** button.)

I'll click **Restart Now**.

I'll choose the same workspace again for this demonstration. It starts Eclipse; but, now, we should have the EGL development Tools available to us.

I open perspective to EGL. I can close the Java EE because I won't be using that perspective. I'll go ahead and open the Debug perspective, get ready for that.

I also want to check out the Remote Systems Explorer (RSE) perspective because Remote Systems Explorer doesn't have to be installed separately when I start out with the JEE bundle.

I'll create a new connection to Linux. I prefer "`localhost`" in all lower case. Click **Next**. We want "`ssh.files`" and **Next**, "`processes.shell.linux`" and **Next**, "`ssh.shells`" and **Next**, "`ssh.terminals`" (and **Finish**). So, it always uses secure shell (SSH).

I'll right-click and launch a terminal. I'll put in my correct user ID and password; but, it gives me a failed-to-connect.

This is because I want to demonstrate Mac OS X is not set up for Remote Login.

(Go to Launch Pad, System Preferences and then Sharing.)

Currently, it's off. I need to check this box to turn it on. I have to unlock the page first.

I'll check Remote Login. It says that it's blocked by firewall. So, I have to go to Security and Privacy, and then, to the Firewall tab and press the **Firewall Options** button.

Currently, the firewall is blocking all incoming connections, which is a very secure way to run Mac OS X. (Uncheck the Block all incoming connections option.)

And then, I'll enable stealth mode to be as secure as possible, even though SSH port 22 is open. (Check the Enable stealth mode option.)

Now, I'm back in my IDE. When I launch the terminal, this is a good thing because now the port's open and it has a fingerprint for OS X. I'll close the Remote Systems Explorer perspective.

This is another lesson in Essential EGL+Batch. Thanks for watching!

Virtualization

If you choose OS X, it might be possible to take EGL for a test drive using VMWare Fusion. Your virtual machine/operating system choices include:

1. Linux on OS X plus VMWare Fusion
2. Windows on OS X plus VMWare Fusion

Linux on Mac OS X

For Mac OS X:

1. Download and install VMWare Fusion.
2. Download the OpenSUSE 11.1 DVD ISO.

3. Burn DVD ISO to DVD-R disk.

4. Create a new virtual machine.

5. Follow instructions provided with VMWare to install the vmware tools for Linux, if necessary.

Caution: Focus on learning EGL. It is not necessary to upgrade your Linux to the latest and greatest kernel.

EDT on Windows

Procedure 1C: Installing EGL Development Tools on Windows

This procedure illustrates how to download and install EGL Development Tools (EDT) version 0.8 on Microsoft Windows XP.

For more information on installing EGL Development Tools on Windows, see also *Essential EGL+Batch - Installing EGL Development Tools on Windows* (video).

Transcript: Installing EGL Development Tools on Windows

Welcome to *Essential EGL+Batch*. In this series, we talk about the EGL Batch environment, and in this particular lesson, we're going to download and install the EGL Development Tools (or EDT).

What you want to do is to open your favorite web browser and in the location field you want to type in "`www.eclipse.org`", which brings you to the main page. And then, you want to select projects. And then, you want to select the list of all projects. Now, there's a lot of projects at the "`eclipse.org`" web site. We want to go to the section on tools. We want to find the section on tools, and then select the EGL Development Tools. So, then, we want the Tools Projects section and then EGL Development Tools. EGL Development Tools enables you to bring up an EGL+Batch environment.

Now, on the EGL Development Tools page, we want to go to Downloads.

On the Downloads page, we'll scroll down and we're looking for the all-in-one package for our platform. And in this case, we'll be installing EGL Development Tools on Windows (32-bit). We click on that link and it brings us to a page where we want to find a mirror that's close to our location on the Internet. And my favorite mirror is the Georgia Tech Software Library because I'm located near Georgia Tech. We want to save the file. I'm going to put it in MyDocuments. It downloads in about six minutes on my computer. I sped that up so you don't have to wait six minutes of video. So, the download is complete. There is the download right there.

Now, to be able to use this we want to be able to unzip it. So, we'll right-click and use the Extract All option, and walk through the wizard. Just keep the defaults and click on the **Next** button. So, it extracts the archive into a new directory called "`eglweb-081-win32`", in this case. I do not want to show the extracted files because I already have the Windows Explorer open. Now, I go into that directory. There's going to be an "`eclipse`" subdirectory. In the "`eclipse`" subdirectory, there's going to be an "`eclipse`" program. I'm going to right-click on the "`eclipse`" program and select Open and I get a message. Now, if you already have a Java Runtime Environment (or JRE) installed, you won't get this message. But, I wanted to show what this message was in case you run into it. We need a JRE in order to run the EGL Development Tools. So, how do we get the JRE? Well, let's see.

We'll go back to the browser and in the location, we'll type in "`www.java.com`". And on this page, we'll go to Downloads. What it's recommending from "`java.com`" right now is JRE version 7, but, we're going to actually look for JRE 6 and install that on Windows. So, select the Looking for Java 6 option. We'll scroll down to the Java 6 download page and we'll get the on-line version for our platform, in this case, Windows.

Again, I'll save it to MyDocuments. And, the download is complete. I'll go ahead an open MyDocuments. That's what I've got: a "`jre-6u38-windows-i586`".

I'll install the JRE. Now, the installer is actually a tiny installer. It downloads the rest of it as a separate step, versus the all-in-one installer. So, just like that, we should have Java installed. It shows a warning. We don't want the Ask Toolbar, in this case.

It doesn't take very long. Close the installer.

I'll go back into the "`eclipse`" folder and find the "`eclipse`" executable and open it. And this time, instead of a message I actually get the start-up. I'll select a workspace; that's the default. As you can see, the EGL Development Tools are now working.

This video is part of the Essential EGL+Batch series. Thanks for watching!

Windows

This operating system is a commercial product; but, it is often bundled with a new computer. Most, if not all, of the examples are compatible with EDT on Windows.

Virtualization

If you choose Windows, it might be possible to take EGL for a test drive using VMware Player. Your virtual machine/operating system choices include:

1. Linux on Windows plus VMware Player.

Linux on Windows

For Linux on Windows:

1. Download and install VMware Player.
2. Download OpenSUSE 11.1 DVD ISO.
3. For best results, burn the DVD ISO to a DVD-R disk.
4. Create a new virtual machine and install the OpenSUSE 11.1 operating system.

Settings: 1 processor, 2GB memory, 20GB virtual disk.

5. Follow instructions provided with VMware to install vmware-tools for Linux, if necessary.

Remote Systems Explorer

Overview

Install Remote Systems Explorer (RSE) in EGL Development Tools (EDT). This video features RSE version 3.3.1. The Install New Software feature works with both Linux and Windows.

Introduction

The Remote Systems Explorer (RSE) enables you to connect to a remote machine to view and transfer files, display processes and use a command line interface from a secure shell terminal.

Installing

When installing EDT on OS X, the Remote Systems Explorer is already installed as part of the JEE Development Tools.

On Linux and Windows, it is installed separately.

Transcript: Installing Remote Systems Explorer

Welcome to *Essential EGL+Batch*. This video series features EGL Development Tools version 0.8.

In this episode, we're going to install Remote Systems Explorer.

The first thing we need to do is switch to the superuser account because, in order to update Eclipse for all users, we have to be a superuser.

We switch to the "`/opt/eclipse.org/eglweb`" directory and start Eclipse with "`./eclipse`."

I'm going to use "`/root/workspace`" as my workspace just to install RSE.

With the workbench up and running, I need to pull down the Help menu and select Install New Software.

We need to add a repository and use the repository URL for Remote Systems Explorer ("`http://download.eclipse.org/tm/updates/3.3/`").

Once that is added, we can see the option TM and RSE 3.3.1 Main Features. We want to select that; that's what we want to install.

Next and **Next**. We read the software license, we read through all these legal terms, select I Agree and **Finish**. And then, Eclipse goes about installing Remote Systems Explorer. So, we're adding Remote Systems Explorer to the EGL Development Tools.

We need to restart the IDE. We **Restart Now**. It brings up the workspace.

I want to confirm that it is possible to switch to the Remote Systems Explorer perspective, make sure that we have RSE installed. There it is. I'm going to **Cancel** that and exit Eclipse.

Now, I need to switch back to my regular account. I'll switch directories to "`/opt/eclipse.org/eglweb`" again and start Eclipse with "`./eclipse`". I'll go to the default workspace for me.

I'm going to go ahead and switch perspectives now to Remote Systems Explorer perspective. By default, we'll get a Local connection, which does not include a terminal.

I want to add a new connection. So, I right-click and do New | Connection. We want a Linux connection. We're going to add a connection to "`localhost`". I prefer to have "`localhost`" spelled out in all lower case. Yes, we're going to verify the host name.

I include "`ssh.files`" and **Next**; "`process.shells.linux`" and **Next**; "`ssh.shells`" and **Next**; "`ssh.terminals`" and **Finish**. So, it always uses secure shell (SSH). That configures my connection to my local Linux.

To connect, I right-click on Ssh Terminals and sign in. Here, I have a connection. This connection is a terminal so I can do things just like I do in another terminal emulator. In this case, I'm going to remove some stuff that was left over from installing EGL Development Tools before.

This is another episode in *Essential EGL+Batch*. Thanks for watching!

Introduction to Eclipse

Eclipse

EGL Development Tools 0.8 is an extension of Eclipse. If you are familiar with Eclipse-based tools, you already know much of how to navigate and work with EDT 0.8.

For an essential introduction to Eclipse, see also *Introduction to Eclipse*, with a presentation and exercises, in this series.

Download Introduction to Eclipse

The following procedure illustrates how to download the Introduction to Eclipse presentation, self-study guide and exercise.

1. Open your favorite web browser;
2. Type "`http://gchii.blogspot.com/2013/03/essential-eglbatch-introduction-to.html`" and press the Enter key.
3. To download the presentation, follow the "`1 Introduction to Eclipse`" link.
4. To download the self-study guide, follow the "`1.0 Introduction to Eclipse`" link.
5. To download the exercise, follow the "`1.1 Introduction to Eclipse`" link.
6. Save the file to the "`/lab/zip`" folder (or, if you prefer, to your Desktop).

Archive File

Introduction

Exporting and importing an archive file is a feature of Eclipse. Use these features to

- ✓ Share source code with others,
- ✓ Copy source code from one workspace to another, and
- ✓ Back up and restore your work.

Non-Requirements

You do not need an external tool. Eclipse provides

- ✓ An import feature for importing files from a ZIP-compatible archive into your workspace, and
- ✓ An export feature for exporting files from your workspace to a ZIP-compatible archive file.

Downloading Source Code

The following procedure illustrates how to download the EGL and Java source code from the Internet.

1. Open your favorite web browser;
2. Type "`http://gchii.blogspot.com/2013/02/essential-eglbatch-source-code-1-thru-22.html`" and press the Enter key.
3. Follow the "`eglprogram-22.zip`" link.
4. Save the file to the "`/lab/zip`" folder (or, if you prefer, to your Desktop).

See also

The following presentation demonstrates both how to export and import. To import the "`eglprogram-22.zip`" archive, see also "Importing from an archive file" in this presentation.

Transcript: Archive File

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

We've been working with the "`eglprogram`" project for a while; so, we have a number of different EGL packages from lessons 1 "`lab.lesson01`" through 22 "`lab.lesson22`". We also have a couple of Java packages from lessons 19 ("`lab.lesson19`") and 21 ("`lab.lesson21`").

Exporting to an archive file

To preserve our work, we're going to export the project. We right-click on the project and select the Export option. We go to the General folder and select the Archive File option.

In this case, what we're going to do is first select everything and then eliminate some of the things we don't need. We don't need the project information. We don't need the settings. We don't need the "`EGLbin`" folder. We don't need the "`bin`" folder, which is the Java "`bin`" folder. And yet, we don't need the "`generatedJava`" folder either because, when we import the archive file, the EGL Development Tools will generate the Java source code for us.

What do we have left? We have lessons 1 through 22, packages for EGL. And we have lessons 19 and 21, for Java source code.

The name of our archive file is "`/lab/zip/eglprogram-22.zip`". It's "-22" because we have twenty-two lessons.

We're going to save in the ZIP-compatible file format. We're going to compress the contents of the file. And we use the option "Create directory structure for files". Press the **Finish** button. If the file already exists, we press the **Yes** button because we want to overwrite it.

Importing from an archive file

***Important:* Now, we switch our new workspace.**

I want to show you what happens if we try to import our archive file without a project. We select an archive file and a message will appear that says, "Cannot import into a workspace with no open projects." So, that won't work.

The first thing we have to do is create a project; but, the kind of project we're going to create is not an EGL project. We're going to create a new project. We expand the General folder and select the Project option. That's a plain project, a simple project.

The name of the project is going to be "`sandbox`". We're going to use the default location. We're not going to add the project to a working set. Press the **Finish** button. So, this project is basically an empty project.

We want to right-click on the "`sandbox`" project and import a ZIP-compatible archive. When we select the archive file, we can get all of the things that are in the archive.

On the left, this is a list of things that are in the archive. We have lessons 1 through 22 in the EGL source code. We have lessons 19 and 21 in the Java source code. This is good. We do not want to overwrite existing resources. Press the **Finish** button.

Again, the "`eglprogram`" folder here in the "`sandbox`" project is just a folder, and not a project.

We need to create a new EGL project in our new workspace. The project name is going to be "`eglprogram`". We're going to use the default location for the project. It's going to be a Basic project. Press the **Next** button.

We're going to use the EDT Compiler. We're going to override generation settings. We're going to disable the JavaScript Generator. We're going to leave just the Java Generator. And, the Java Generator is going to generate to the "`generatedJava`" folder. That is why we don't have to have the "`generatedJava`" folder in our archive file. Press the **Next** button.

Press the **Finish** button. That creates our EGL project.

With a new EGL project, there's not going to be any source code for EGL. There's not going to be any source code for Java.

How do we get the source code? The easiest way to do this is with a view that's called the Navigator. We're going to pull down the Window menu and select the Show View | Other option. Expand the General folder and select the Navigator option. Press the **OK** button.

The Navigator view shows files as they really are on disk. The Project

Explorer view shows a more abstract view for EGL development.

The first thing we want to do is find the Java source code and the "`lab`" folder. We're going to copy the "lab" folder. (Right-click on the "`lab`" folder and select the Copy option.) With just one paste, we're going to paste the "`lab`" folder and all of its subdirectories in the "`src`" folder for our project.

We're going to do the same for our EGL source code. (Expand the "`EGLsource`" folder in the "`sandbox`" project. Right-click on the "`lab`" folder and select the Copy option. Right-click on the "`EGLsource`" folder in the "`eglprogram`" project and select the Paste option.)

Now that we have the information imported into our project, we no longer need the "`sandbox`" project; so, we can delete it. (Right-click on the "`sandbox`" project and select the Delete option. Check the "Delete project contents on disk" option and press the **OK** button.)

To continue to work with the EGL project, we want to go back to the Project Explorer view. There's our source code for EGL and the packages. We want to go to "`lab.lesson20`", open up the "`Lesson20.egl`" file and take it for a test drive. We want to make sure the software still works. And, it does!

Caution: While it is possible to import an archive file directly into an EGL project, I do not recommend it for a beginner. Instead, I demonstrate a technique that won't destroy your work by accident.

Caution: Use caution when using the import and overwrite existing resources. There is no easy **Undo**. Make sure you have a backup copy of your work *before* you perform such an import.

This has been another lesson in Essential EGL+Batch. Thanks for watching!

Essential EGL Topics

Lessons

The following lessons present essential EGL topics, featuring EGL Development Tools.

1. Program
2. Library
3. Library Variable
4. Date
5. Timestamp
6. BatchLib
7. Character
8. Integer
9. Float
10. Delegate (Bonus Video)
11. Record
12. Fixed-Length Record
13. Nested Record
14. Record Array
15. Exception
16. Log Exception
17. System Properties
18. Custom Properties
19. External Type - JavaSystem
20. Delegate Field (Bonus Video)
21. Arguments
22. Connection Properties
23. SQL Drop and Create
24. SQL Delete, Insert and Update
25. SQL Select
26. SQL Prepared Insert
27. SQL Prepared Select

28. SQL Export
29. SQL Insert
30. Dictionary (Bonus Video)

Procedure 1: Prerequisite for Lesson 01

1. Download and install EDT 0.8.x.
2. Start EDT 0.8.x.

Result: The workspace dialog box is displayed.

2. Create (or select) a workspace.

Caution: Do not use the feature called “always use this workspace”.

3. The name of the workspace used in this series is “`/lab/eglbatch`”. For Windows, this is “`C:\lab\eglbatch`”. For Linux, this is “`/lab/eglbatch`”. For OS X, this is “`~/lab/eglbatch`”. You can work through all of the lessons in this series without creating a new workspace.

4. Creating an EGL project. In the lessons, we use “`eglprogram`” for a project name. You can work through all of the lessons in this series without creating a new project.

5. EGL package names. In the lessons, we use “`lab.lesson`*`NN`*” for a package name, where NN is the lesson number.

Lesson 01 Program

Overview

Create, run and debug EGL program.

Background

A *program* is primarily a collection of variables and functions. To define a program, we start with the keyword "`program`" and end with the keyword "`end`". To define a function, we start with the keyword "`function`" and end with the keyword "`end`".

A program is one of a few dozen kinds of EGL parts. Later, we'll create other kinds of parts, such as "library", "record" and "service".

Running a program

In this lesson, we demonstrate how to run a program from within Eclipse. Right-click somewhere in the editor of an open program and select the Run As | EGL Main Application option. Or, right-click on a program in the Project Explorer view and select the Run As | EGL Main Application option. In either case, Eclipse launches the program as a separate process of an operating system, as if it were launched from a command line.

The standard output stream and standard error stream of a running program is connected to the Console view of Eclipse. Enabling the build automatically feature and often using Run As | EGL Java Main Application, you can develop an EGL program progressively, checking your work as you go. You should always have a runnable program.

In this lesson, we'll demonstrate how to create a Runnable JAR file and run a program outside of Eclipse. From outside of Eclipse, use the "java" command and the "-jar" command line option to run a program assembled in a Runnable JAR file.

Debugging a program

In this lesson, we demonstrate how to debug a program from within Eclipse.

With the EGL debugger, you can find and fix a defect in your program. Right-click somewhere in the editor of an open program and select the Debug As | EGL Main Application option. Or, right-click on a program in the Project Explorer view and select the Debug As | EGL Main Application option. In either case, Eclipse launches the program as a separate process of an operating system and attaches a built-in EGL debugger to the running process.

The Debug perspective provides views for debugging a program. Use the **Resume**, **Terminate**, **Step Into**, **Step Over** and **Step Return** buttons while your program is suspended. Use the **Suspend** and **Terminate** buttons while your program is running. Examine variables in the Variables view.

The minimum requirements for an EGL program is any kind of project, including general EGL project, an EGL+dynamic web application project, an EGL+Web Service Provider project, a EGL+JSF project.

Transcript: Lesson 01 Program

Welcome to *Essential EGL+Batch*. In this lesson, we're going to

- ✓ Write an EGL program;
- ✓ Run the EGL program;
- ✓ Debug the EGL program;
- ✓ Run the EGL program as a Java application; and also
- ✓ Create a Runnable JAR file so that we can run the program without using the EGL environment. We just need a Java Virtual Machine.

So, first, what we're going to do is see the completed EGL source code. The EGL source code is in a project called "`eglprogramref`"; and, it's lesson one. So, today, this is the program that we're going to write. It just writes out a string "`Lesson 01 - 0.0.0`". So, I'm going to copy the text of the lesson for today.

So, first, we need to open a new project; we need to create an EGL project. The project name will be "`eglprogram`". We're going to use the default location for the project in the workspace. And, this is going to be a basic project.

Now, we have a choice of compilers. The one that comes installed is EDT Compiler. We're going to override the generation so that it's just going to produce the Java code. The Java code will go in a folder called "generatedJava". And then, we finish; and now, we have a new project.

Now, the project comes with an EGLSource folder, an EDT Systems Library folder, and a source ("src") folder. Plus, it has a JRE System Library folder. And, those are the folders that come with the project when you start.

So, what we're going to do first is we're going to create an EGL package so that we have some place to work with. So, we want to create a new package. The package name will be "lab.lesson01". And that's created a new package.

And then, we right-click on the package. We want to create a new program. The new program is going to be called "Lesson01". And, by default, what happens is EGL puts some comments and puts some code in here for a brand new program. We're going to override that with the program that we already wrote, and paste that into the editor.

Listing: `lab/lesson01/Lesson01.egl`

```
package lab.lesson01;

program Lesson01

  function main()
    SysLib.writeStdout("Lesson 01 - 0.0.0");
  end
end
```

So, what do we have? We have a package statement. The package statement has to match the EGL package that we created. The package is "lab.lesson01", all lower case.

The name of our program is "Lesson01". As you can see, it has one function called "main".

What's going to happen when we run the program? The infrastructure for EGL is going to call the "main" function. The "main" function is going to use the "SysLib" library to write out the text.

Technically speaking: In a command-line environment, including EGL+Batch, a program always has three standard streams: input, output and error. The standard output stream is for messages and other data; the "`writeStdout`" function writes to this stream. The standard error stream is for error messages and bad data; the "`writeStderr`" function writes to this stream. From within Eclipse, the output and error streams are redirected to the Console view when we launch a program.

Listing: Lesson 1 Console

```
Lesson 01 - 0.0.0
```

There, in the Console view, you see the text right there, "`Lesson 01 - 0.0.0`". So, we've run our very first EGL program!

Now, when we typed in the program, we are going to get a generated Java file. We'll talk about that a little bit more in a moment. That's what's actually running.

Now, we're going to use the Java debugger and we're going to be able to debug our EGL program. Right-click somewhere in the program and Debug As | EGL Java Main Application.

And what it's going to do since we started the debugger and put in a breakpoint, it's going to ask us to go to the Debug perspective, which is, Yes, that's what we want to do.

In the Debug perspective, this is similar for all of the Java debugging. It gives the name of the program, the launcher, the thread and the very line that we're on.

In the debugger, you can see all of the variables that are used in your EGL program. We don't have any variables in this program.

It shows you a list of breakpoints, which you can turn on and off with a checkbox.

The **Resume** button runs the program at full speed. **Suspend** enables you, while it is running, to pause it. The **Terminate** button enables you to stop it. The **Step Into**, **Step Over** and also **Step Return** buttons let you step through your program.

In this case, we're going to use the **Step Over** button. And, as you can see, that one line of code outputs to the console the text.

Resume this. It ran all the way to the end; and now, it's done, it's finished.

We'll go back to our EGL perspective. And now, we're going to go and run this at full speed in the generated Java. So, it has a corresponding Java file.

Now, in the EGL package, it was "`lab.lesson01`", well the Java package is going to be "`lab.lesson01`". The name of our program was "`Lesson01`". The name of our main function is "`main`".

This is the Java code that's generated. To run it at full speed as a Java application, we go to the Java file and there it is. Full speed means that it is going to run the fastest as a Java program.

Now we could actually debug the application as a Java program; but, instead of seeing the original EGL, as we did in the EGL debugger, you would see the Java code, which is not very useful, the generated Java code.

Now, we can export this as a Runnable JAR file, now that we've run it once as a Java application. It has to have a launch configuration to use. The launch configuration includes all of the information for the JARs and the dependencies. It's going to include all of that in the exported JAR. We export it to "`eglprogram.jar`". As it says, it's going to include all of the libraries. If you've done it more than once, you're going to overwrite the JAR. So, it exports everything that's needed into a single JAR file.

Now, we going to go to the Remote Systems Explorer and get a terminal. In the terminal, we'll be able to run, because we're running this on Linux. It is very similar in Windows. I'll bring open the terminal a little bit.

We want to go to where we output the JAR. There it is, "`eglprogram.jar`". And run it with the jar option. So, it's java space hyphen jar space and the name of our jar.

Listing: Lesson 1 Terminal

```
$ cd /lab/zip
$ java -jar eglprogram.jar
Lesson 01 - 0.0.0
```

And there you can see, it's output the same line of text. So, it's running at full speed without the EGL environment. It just requires a Java Virtual Machine.

So, this has been a lesson in *Essential EGL+Batch*. Thanks for watching.

Lesson 02 Library

Create, run and debug an EGL library.

Background

An EGL library is a collection of variables and functions. To define a library, we start with the keyword "`library`" and end with the keyword "`end`".

A library is reusable. It can be defined in a Batch project and then used by other EGL projects. It can be used by a program, web service and web page (in a JSF handler).

As a matter of convention, all business logic is written into libraries. A library may provide functions to access a database. It may use other libraries.

What is the difference between a library and a program? While a program can be invoked directly from a command line, a library cannot. A program is part of the presentation layer, part of the user interface. As a matter of convention, it is only a thin wrapper. It contains no business logic. It uses functions in a library.

Starting with this lesson, we are going to put our examples in a library and use a program as a thin wrapper for a library, as it should be done in a production-quality application.

Caution: While it is technically possible for a program to use functions in another program, this is not recommended.

To define a function, we start with the keyword "`function`" and end with the keyword "end". Within the function, we write what the function is going to do.

Preview

Lesson 2 is an introduction to using a library in a program. Lesson 3 defines a common library used by a collection of programs.

Goal

At the end of these lessons, we'll have a useful framework for building EGL

+Batch programs. Every part of the framework is going to be explained and written into a reusable library called "`BatchLib`". Therefore, the focus of lesson 2 is something actually used in `BatchLib`, such as a library part. I use the `private`, `const` and `in` keywords because I believe that it is important for professional source code.

In a production program, I would not write `StdLib.writeStdout("Lesson 01 - 0.0.0").` Rather, I would write something like this:

```
const PROGRAM_NAME string = "Lesson 01";
const VERSION string = "0.0.0";
StdLib.writeStdout(PROGRAM_NAME :: " - " :: VERSION);
```

Transcript: Lesson 02 Library

Welcome to *Essential EGL+Batch*. This series features EGL Development Tools version 0.8.

In this lesson, we're going to introduce a built-in type called `string`, modifiers `const` and `private`, write and call an EGL function, plus pass parameters to a function. We're going to write an EGL library. And finally, we're going to find and fix a bug in the EGL library.

We're going to reuse the basic EGL project from lesson one called "`eglprogram`". See lesson one for instructions to create a basic EGL project.

Let's take a look at the program and library that we're going to create during this lesson. We'll take a quick look. The program is "`Lesson02`". It has a call to a function called "demo". The function is defined here. In the library—notice it's a library and not a program—we've defined two variables. We've defined two functions, one "`startFunction`" and one "`endFunction`". And that's pretty much it for the library.

In order to work on lesson 2, we need to have a package. So, let me show you a hard way to create the new package for lesson 2. We do right-click, New | Package and we type in "`lab.lesson02`". Ordinarily, we would press the **Finish** button to create the package; but, we're going to show you a shorter way.

We right-click on an existing package and do New | Package. It provides us with the name of an existing package, which we can edit. Press the **Finish** button. And now, we have our new package.

Now, let's go get the source code from our reference project. We'll highlight the library and the program source code. We'll select that. And then, right-click and Copy. And then, we'll be able to go to our "`eglprogram`" and the package. Right-click on the package and select the Paste option. So, it copied the library and the program to our lesson for today.

Listing: `lab/lesson02/Lesson02.egl`

```
package lab.lesson02;

program Lesson02

   function main()
      SysLib.writeStdout("Lesson 02");
      demo();
   end

   private function demo()
      functionName string = "demo";
      BatchLib.startFunction(functionName);
      BatchLib.endFunction(functionName);
   end
end
```

Now, what do we have? Let's go through this a little bit. We have the package which is has to match the name of the package. We have a program. We have a `main` function, like we learned in lesson 1. And, we have a `demo` function. This is where the `demo` function is called from the main function. That is the syntax without any parameters.

Notice we have a new construct here, which is the name of a library and dot. That enables us to use code and variables from a library.

Listing: `lab/lesson02/BatchLib.egl`

```
package lab.lesson02;

library BatchLib
   private const PREFIX_START string = "START ";
```

```
    private const PREFIX_END string = "END ";

    function startFunction(name string in)
        message string = PREFIX_START :: name;
        SysLib.writeStdout(message);
    end

    function endFunction(name string in)
        message string = PREFIX_END :: name;
        SysLib.writeStdout(message);
    end
end
```

Just like a program, it has to have the same package name. In this case, we have two constant variables. The word "`private`" means that it is only available in this library. The keyword "`const`" means that it cannot be modified. And then, we have "`startFunction`" and "`endFunction`".

For parameters, here is the name of a parameter. The type is string. And "`in`" means that it is read-only; it cannot be modified.

We're defining a variable called "`message`". It's also the type "`string`". This is a construct for concatenating two strings. We'll take the variable "`PREFIX_START`" and the parameter "`name`", and we'll put it together in a string called "`message`". The string concatenation operator is actually a colon-colon (::). The equivalent is the plus sign (+). But, the plus sign is for math, and the colon-colon is only for string concatenation, putting two strings together.

We have a very similar thing with the "`endFunction`". But, it uses the "`PREFIX_END`" instead of "`PREFIX_START`".

So, what is this going to do for us? Let's run the program and see what happens.

Right-click somewhere in the editor. We do Run As | EGL Java Main Application. We find our console. That's where the output from the program goes.

I'll open that up and see what we have. This is where it printed out "`Lesson 02`", and then printed out "STARTdemo" and "ENDdemo".

Oh, but, it seems that there is a space missing between the word "START" and the word "demo". Now, how did that happen? What we need to do is debug our program. So, we'll right-click somewhere in the editor. Debug As | EGL Java Main Application.

You'll notice that it also goes through very quickly because we haven't set any breakpoints. It produces exactly the same output as Run As. Without a breakpoint, our debugger is never engaged. We never go to the Debug perspective.

So, I'm going to set a break point, right here where the `demo` function is called because that's pretty close to where we have a problem. I'll debug. This time with a breakpoint we get this message. Yes, we want to switch to the Debug perspective.

We're on line six where the `main` function is going to call the `demo` function. We have buttons up here. In this case, it is going to be important that we want to **Step Into** the demo function. If we use **Step Over**, it would run the entire demo function and still be in the main function when we're done. Here's the function name. Let's see. Now, we have some variables to look at. We'll look in the debugger for them.

Let's see. Let's see if we can find the function name. That's actually function, but it's without any variables. We use **Step Over**. Now, there is "`functionName`" and it's value is "demo".

The text of the value is right here. You can actually type in something else if you want to; but, it has no effect on the program at this point. If you hit the Enter key, it just moves the text up because this is an editor, a multiline editor.

If you want to copy the value of a variable into your EGL program, you can highlight it, right-click and Copy. So, you can use the values of variable that are in the Variable view.

So, we haven't run into the problem yet. Let's go ahead and **Step Into** the "`startFunction`". Notice that it switches to the "`BatchLib`" library and we're right here in the middle of the "`startFunction`". Notice that now "`PREFIX_END`" is defined with the text "END", and "`PREFIX_START`" is defined with the text "START".

In the "`startFunction`", we have "`name`", which is actually the parameter. That is the only variable so far.

Let's **Step Over**. And then, we'll be able to find the "`message`". We see right here that, ah! there is no space between "START" and "demo". How is that?

Let's see. Ah! there is no space at the end of the prefix ("`PREFIX_START`"). Notice that there's no space at the end of "`PREFIX_END`" as well. So, we can fix two bugs with one debugging session.

(With the **Resume** button,) I'll run this to the end.

Save the changes that I made in the debugger and we'll start the debugger again. **Yes**, we want to go to the debugger.

We'll start right back over at "`demo`" again. **Step Into** that. We want to **Step Into** the "`startFunction`" again to see if we fixed the problem. Let's see what the "`message`" is now: "START demo". It looks like it might work.

Let's see what comes out on the console: "START demo". It looks like it worked.

We can **Step Over** the "`endFunction`". It has also fixed a bug with the "`endFunction`".

It looks like our program is working now. We can run it as an EGL Java Main Application. Notice that the breakpoints are ignored.

This is the end of Lesson 2. Thanks for watching!

Lesson 03 Library Variable

Create an EGL library. Use the debugger to see how a variable of a library is used.

Transcript: Lesson 03 Library Variable

Welcome to *Essential EGL+Batch*. This series features EGL Development Tools version 0.8.

Let's take a look at the finished product for today's lesson, Lesson 3.

Listing: `lab/lesson03/BatchLib.egl`

```
package lab.lesson03;

library BatchLib
   private const PREFIX_START string = "START ";
   private const PREFIX_END string = "END ";

   private functionName string?;

   function startFunction(name string in)
      functionName = name;
      message string = PREFIX_START :: functionName;
      SysLib.writeStdout(message);
   end

   function endFunction()
      if(functionName == null)
         return;
      end
      message string = PREFIX_END :: functionName;
      SysLib.writeStdout(message);
   end
end
```

Listing: `lab/lesson03/Lesson03.egl`

```
package lab.lesson03;

program Lesson03

   function main()
      SysLib.writeStdout("Lesson 03 - 0.0.0");
      demo();
   end
```

```
  private function demo()
    functionName string = "demo";
    BatchLib.startFunction(functionName);
    BatchLib.endFunction();
  end
end
```

In today's lesson, what we're going to do is we're going to improve upon the "`BatchLib`" library we wrote in Lesson 2. What we're going to do is enable the calling program to set a function name, and then, in the "`endFunction`", we're not going to pass the parameter; we want the library to actually save the function name.

In order to do this, we're going to create a "`functionName`" variable, set it in the "`startFunction`". So, this "`functionName`" variable will actually be part of the library. And then, we'll be able to use it in the "`startFunction`" and the "`endFunction`."

The first thing we want to do is be able to create a package in our "`eglprogram`" project. So, we're going to create a new package called "`lab.lesson03`". And inside of "`lab.lesson03`", we're going to create a new program called "`Lesson03`". Press the **Finish** button. EGL Development Tools creates a program with comments and a couple of things for us. We're just going to leave the function "`main`" for right now. I'll save that.

Let's take look at how to create a library. We right-click on the package and we want to do a New | Library this time. The name of the library is going to be called "`BatchLib`". And yes, we're using the same name that we were using in Lesson 2.

Similar to a program, when you create a new library, EGL creates a bunch of things for you. This time, we not really going to use anything that EGL created.

This is the library from last time. What we want to do is fix up this library. Let me select it and copy it to the clipboard.

I'll go back to our new "`BatchLib`" library. I'll Select All, select all this text, and then, Paste in the library we had last time.

This is the library we had last time. It was still lesson 2, so we need to change that, fix it up, so that it's part of lesson 3. This is our package name. This is our library name. Inside of our library, we have the "PREFIX_START" and "PREFIX_END", just like we had before.

Tip: The right-click and Close Others enables us to close everything but the current editor.

What I want to do is take out this parameter. As soon as I take out the "name" parameter, it says that the name cannot be resolved. What we're going to end up with is something like "functionName". Of course, "functionName" is also a variable that's not defined. So, let me Undo that.

Let's add a variable called "functionName". Make it private. Make it string type, a string type with a question mark, which means that it's nullable. We're going to declare that outside of any function so that it's part of the library.

If I type in "fu" and press Control+Space, it's an interesting thing that the editor pops up a list of all the available things that starts with "fu".

If I type in "na" and press Control+Space, there's only one thing that starts with "na" so it fills it in immediately.

Let me fix up the names of these variables to be "functionName" instead of the parameter "name".

The "startFunction" sets the "functionName"; the "endFunction" uses it. There's just one thing. What happens if someone calls "endFunction" before they call "startFunction"? I'm going to put in a test here that "functionName", if it's null, I'm going to go ahead and return.

Oh, and let me do this again. Let me type in this in a little bit more slowly. When I return from a function, it's just the return statement like that.

Now, watch what happens when I type "end". I didn't move the keyword "end" to the left; the editor does that for me. The semicolon (;) is not required on the "end" keyword because it's a structural element and not a statement.

I can format the source code with EGL Source | Format.

So, that's what we have now. We have a library called "BatchLib". It has, part of the library, a variable called "functionName". So, "endFunction" no longer needs a parameter; it's going to use whatever is set when "`startFunction`" is called.

Let's see how we use this in our Lesson 3 program. Now, watch what happens when I type in "`BatchLib`" and just hit the dot (.). It comes up with a list of things that I can use. Control+Space will bring it up, this list for quick entry, also the dot.

If I type very quickly, you'll notice the dot does not bring up the choices. If I put in a dot and wait, then it come up with choices. Then, I can pick a function out of the library. That works with both custom libraries and built-in libraries from EGL, like "`SysLib`".

I'm going to put together, exactly like Lesson 2, a function called "`demo`". Notice again, when I type in the keyword "`end`", it puts it in the right place for me; I don't have to do that.

I'm going to type a dot and pick "`startFunction`". I'll use the "`functionName`", which I haven't defined yet.

I'll put the dot, and this time, I'll select the "`endFunction`" with the Enter key.

Now, I'll put in "`functionName`" as a string. It's equal to "`demo`".

I'll save all these changes and what do I have? We have a program that's going to start out at "`main`" and then "`main`" is going to call "`demo`". In turn, "`demo`" is going to be calling the library.

It's pretty obvious that Lesson 2 and Lesson 3 programs cannot be confused. But what about "`BatchLib`"? We have two different libraries with the same name. The distinction between the two is the package. You'll notice that this package is "`lab.lesson02`"; so, this is the Lesson 2 "`BatchLib`". This package is "`lab.lesson03`"; so, this is the Lesson 3 "`BatchLib`". So, these are not the same library. We still have Lesson 2 library, and it still requires a "`name`" parameter in the "`endFunction`". And then, we have Lesson 3 "`BatchLib`" library, which doesn't require a parameter in the "`endFunction`". Let me close everything but the "`Lesson03`" program.

We'll run this with Run As | EGL Java Main Application, just as we have been doing. What do we get? Let's take a look at the console.

Listing: Lesson 3 Console

```
Lesson 03 - 0.0.0
START demo
END demo
```

Look at this. It is not surprising that “START demo” is coming from the “`startFunction`”. Where did the “demo” word come from in the “`endFunction`”? It is not passed as a parameter.

Let’s set a breakpoint here on “`startFunction`”. We’ll debug this program to see where the function name is being stored. (Right-click on the editor and select the Debug As | EGL Java Main Application option.)

Yes, we want to switch to the Debug perspective.

Under variables, you’ll notice that the “`BatchLib`” library has not yet been initialized; that’s why it says “null”. In the “`demo`” function, we have already passed where it assigns the function name to “demo”. We’re about to call “`startFunction`”. So, I’ll **Step Into** that.

Here, it’s going to initialize the “`BatchLib`” library first, before we can **Step Into** the “`startFunction`”. This is the “`BatchLib`” library. Notice that, after it is initialized, the “`functionName`” is still null. “`PREFIX_END`” is “END ”; “`PREFIX_START`” is “START ”; but “`functionName`” is null.

This is the “`startFunction`” of the “`BatchLib`” library and the “`name`” parameter is “`demo`”. Let me scroll up to where it says “`functionName`” and watch what happens when I **Step Over**. You’ll notice immediately in the Variables view, it shows you the new value. And it also puts it in yellow to say that it was recently changed.

The `message` is written out, which is no surprise. It shouldn’t be a surprise that it writes out “START demo”.

Let me go back. Watch how it has no parameter; but, we’ll **Step Into** “`endFunction`”. Where does it get the function name? How does it know the function name?

Between calls to the “`BatchLib`” library, notice that the “`functionName`”, which is part of the library, retains its value. So, this particular program, as we're running, variables that are part of the library retain their value. That's where the value "`demo`" comes from. (By pressing the **Resume** button,) I'll just finish this program. I'm done debugging. So all of the functions that are in the library can use a variable that is part of a library.

This has been another lesson of Essential EGL+Batch. Thanks for watching!

Lesson 04 Date

Get the current date or set a specific date. Format and print a date.

Transcript: Lesson 04 Date

Welcome to *Essential EGL+Batch*. In this series, we feature EGL Development Tools version 0.8.

Today's lesson is all about dates. This is lesson 4.

Let's take a look at the code that we're going to end up with at the end of lesson 4. We have two things that we're going to build.

Listing: `lab/lesson04/Lesson04.egl`

```
package lab.lesson04;

program Lesson04

   function main()
      SysLib.writeStdout("Lesson 04");
      DateLib.demo();
   end
end
```

One is the lesson 4 program. We've seen this before in this series. It's a program called "`Lesson04`" with its usual function called "`main`". That's where all of the action starts. And then, "`main`" is going to call the "`demo`" function inside the "`DateLib`" library.

Listing: `lab/lesson04/DateLib.egl`

```
package lab.lesson04;

import lab.lesson03.BatchLib;

library DateLib

   function demo()
      goOnDate();
   end

   private function goOnDate()
```

```
        BatchLib.startFunction("Go On Date");

        today date{};
        displayDate(today);

        november11 date = "11/11/2011";
        displayDate(november11);

        december12 date = "12/12/2012";
        displayDate(december12);

        BatchLib.endFunction();
    end

    private function displayDate(d date in)
        s string = StringLib.format(d, "yyyy-MM-dd");
        SysLib.writeStdout(s);

        t string = StringLib.format(d, "MM/dd/yyyy");
        SysLib.writeStdout(t);
    end
end
```

Let's take a look at the "`DateLib`" library. Its package is "`lab.lesson04`". It's definitely a library. The public function is called "`demo`"; that's where all of the action starts. We have a function called "`goOnDate`". To mark the beginning and end of the "`goOnDate`" function, we use the "`BatchLib`" library's "`startFunction`" and "`endFunction`".

The "`displayDate`" function is called multiple times. It formats a given date. We'll get more of that in detail in a little bit.

To set today's day, we'll use a date type.

What we'll do first is we will create the necessary package for Lesson 4. We right-click on a package and do New | Package. We've seen this several times in this series. And it's "`lab.lesson04`".

For this lesson, what I'm going to do is copy the source code, and then, paste it in under the package for the "`eglprogram`" project.

Let's take a look at what we have. We have a "`Lesson04`" program and the "`DateLib`" library.

One of the things that I'm going to do is use the right-click and Open On Selection. So, if we put our cursor in a function name, and then, right-click and select the Open On Selection option, it will show us the source code. That's even if the source code is in another package.

I would also like to demonstrate this too because nested functions, functions calling other functions, you can kind of get lost.

There is a **Back** button that takes you back to where you used the Open On Selection feature. There is also a **Forward** button. We can go all the way back to "`main`" and we can go all the way forward to where we were in the "`startFunction`". You can basically go back to where you used the Open On Selection.

Let's talk about getting today's date. The default is the date type. Open curly braces and close curly braces initializes the date to today. So, that's how we get today's date.

The "`displayDate`" function will display that given date in two formats. It accepts a date parameter and "`date`" is read-only within this function. This is the first call to the "`format`" function, which is under "`StringLib`" standard library in version 0.8.2. The "`StringLib`" library has a bunch of different types that we are allowed to use with the "`format`" function. Unlike earlier versions of EGL, the "`format`" function is overloaded now, instead of having different names for the format function based on type. The result of a format is a string so we can print it out on the console.

Here, we'll format the date a little bit differently. Every time we call this function, it is going to format the given date twice and then print it out on the console.

We'll go back to where we were in the "`goOnDate`" function.

To set a date to a specific date, we use this construct here of the date variable is equal to something. It has to be in this format to set a date. You can do this like we have here. We've literally hardcoded it; but, this could be a value out of a database. The "`startFunction`" and "`endFunction`" will show us where the "goOnDate" function has been called.

So, that's all there is to the "`demo`" function.

Let's run this program and see what we get. We want to right-click, Run As |

EGL Java Main Application.

Listing: Lesson 4 Console

```
Lesson 04
START Go On Date
2013-01-25
01/25/2013
2011-11-11
11/11/2011
2012-12-12
12/12/2012
END Go On Date
```

Let’s take a look at the console. This is lesson 4 output. This is the start of the “`goOnDate`” function. And then, “`displayDate`” is called. This is the date displayed in that format. Let’s take a look at that formatting string: “yyyy-MM-dd”. So, it’s “yyyy” for the four-digit year, “MM” for the two-digit month and “dd” for the two-digit day. You can put that in any order. You can put different literals in there so that you can format the date as you choose. Or, you can have no literals at all and it will come out as a number, a bunch of digits.

The second date was 11-11-11. The third date was 12-12-12.

This is another lesson in *Essential EGL+Batch*. Thanks for watching!

Lesson 05 Timestamp

Get the current timestamp. Format and print a timestamp. Convert usage from RBD to EDT.

Transcript: Lesson 05 Timestamp

Welcome to Essential EGL+Batch. In this lesson, we're going to talk about the timestamp. This series is based upon EGL Development Tools version 0.8.

First, let's take a look at the finished source code. This is lesson 5. The program is very similar to program's we've already written; but this time, it uses the "`TimestampLib`" library.

Here is the "`TimestampLib`" library. There are a couple of things in the "`TimestampLib`" library that we're going to work with. That's basically the source code. Let's get going.

We'll copy the source code into the clipboard. We'll go to our "`eglprogram`" project, create a new package and paste in the source code.

The name of the package is "`lab.lesson05`".

I right-click on a package and select the Paste option. Okay.

So, what do we have? Let's take a look at the program.

Listing: `lab/lesson05/Lesson05.egl`

```
package lab.lesson05;

program Lesson05

  function main()
     SysLib.writeStdout("Lesson 05");
     TimestampLib.demo();
  end
end
```

We've seen this before. The only thing that's different between this and lesson 4 is that we are using the "`TimestampLib`", instead of the "`DateLib`" library.

Let's look into the code.

Listing: `lab/lesson05/TimestampLib.egl`

```
package lab.lesson05;

import lab.lesson03.BatchLib;

library TimestampLib
   private const TIMESTAMP_DISPLAY1 string = "yyyy-MM-dd HH:mm:ss.ffff";
   private const TIMESTAMP_DISPLAY2 string = "MM/dd/yyyy HH:mm:ss";

   function demo()
      itsAboutTime();
   end

   private function itsAboutTime()
      BatchLib.startFunction("It's About Time");
      now timestamp = getTimestamp();
      displayTimestamp(now);
      BatchLib.endFunction();
   end

   private function displayTimestamp(ts timestamp in)
      s string = StringLib.format(ts, TIMESTAMP_DISPLAY1);
      SysLib.writeStdout(s);

      t string = StringLib.format(ts, TIMESTAMP_DISPLAY2);
      SysLib.writeStdout(t);
   end

   function getTimestamp() returns(timestamp)
      result timestamp("yyyyMMddHHmmssfffff");
      return(result);
   end

   function formatTimestamp(t timestamp in) returns(string)
      result string = StringLib.format(t, TIMESTAMP_DISPLAY1);
      return(result);
   end
end
```

We have a function called “`demo`”, just like in the “`DateLib`” library. And then, we have it calling “`itsAboutTime`” because this lesson is all about time.

Let’s look at the “`itsAboutTime`” function. It uses the “`startFunction`” and “`endFunction`” to mark the beginning and end. It uses the timestamp type.

In this case, we’re going to use a function, a convenient function, called “`getTimestamp`” to get the timestamp. We have a construct here that returns a timestamp. By initializing it this way, we get the current time. We have "yyyy" for the four-digit year, "MMddHHmmss" and five little f's for the month, day, hour, minutes, seconds and milliseconds. We're going to get the "`result`" variable filled with today's timestamp and then return the result. That's what the "`getTimestamp`" function does.

We go back to where it is called in the "`itsAboutTime`" function. We are creating a variable called "`now`". And then, we are going to display it with "`displayTimestamp`".

The "`displayTimestamp`" function is going to format the timestamp in two different ways. It is using the original "`StrLib`" library from RBD. There's an RBD-compatible library that we're using. We're going to fix that up in this lesson.

"`TIMESTAMP_DISPLAY_ONE`"—let's take a look at that. It's in this format, a very long format, including the milliseconds. So, that will be displayed first.

And then, "`TIMESTAMP_DISPLAY_TWO`" will format it again. The same timestamp will be formatted again, but using a different timestamp format string. We'll open that on selection. This is a little bit shorter because it doesn't include milliseconds. It does month, day, year, hour, minutes and seconds.

Those two things should not be confused. The format timestamp and the constructor timestamp are very different. The constructor timestamp determines the precision of the timestamp variable. The display timestamp determines how it's going to be turned into a string, how it's going to be converted to a string.

That's just about it. We're using the same "`BatchLib`" library from lesson 3.

Let's take this for a test drive and see what we have. Right-click in the editor and select the Run As | EGL Java Main Application option.

Listing: Lesson 5 Console

```
Lesson 05
START It's About Time
2013-01-25 17:21:51.8990
01/25/2013 17:21:51
END It's About Time
```

We'll open up the console. This is lesson 5; that's to be expected. The "`itsAboutTime`", that's where this function starts. And then, this is the formatted timestamp for right now. And then, formatted again, the same timestamp is a little bit shorter. The format string, you can put the different components in any order, use different delimiters and format it however you like when you convert to a string.

You'll notice that we have the RBD-compatible version of this, where the name of the function contains the type that is being formatted. But, with EGL Development Tools, the "`format`" function is *overloaded*. In other words, it takes different types. It's the same name over and over again, even though it formats different types. So, we just converted the RBD-compatible "`StrLib`" to "`StringLib`".

There's one other thing here. Organize Imports means that the EGL tool will determine what imports are needed. And since we've eliminated the dependency on "`StrLib`", Organize Imports gets rid of the import of "`StrLib`".

These two functions that we're using to demonstrate the timestamp feature are private. But, these two functions down at the bottom, "`getTimestamp`" and "`formatTimestamp`", are going to be used in future lessons so we need to make them not private; we leave out the private keyword. The "`private`" keyword means that it is only going to be used within this library. Without the private keyword, it can be used by other libraries and programs.

Now that we've converted it to the EDT way of doing things, let's take it for another test drive. We get the same results.

This has been another episode of *Essential EGL+Batch*. Thanks for watching!

Lesson 06 BatchLib

Update the “`BatchLib`” library to print start and end times for a program. Use the “`TimestampLib`" library from Lesson 5.

Listings

The following listings for “`Lesson06.egl`” and “`BatchLib.egl`” show the EGL source code as it appears at the end of lesson 6.

Listing: `lab/lesson06/Lesson06.egl`

```
package lab.lesson06;

program Lesson06

   function main()
      BatchLib.startProgram("Lesson 06");
      demo();
      BatchLib.endProgram();
   end

   private function demo()
      BatchLib.startFunction("demo");
      BatchLib.endFunction();
   end
end
```

Listing: `lab/lesson06/BatchLib.egl`

```
package lab.lesson06;

import lab.lesson05.TimestampLib;

library BatchLib
   private const PREFIX_START string = "START ";
   private const PREFIX_END string = "END ";

   private functionName string?;
   private programName string?;

   function startFunction(name string in)
      functionName = name;
      message string = PREFIX_START :: functionName;
      SysLib.writeStdout(message);
   end
```

```
    function endFunction()
        if(functionName == null)
            return;
        else
            message string = PREFIX_END :: functionName;
            SysLib.writeStdout(message);
            functionName = null;
        end
    end

    function startProgram(name string in)
        programName = name;
        ts timestamp = TimestampLib.getTimestamp();
        s string = TimestampLib.formatTimestamp(ts);
        message string = PREFIX_START :: programName :: " - " :: s;
        SysLib.writeStdout(message);
    end

    function endProgram()
        if(programName == null)
            return;
        end

        ts timestamp = TimestampLib.getTimestamp();
        s string = TimestampLib.formatTimestamp(ts);
        message string = PREFIX_END :: programName :: " - " :: s;
        SysLib.writeStdout(message);
    end
end
```

Transcript: Lesson 06 BatchLib

Welcome to *Essential EGL+Batch*. This series is based upon EGL Development Tools version 0.8.

In this lesson, which is lesson 6, we're going to use some of the "`TimestampLib`" library that we developed in lesson 5. We are going to go back to the "`BatchLib`" library from lesson 3 and fix it up a bit so that we'll find also, not just the start and end of a function, but the start and end of a program.

So, let's take a look at the starting code for lesson 6.

Listing: Starting code for "`Lesson06`" program

```
package lab.lesson06;

program Lesson06
```

```
   function main()
      SysLib.writeStdout("Lesson 06");
      demo();
   end

   private function demo()
      functionName string = "demo";
      BatchLib.startFunction(functionName);
      BatchLib.endFunction();
   end
end
```

The starting code is going to be exactly what we've been working with as far as the program; but, in the "`BatchLib`" library, we're going to add a program name. We're going to add two functions, which are "`startProgram`" and "`endProgram`".

Listing: Starting code for "`BatchLib`" library

```
package lab.lesson06;

library BatchLib
   private const PREFIX_START string = "START ";
   private const PREFIX_END string = "END ";

   private functionName string?;
   private programName string?;

   function startFunction(name string in)
      functionName = name;
      message string = PREFIX_START :: functionName;
      SysLib.writeStdout(message);
   end

   function endFunction()
      if(functionName == null)
         return;
      else
         message string = PREFIX_END :: functionName;
         SysLib.writeStdout(message);
         functionName = null;
      end
   end

   function startProgram(name string in)
      programName = name;
      message string = PREFIX_START :: programName;
      SysLib.writeStdout(message);
   end
```

```
    function endProgram()
        if(programName == null)
            return;
        end

        message string = PREFIX_END :: programName;
        SysLib.writeStdout(message);
    end
end
```

I need to copy these source code files. And then, I'm going to create a new package for lesson 6 in the "`eglprogram`" project. And, you've seen how to do this before. The name of the package is "`lab.lesson06`". I'll right-click and select the Paste option to bring in that source code.

What do we have? We have a program called "`Lesson06`" that prints out "Lesson 06". This is pretty much what we've been doing since the beginning of this series.

Let's do some fixing up. For example, in the "`demo`" function, we really don't need to declare a variable called "`functionName`". Instead, we can take the literal string and pass it to the "`startFunction`" function directly. So, what we're going to do is we're going to eliminate that extra variable. What that elimination does is it's not going to be able to show up in our debugger because it's a literal. But, that's simpler code.

Listing: Final "demo" function

```
    private function demo()
        BatchLib.startFunction("demo");
        BatchLib.endFunction();
    end
```

This is just a demo function. It doesn't do anything, except demonstrate the technique.

Typically, what we've been doing in the "`main`" function is writing directly to the console the name of our lesson. What we really need to do is incorporate that into "`BatchLib`" as a "`startProgram`" function. The parameter will be the name of the lesson. After we've done whatever we want to do, we want to call the "`endProgram`" function.

Listing: Final "`main`" function

```
  function main()
     BatchLib.startProgram("Lesson 06");
     demo();
     BatchLib.endProgram();
  end
```

It will start out with the name of the lesson, and then, it will end with the name of the lesson. Meanwhile, it will start with each of the functions and end with each of the functions, in kind of a nested way.

How do we do this? This is the "`BatchLib`" library that is coming from lesson 3. We've seen how to do this before when we added the function name. We've added the program name. We stole from "`startFunction`"; we created another function called "`startProgram`". We took "`endProgram`" pretty much from "`endFunction`". We just changed the names of the variable that it's using. We're using "`programName`" in these functions instead of "`functionName`"; but, it's basically the same code.

What does this look like when it runs? Let's take it for a test drive. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option. We'll take a look at the console.

Listing: Lesson 6 Console (1)

```
START Lesson 06
START demo
END demo
END Lesson 06
```

As you can see, we see the start of the lesson, and then we see the start of the function that we're demonstrating, and then the end of the function, and then, the end of the lesson. So, that's the end of the program.

There's a couple of other things we'd really like to do with this. So, let's get back into it.

Really, what we would like to have is, at the beginning of the program, we'd like to see the time that it started. In lesson 5, we created a "TimestampLib" library that can do this for us.

First, we're going to declare our timestamp. That's the timestamp type and called "ts". We're going to use the "TimestampLib" library.

Well, when we do the dot, it doesn't do anything for us. So, let's put in a fictional function and do Organize Imports.

Now, it recognizes "TimestampLib". So, we can go back and do Control +Space or a dot, and it will bring up the functions that are in "TimestampLib" from lesson 5.

We want to get a timestamp; we're going to capture that timestamp. And then, we're going to format that timestamp. We're going to use the "formatTimestamp" function to format a timestamp.

Let's see. It's going to be "ts" as the timestamp that we're passing to it.

Does it take a parameter? Do we have to pass a timestamp format parameter like we do with "StringLib"? Actually, the editor is complaining about the extra parameter.

Let's see. What is it saying? We mouse over the red X. It cannot be resolved, that's right, because "TIMESTAMP_DISPLAY_ONE" is private. We can check the syntax for the "formatTimestamp" function from lesson 5. We'll see that what we want to do is make the timestamp display part of the library instead of part of each of the individual programs. So, we don't want a parameter that allows us to change the timestamp format.

Listing: Final "`startProgram`" function

```
function startProgram(name string in)
    programName = name;
    ts timestamp = TimestampLib.getTimestamp();
    s string = TimestampLib.formatTimestamp(ts);
    message string = PREFIX_START :: programName :: " - " :: s;
    SysLib.writeStdout(message);
end
```

I'll take the same code to capture the timestamp at the end of the program so that we can print that. Copy and Paste. Let's see. It will say something like "END", the name of the lesson, and then it'll give us the timestamp.

Listing: Final "`endProgram`" function

```
function endProgram()
   if(programName == null)
      return;
   end

   ts timestamp = TimestampLib.getTimestamp();
   s string = TimestampLib.formatTimestamp(ts);
   message string = PREFIX_END :: programName :: " - " :: s;
   SysLib.writeStdout(message);
end
```

So, it looks like everything works out. That's the lesson 5 "`TimestampLib`" library.

Let's take this for a test drive and see what happens. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option. Open up the console.

Listing: Lesson 6 Console (2)

```
START Lesson 06 - 2013-01-25 17:39:36.5370
START demo
END demo
END Lesson 06 - 2013-01-25 17:39:36.0720
```

Look! The lesson started at exactly that moment and the lesson ended at exactly that moment. So, it took what? Do the math. It took almost a second to run.

There is one little, small typo. It needs an extra hyphen. I found that. I didn't have to go into the debugger to find that because I just wrote that piece of code. Now, this is a little more symmetrical. Well, that's it for lesson 6.

This has been part of the *Essential EGL+Batch* series. Thanks for watching!

Lesson 07 Character

Get a characters from a string and limit the number of characters in a string.

Transcript: Lesson 7 Character

Welcome to *Essential EGL+Batch*. This series is based upon EDT, EGL Development Tools version 0.8.

In this lesson, which is lesson 7, we're going to be talking about characters. Characters are part of a string, or substrings.

Let's take a look at the finished code for lesson 7. We have a program, as usual, called "`Lesson07`". It uses the new "`BatchLib`" library from lesson 6.

Listing: `lab/lesson07/Lesson07.egl`

```
package lab.lesson07;

import lab.lesson06.BatchLib;

program Lesson07

  function main()
    BatchLib.startProgram("Lesson 07");
    CharLib.demo();
    BatchLib.endProgram();
  end
end
```

And, we have a library called "`CharLib`". It has a function called "`yourSuchACharacter`" that deals with trying to get substrings. And also, another function called "`knowYourLimitations`", which is dealing with a string type that has a fixed length.

Listing: `lab/lesson07/CharLib.egl`

```
package lab.lesson07;

import lab.lesson06.BatchLib;

library CharLib
```

```
    function demo()
        youreSuchACharacter();
        knowYourLimitations();
    end

    private function youreSuchACharacter()
        BatchLib.startFunction("You're Such a Character");
        text string = "123456789a123456789b";
        SysLib.writeStdout("text=" :: text);

        // get first character
        s1 string = text[1 : 1];
        SysLib.writeStdout("s1=" :: s1);

        // get first eight characters
        s8 string = text[1 : 8];
        SysLib.writeStdout("s8=" :: s8);

        // get first twenty characters
        s20 string = text[1 : 20];
        SysLib.writeStdout("s20=" :: s20);

        // get last four characters
        s4 string = text[text.length() - 3 : text.length()];
        SysLib.writeStdout("s4=" :: s4);

        BatchLib.endFunction();
    end

    private function knowYourLimitations()
        BatchLib.startFunction("Know Your Limitations");
        text string = "this is a string.";
        SysLib.writeStdout("text=" :: text);

        // get first character
        t1 string(1) = text;
        SysLib.writeStdout("t1=" :: t1);

        // get first eight characters
        t8 string(8) = text;
        SysLib.writeStdout("t8=" :: t8);

        // get first twenty characters
        t20 string(20) = text;
        SysLib.writeStdout("t20=" :: t20);

        BatchLib.endFunction();
    end
end
```

We're going to copy the source code to our "`eglprogram`" project, create a package for lesson 7 called "`lab.lesson07`", and then, Paste in the source code.

Let's take a closer look at the program. You'll notice that it doesn't use the "`StdLib`" library to write a message to the console. Rather, it depends upon the "`BatchLib`" library. The only thing that this does is it has a "`main`" function which calls the "`demo`" function. The "`demo`" function is in a library called "`CharLib`".

The "`demo`" function calls the "`yourSuchACharacter`" function. What does this function do? As usual, it calls "`startFunction`" with the name of the function. It declares a string called "`text`" with exactly twenty characters in it. It writes out that string to the console so that we can see the original string.

This is the EGL way of getting the first character of a string. It uses the square brackets, which treats the "`text`" variable something like an array of characters. It uses a colon (:). "1:1" gives you the first character. "1:8" will get you the first eight characters. It's from a starting position to an ending position. Characters "1:20" gives you twenty characters. Again, that's starting position, ending position; the "20" is not a length; but, it's actually a position. It writes out that.

To get the last four characters, we have to do a little bit of math. It's the length of the string; we'll do "`text.length()`". If we want four characters, it's actually minus 3 because we're dealing with position. The last position in the string is "`text.length()`". So, that will get four characters, the last four characters of the string. That's pretty much it for the "`yourSuchACharacter`" function.

That's one way of doing it; but, there's another way, which is limiting the size of the string based upon the string type. It's in the "`knowYourLimitations`" function. Limiting the number of characters in a string based upon type is what this function demonstrates.

This is very similar in declaring a variable called "`text`".

Notice this. We've put the size of "`t1`" in parentheses. We want string open parenthesis one. The most number of characters that will every be in "`t1`" is one, a single character. So, if we try to assign "`text`" to "`t1`", it will only copy the first character.

The same thing with "`t8`". Only eight characters can be stuffed into "`t8`".

With "`t20`", up to twenty characters can be stuffed into "`t20`". So, although "`text`" is not quite twenty characters, we'll see what happens. That's pretty much it for the demonstration of character.

We can format that code and save it. Right-click and select the EGL Source | Format option. Press the **Save All** button.That's all of the source code for the "`CharLib`" library.

Let's get back to the main program. It looks pretty good. Let's take it for a test drive. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option. Switch to the console and open it.

Listing: Lesson 7 Console

```
START Lesson 07 - 2013-01-25 18:45:42.3770
START You're Such a Character
text=123456789a123456789b
s1=1
s8=12345678
s20=123456789a123456789b
s4=789b
END You're Such a Character
START Know Your Limitations
text=this is a string.
t1=t
t8=this is
t20=this is a string.
END Know Your Limitations
END Lesson 07 - 2013-01-25 18:45:42.9290
```

From lesson 6, we get exactly when the program started. We have the beginning of the function, "`yourSuchACharacter`". This is a twenty character string. See "`s1`" is equal to "1"; "`s8`" is the first eight characters; "`s20`" is twenty characters; and "`s4`" is exactly four characters. That's the end of that function.

Now, doing it a different way, using a different technique, with the limited strings, the number of characters is defined as part of the type. You'll notice that when you assign spaces, "`t8`" includes the space. The space is printed to the console.

But, when you assign a string that's smaller than the size of the other string —"`text`", right there, "this is a string" period, is less than twenty characters—it doesn't fill out the rest with spaces.

This has been *Essential EGL+Batch*. Thanks for watching!

Lesson 08 Integer

Count up and down with an integer. Create and modify an array of integers.

Transcript: Lesson 8 Integer

Welcome to *Essential EGL+Batch*. This is based upon EGL Development Tools version 0.8.

The first thing we want to do today is we want to create a new package. This is lesson 8; so, we're going to create a new package called "`lab.lesson08`". You've seen this before.

From the package, right-click on the package, we're going to create a new library. The library is going to be called "`IntLib`".

Listing: `lab/lesson08/IntLib.egl`

```
package lab.lesson08;

import lab.lesson06.BatchLib;

library IntLib
   private const LOOP_MAX int = 5;

   function demo()
      countUp();
      countDown();
      arrayOfIntegers();
   end

   private function countUp()
      BatchLib.startFunction("counting up");
      iMax int = LOOP_MAX;
      for(i int from 1 to iMax by 1)
         SysLib.writeStdout(i);
      end
      BatchLib.endFunction();
   end

   private function countDown()
      BatchLib.startFunction("counting down");
      iMax int = LOOP_MAX;
      for(i int from iMax to 1 decrement by 1)
         SysLib.writeStdout(i);
      end
      BatchLib.endFunction();
```

```
    end

    private function arrayOfIntegers()
        BatchLib.startFunction("array of integers");
        list int[] =[5, 10, 11, 20, 25, 30];

        showIntegerArray(list, 0);

        position int = 3;
        SysLib.writeStdout("= setting value at position " :: position);
        list[position] = 15;
        showIntegerArray(list, position);

        const value int = 35;
        SysLib.writeStdout("+ adding a value of " :: value);
        list.appendElement(value);
        showIntegerArray(list, list.getSize());

        const pos int = 2;
        SysLib.writeStdout("- removing a value at position " :: pos);
        list.removeElement(pos);
        showIntegerArray(list, pos);

        BatchLib.endFunction();
    end

    private function showIntegerArray(list int[] in, pos int in)
        message string? = null;
        iMax int = list.getSize();
        for(i int from 1 to iMax by 1)
            s string = list[i];
            if(i == pos)
                s += "*";
            end
            if(message == null)
                message = s;
            else
                message += ", " :: s;
            end
        end
        SysLib.writeStdout(message);
    end
end
```

The first thing I want to do is I want to add a private constant integer called "`LOOP_MAX`". This will make it very easy for me to set number of times that we go through a loop.

We're going to have a function called "`demo`", which is not private so it can be called outside the library.

I'm going to create a private function called "`countUp`". This function is going to count from one upward.

I'll create a variable called "`iMax`", which is an integer, and will set that equal to the "`LOOP_MAX`" constant.

This is the syntax for constructing a loop. That's "`for(i int from 1 to iMax by 1)`".

And then, inside the loop, I'm just going to write out the value of of "`i`". So, that will count up.

Oh, and we have to put in calls to "`startFunction`" and "`endFunction`" so we know where we are in the code.

I put in both statements and then I'm going to right-click and go down to EGL Source | Organize Imports. There are three libraries called "`BatchLib`". I'm going to take the one from lesson 6.

We'll create another function, a private function. This time, it will be called "`countDown`".

Let me demonstrate that, if you continue typing after you do a dot and pause, you can type the first character in a name and then the list will shorten to match those that start with just that character.

Let me do that again. After I type the letter "e", you see that we have just "`endFunction`" and "`endProgram`" in the list.

To our loop! Again, we're going to do "`iMax`" equals "`LOOP_MAX`". It's an integer.

And then, this is the syntax for counting down. It's "`for(i int from iMax to 1 decrement by 1)`". So, there is a distinction made between counting up and counting down in EGL. We'll print "`i`" on the console.

Next, we want to add a function to display an array of integers. Let's work with an array of integers now. We use an integer as an index for an array. So, an integer has no decimal part so it's a whole number.

This time, I'll quickly type a dot and the first few characters of the function name. When I hit Control+Space—let me do that again. When I hit Control +Space, it will fill in the rest of the name of the function for me.

Here's how we develop an array: "list" is the name of the variable; "int", and then, open square bracket ([)and close square bracket (]) is the type. It's an array of int.

We can fill in values for the array using the square brackets and a comma-delimited list. This is going to be an array. Arrays are always variable in size. This is a convenient way of initializing.

I want to build a function to use repeatedly here, which enables us to display the array. It's going to take two parameters: the array of values, which is the array of integers, and the second parameter will be a position that we want to highlight or emphasize.

Again, in the "showIntegerArray" function, "list" is the name of the variable; "int[]" is the type; and it's an in parameter, which means it's not updatable within this function. So, we can't make assignments to the "list" variable.

As far as the content of the "showIntegerArray" function, instead of typing it in one line at a time, I think I'll just Paste it and then review it.

What we have is a variable called "message". It's a string with a question mark ("string?"), which we've seen before, which means it's a nullable string. And, although I don't have to, I'm going to explicitly set it to null. It's easier to read. It's easier for the next programmer to understand.

Now, "list.getSize()" is how we find the size of an array. And then, we do the count up part, to count through the array of things.

"list[i]" is what enables us to get a value. We'll assign it to "s", which is a string. On the right side of the assignment, it's an integer; on the left side, it's a string. So, there's an implied conversion. Explicitly, that would be the "as" operator, where I can say "as string". It will convert from one type to another with the keyword "as". I'll put it back to just implied.

If the current position is equal to the position that is given, I'll add an asterisk.

If message is null, which means that it's the first time through the loop, (or I could have written "i==1"), I'll set the message. Otherwise, I'm going to use the plus-equals (+=) operator to append additional text to the message. So, that should give me a nice evaluation of the array, a nice content of the array.

Note: I should have used the colon-colon-equals (::=) operator for string concatenation instead of the plus-equals (+=), which is for math.

There are a couple of things we want to do with arrays to show how to work it. Let me first set the value of one of the elements in the array. Again, "`position`" is an integer; but, the concatenation operator, which is colon-colon, implies an "`as string`". You can write that explicitly or it's implied that it's adding an integer to a string.

To set a particular position, we put "`list`", open square bracket ([), and then the index, and close square bracket (]) on the left hand side of the assignment statement. So, we can assign position three with the value of 15. It was 11; we're going to change it to 15.

I'll use the "`showIntegerArray`" function. I hit the Control+Space to finish typing out the name of the function. It's pretty smart.

Then, I'm going to work with adding a value to an array. The value that I want to add is 35. The way we add a value to an array is with the "`appendElement`" function. There are quite a number of things that you can do with an array, one of them being "`appendElement`". When we have an integer array, we can add an integer. That adds an integer to the array. It's adding, not as in mathematically adding; but, it's adding as in putting an additional element in there so the array looks something like that, with 35 at the end. So, arrays are always dynamic when they are declared like this, using this syntax.

In the call to "`showIntegerArray`", I'll use "`list.getSize()`" to emphasize the last value in the array because the array size has changed.

The next thing I want to do is remove a value from the list. It is removed based upon its position. We're going to remove from position 2. So, we'll drop the 10 out of the list.

To remove an element, use "`list.removeElement()`". It takes a parameter of the position; so, it's removing based upon position.

I'll save the file.

We're going to add a new program. It's going to be called "Lesson08". Basically, the "main" function is going to call "IntLib.demo()".

Listing: lab/lesson08/Lesson08.egl

```
package lab.lesson08;

import lab.lesson06.BatchLib;

program Lesson08

  function main()
    BatchLib.startProgram("Lesson 08");
    IntLib.demo();
    BatchLib.endProgram();
  end
end
```

I wanted to show that the import is at the top of each source code file. So, the program is not going to inherit the imports from the library. Let me do this again. It's right-click, EGL Source and Organize Imports. Since there is more than one thing called "BatchLib", we have to choose one. If there were one thing called "BatchLib", then it's filled in for us.

Another way to do that is use the fully qualified name of the "BatchLib" library. So, it's "lab.lesson06.BatchLib". It's another name for the same thing. The import statement is a convenience so that you can more easily maintain the code when you have multiple occurrences of "BatchLib". It's consolidated at the top of the file.

I forgot a line of code in the "showIntegerArray" function so I'll type that in now. Okay. That looks a little bit better.

We'll run it again. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option. Switch to the Console view.

Listing: Lesson 8 Console

```
START Lesson 08 - 2013-01-29 17:26:21.5330
START counting up
```

```
1
2
3
4
5
END counting up
START counting down
5
4
3
2
1
END counting down
START array of integers
5, 10, 11, 20, 25, 30
= setting value at position 3
5, 10, 15*, 20, 25, 30
+ adding a value of 35
5, 10, 15, 20, 25, 30, 35*
- removing a value at position 2
5, 15*, 20, 25, 30, 35
END array of integers
END Lesson 08 - 2013-01-29 17:26:22.0880
```

Here is our original array of integers. It has an 11 in position 3. We set the value at position 3 to 15. There it is! Adding a value of 35 adds a value at the end. Removing a value at position 2, the 10 is gone!

This has been another lesson in *Essential EGL+Batch*. Thanks for watching!

Lesson 09 Float

Calculate and format floating point number. Use "`MathLib`", an EGL standard library.

Transcript: Lesson 9 Float

Welcome to *Essential EGL+Batch*. In this lesson, we're going to learn about floats. This series is based upon EGL Development Tools version 0.8.

To get started, we need to create a new package called "`lab.lesson09`". A floating point type is what we're going to deal with today. Last time, we dealt with integers, which is a whole number; but, a floating point has a number and some things after the decimal place, some additional digits. Floating point means that it has an exponent so it can represent numbers that are very, very big (or very, very small).

We'll create a library called "`FloatLib`".

Listing: `lab/lesson09/FloatLib.egl`

```
package lab.lesson09;

import lab.lesson06.BatchLib;

library FloatLib

  function demo()
    doMath();
    formatFloat();
  end

  private function doMath()
    BatchLib.startFunction("do math");

    result1 float =(4.8 + 3.2) * 2.5;
    SysLib.writeStdout("1. ( 4.8 + 3.2 ) * 2.5 = " :: result1);

    const value2 float = -2.1;
    result2 float = MathLib.abs(value2);
    SysLib.writeStdout("2. abs( -2.1 ) = " :: result2);

    const value3 float = 10.0;
    const value4 float = 12.0;
    result3 float = MathLib.max(value3, value4);
```

```
        SysLib.writeStdout("3. max( 10.0, 12.0 ) = " :: result3);

        result4 float = MathLib.min(value3, value4);
        SysLib.writeStdout("4. min( 10.0, 12.0 ) = " :: result4);

        const value5 float = 23.45;
        result5 float = MathLib.round(value5, -2);
        SysLib.writeStdout("5. round( 23.45, -2 ) = " :: result5);

        BatchLib.endFunction();
    end

    private function formatFloat()
        BatchLib.startFunction("format float");

        const value float = 123.456789;

        const pattern1 string = "####&";
        SysLib.writeStdout("1. " :: StringLib.format(value, pattern1));

        const pattern2 string = "####&.&&";
        SysLib.writeStdout("2. " :: StringLib.format(value, pattern2));

        const pattern3 string = "####&.&&&&&&";
        SysLib.writeStdout("3. " :: StringLib.format(value, pattern3));

        const pattern4 string = "####&.&&&&&&&&&";
        SysLib.writeStdout("4. " :: StringLib.format(value, pattern4));

        const pattern5 string = "&&&&&";
        SysLib.writeStdout("5. " :: StringLib.format(value, pattern5));

        const pattern6 string = "&&&&&.&&";
        SysLib.writeStdout("6. " :: StringLib.format(value, pattern6));

        const pattern7 string = "&&&&&.&&&&&&&&&";
        SysLib.writeStdout("7. " :: StringLib.format(value, pattern7));

        const pattern8 string = "****&&&&";
        SysLib.writeStdout("8. " :: StringLib.format(value, pattern8));

        const pattern9 string = "###,###&.&& USD";
        SysLib.writeStdout("9. " :: StringLib.format(value, pattern9));

        BatchLib.endFunction();
    end
end
```

We'll start a function called "`demo`".

I'll do the usual of creating another function, which is private, and also calling the "startFunction" and "endFunction" functions from the "BatchLib" library from lesson 6.

I want to demonstrate here the right-click and EGL Source | Organize Imports, what happens when there's more than one thing called "BatchLib"—just as a reminder; we did this before. There it is again.

Now, let's get started on floats. We want to create an expression of 4.8 plus 3.2 times 2.5. Notice, when I type a dot inside quotation marks, although I'm editing a string, Content Assist wants to pop up.

Let me do that again. When it pops up unexpectedly, just hit the Escape (Esc) key to make it go away.

Let's also do an absolute value. We'll use a value of -2.1. It's absolute value should be (positive) 2.1. There's a "MathLib" library that's built into EGL. Type a dot and wait; Content Assist pops up. We'll select the absolute value ("abs") function.

(For the "result1" variable, I have to go back and change the int type to a float type because we're dealing with floating point numbers in this lesson.)

I'm going to have two values that are float. I'm going to use the "MathLib" library again to get the maximum of two values. Type a dot and wait; Content Assist pops up. I'll type an "m" to get to the "max" function. So, we'll find out the max of "value3" and "value4". I'll put that on the console.

And then, I'll take the "MathLib" library again, look for the "min" function, which takes the minimum of two values, and print that out to the console.

Let's also look at the "round" function. I'll put in a value of 23.45 and see the result when it's rounded to the nearest two decimal places. So, we'll use the "round" function. Two decimal places is a negative 2.

That looks pretty good. I think we got that right.

I also want to format a floating point number. I'll create another function and show you the different format patterns. In this case, I'm going to paste in the text of the "formatFloat" function.

We're going to use a constant value of 123.456789. Take a look at these patterns. The pound sign or hash mark (#) is a space-filled value. The ampersand (&) is a zero-filled value. The decimal point in a pattern aligns with a decimal point in a floating point number. With so many decimal places, you can have additional decimal places that are zero-filled. Or, the other way, you can have zero-filled preceding zeros. If you use literals, like an asterisk (*), in a format string, it will print out the literal. And, if you use a literal between two hash marks, like this (“###,##&.&& USD”), if there are no digits to print, it won't print the literal. At the end, I can use a literal space and a literal “USD” for U.S. Dollars, or other symbols.

To go with this floating point library demonstration, let's create a program. Right-click on the “lab.lesson09” package and select the New | Program option. It will be called the usual “Lesson09” and the usual call to the “startProgram” and “endProgram” functions.

Listing: `lab/lesson09/Lesson09.egl`

```
package lab.lesson09;

import lab.lesson06.BatchLib;

program Lesson09

   function main()
      BatchLib.startProgram("Lesson 09");
      FloatLib.demo();
      BatchLib.endProgram();
   end
end
```

Let's take it for a test drive. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option.

Let's take a look at the console and see what we have.

Listing: Lesson 9 Console

```
START Lesson 09 - 2013-01-29 17:58:03.6430
START do math
1. ( 4.8 + 3.2 ) * 2.5 = 20.0
2. abs( -2.1 ) = 2.1
3. max( 10.0, 12.0 ) = 12.0
4. min( 10.0, 12.0 ) = 10.0
```

```
5. round( 23.45, -2 ) = 23.45
END do math
START format float
1.   123
2.   123.45
3.   123.456789
4.   123.456789000
5. 00123
6. 00123.45
7. 00123.456789000
8. ****0123
9.     123.45 USD
END format float
END Lesson 09 - 2013-01-29 17:58:04.1970
```

So, for lesson 9, it does the math.

(4.8 + 3.2) * 2.5 = 20.0

Absolute value is done, the max, the min and the round. With rounding, negative 2, we're talking about rounding to the second decimal place. Let's put in some additional numbers in the third decimal place, such as a six, as in 23.4567, and run this again.

Isn't that amazing? It generates, compiles and displays the output in the blink of an eye.

The "`round`" function has rounded it up to 23.46.

Let's try another number. Let's try 23.4549 and see what happens. Run it again.

This has been another lesson in Essential EGL+Batch. Thanks for watching!

Up next

Lesson 10 Bonus Lesson is next. A *bonus lesson* is a quick and simple lesson that uses source code from previous lessons.

Lesson 10 Delegate

Use functions from lessons 4 thru 9.

Transcript: Lesson 10 Delegate

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

This is a bonus video. A bonus video means that we're going to do something short and simply and we're going to have a payoff from some of the previous lessons.

We're going to create a new EGL package called "`lab.lesson10`". We're going to create a new program called "`Lesson10`".

Listing: `lab/lesson10/Lesson10.egl`

```
package lab.lesson10;

import lab.lesson04.DateLib;
import lab.lesson05.TimestampLib;
import lab.lesson06.BatchLib;
import lab.lesson07.CharLib;
import lab.lesson08.IntLib;
import lab.lesson09.FloatLib;

program Lesson10

   function main()
      BatchLib.startProgram("Lesson 10");

      const list onDemo[] =[DateLib.demo, TimestampLib.demo, CharLib.demo,
                  IntLib.demo, FloatLib.demo];
      iMax int = list.getSize();
      for(i int from 1 to iMax by 1)
         item onDemo = list[i];
         item();
      end

      BatchLib.endProgram();
   end
end

delegate onDemo();
```

In lesson 10, we're going to learn about a new type. Like I said, we're going to have a payoff from things we've already done. So, we'll do the usual things, like use the "`BatchLib`" library from lesson 6.

I'm going to do the Organize Imports, which is right-click and select the EGL Source | Organize Imports option. I'll pick the "BatchLib" library from lesson 6. I want to demonstrate a technique to speed things up, which is to select text and then hit Control+Space. So, I select "`x()`" and press Control+Space. Since text was selected, it will be replaced when I select something from Content Assist. It makes programmers more productive.

Let's go to the new type. The new type is *delegate*. What is a delegate? A delegate is a reference to a function. Once it is a reference to a function, it can be called like a function.

The name of the type is "`onDemo`". The number of parameters is zero (0) because there is nothing between the open and close parentheses. The return type is nothing because it has no returns phrase.

Let's create a constant list of "`onDemo`" delegates. So, it's a variable called "`list`", the type is "`onDemo`", and the square brackets makes it an array. We'll type in the names of some functions we've already built in previous lessons.

The first thing I want to do is the EGL Source and Organize Imports. It will bring in the "`demo`" functions from all those different libraries.

And then, we'll do EGL Source and Format so that it shows really nicely in our editor.

What we can see is that we have the "`DateLib`" library from lesson 4, the "`TimestampLib`" library from lesson 5, the "`BatchLib`" library from lesson 6, the "`CharLib`" library from lesson 7, the "`IntLib`" library from lesson 8, and the "`FloatLib`" library from lesson 9.

What we need to do is count up. We've seen this before in lesson 8. We're going to count through the items in the list. We'll create a variable called "`item`", which is also the delegate type "`onDemo`". It's going to take the value of each of the functions in turn. So, it will match each of these functions. We can call the item. As it goes through this list, it's going to count through the list. It's going to call the "`demo`" function from each of the libraries. It's pretty amazing in a little bit of code.

Let's take this for a test drive. We can run all of our lesson programs with right-click and select the Run As | EGL Java Main Application option. Let's take a look at the console.

Listing: Lesson 10 Console

```
START Lesson 10 - 2013-01-29 18:11:51.8780
START Go On Date
2013-01-29
01/29/2013
2011-11-11
11/11/2011
2012-12-12
12/12/2012
END Go On Date
START It's About Time
2013-01-29 18:11:52.4180
01/29/2013 18:11:52
END It's About Time
START You're Such a Character
text=123456789a123456789b
s1=1
s8=12345678
s20=123456789a123456789b
s4=789b
END You're Such a Character
START Know Your Limitations
text=this is a string.
t1=t
t8=this is
t20=this is a string.
END Know Your Limitations
START counting up
1
2
3
4
5
END counting up
START counting down
5
4
3
2
1
END counting down
START array of integers
5, 10, 11, 20, 25, 30
= setting value at position 3
```

```
5, 10, 15*, 20, 25, 30
+ adding a value of 35
5, 10, 15, 20, 25, 30, 35*
- removing a value at position 2
5, 15*, 20, 25, 30, 35
END array of integers
START do math
1. ( 4.8 + 3.2 ) * 2.5 = 20.0
2. abs( -2.1 ) = 2.1
3. max( 10.0, 12.0 ) = 12.0
4. min( 10.0, 12.0 ) = 10.0
5. round( 23.45, -2 ) = 23.45
END do math
START format float
1.   123
2.   123.45
3.   123.456789
4.   123.456789000
5. 00123
6. 00123.45
7. 00123.456789000
8. ****0123
9.      123.45 USD
END format float
END Lesson 10 - 2013-01-29 18:11:52.4870
```

There you have it.

This has been another lesson of *Essential EGL+Batch*. Thanks for watching!

Lesson 11 Record

Create and use a simple record.

Transcript: Lesson 11 Record

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

This is lesson number 11. In this lesson, we'll work with the EGL record.

So, we'll create a new package, as usual, called "`lab.lesson11`".

We'll do something a little bit differently. Instead of using the New | Library or New | Program options, we're going to use the New | Source File option.

This is an EGL source file. Right-click on the EGL package and select the New | Source File option. The name of the source file will be "`VersionInfo`".

Listing: `lab/lesson11/VersionInfo.egl`

```
package lab.lesson11;

record VersionInfo
   name string?;
   version string?;
end
```

We're going to use a record type. A record is basically a collection of fields, or a collection of variables. Then, you can use a record as a container for all of these different fields.

When we create the library, we'll also use New | Source File. The name of the library is "`VersionLib`".

Listing: `lab/lesson11/VersionLib.egl`

```
package lab.lesson11;

import lab.lesson06.BatchLib;

library VersionLib
   private const PROGRAM_NAME string = "Lesson 11";
   private const PROGRAM_VERSION string = "0.0.0";
```

```
    function demo()
        showVersionInfo();
    end

    private function showVersionInfo()
        BatchLib.startFunction("show version");
        info VersionInfo = getVersionInfo();
        SysLib.writeStdout("[version] " :: info.name :: " - " :: info.version);
        BatchLib.endFunction();
    end

    function getVersionInfo() returns(VersionInfo)
        //      result VersionInfo{};
        //      result.name = PROGRAM_NAME;
        //      result.version = PROGRAM_VERSION;
        result VersionInfo{name = PROGRAM_NAME, version = PROGRAM_VERSION};
        return(result);
    end
end
```

Note: When we use New | Source File, it doesn't put in additional comments; it doesn't give examples of things. You have to know what EGL code to put in. This is really good for a record, creating a record in its own source file.

As usual, we'll create a function called "`demo`" in our library. This function is going to call a function called "`showVersionInfo`".

So, it will be a private function. We'll do the decoration using the "`BatchLib`" library from lesson 6.

Now, we need to create a function called "`getVersionInfo`". The purpose of this function is to populate a record called "`VersionInfo`" and return it. So, we have a "`returns`" clause that tells us the name of the type that it's going to return. As a convention, I'm going to call it "`result`". Notice that when I put in "`result`" and dot, it gives me a list of the fields that are in the record. To access a particular field, to set it or get it, we use the name of the variable, dot and the name of the field in the record.

We go back to "`showVersionInfo`". I'm going to get a copy of the version info in a variable called "`info`". I'm going to write out the information to the console. Right now, the "`getVersionInfo`" function is using a C- or Java-style way of populating a record. We'll get back to that in a moment.

Listing: C- or Java-style "`getVersionInfo`" function

```
function getVersionInfo() returns(VersionInfo)
   result VersionInfo{};
   name = PROGRAM_NAME;
   version = PROGRAM_VERSION;
   return(result);
end
```

Okay. That looks good.

I'm going to format it with EGL Source and Format. That's right-click and select the EGL Source | Format option.

Now, we're going to create a new program. I click on the icon for the Project Explorer view. Right click on the "`eglprogram`" project and select New | Source File again. The name of the program is "`Lesson11`". Press the **Finish** button.

Listing: `lab/lesson11/Lesson11.egl`

```
package lab.lesson11;

import lab.lesson06.BatchLib;

program Lesson11

   function main()
      BatchLib.startProgram("Lesson 11");
      VersionLib.demo();
      BatchLib.endProgram();
   end
end
```

This time, I'm going to put in a program. When I put in a program, one of the requirements of a program is you have to have a function called "`main`". So, it's an error, an error message will show up until you have a function called "`main`". Let me do the usual decoration for a program, including the "`BatchLib`" library from lesson 6.

The "`demo`" function is going to call the "`VersionLib.demo()`" function.

Let's run this. Let's take this for a test drive and see what we have. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option.

I click on the Console icon to bring up the Console view.

Listing: Lesson 11 Console

```
START Lesson 11 - 2013-02-01 20:06:32.9660
START show version
[version] Lesson 11 - 0.0.0
END show version
END Lesson 11 - 2013-02-01 20:06:33.5020
```

We have "`Lesson 11`" and "`show version`". Notice that the version info is "`Lesson 11 - 0.0.0`". And that's the end of the program.

We used the C- or Java-style of creating a record. Let's switch this now to do the EGL way of doing things.

I'm going to select three lines of code, and then, use right-click and select the EGL Source | Toggle Comment option to change to a comment for selected text.

I'll use the EGL Source and Format again to format the source code. Right-click somewhere in the editor and select the EGL Source | Format option.

I'm going to do this with the EGL way of doing things. Between the curly braces ({}), we can initialize the fields in a record. It's a comma-delimited list. So, in one line of code—it's actually very efficient in EGL—we populate a record.

I don't like the way this looks so I'm going to select three lines of code and press the Tab key. It will shift or indent the entire selected text to the right. That looks a little bit better, not that it has anything to do with the performance or behavior of the program.

I'll run this again.

This has been another lesson in *Essential EGL+Batch*. Thanks for watching!

Lesson 12 Fixed-Length Record

Convert a simple fixed-length record to a normal EGL record.

Transcript: Lesson 12 Fixed-Length Record

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

Today, we're working with lesson number 12. In this lesson, we're going to work more with records and work with a fixed-length record. So, we need to create the usual package and create a new source file. The package will be called "`lab.lesson12`". The source file will be called "`TimeResponse`".

Listing: `lab/lesson12/TimeResponse.egl`

```
package lab.lesson12;

record TimeResponse
   succeeded boolean?;
   statusCode string?;
   message string?;
   ts timestamp("yyyyMMddHHmssfffff");
end

record TimeResponseFL
   succeeded string(1);
   statusCode string(3);
   message string(20);
   ts string(25);
end
```

Inside the source file, what we're going to do is create a record. Actually, we're going to create two records. The first record will be the EGL record called "`TimeResponse`". The last field in the record is a timestamp.

And then, we're going to create a fixed-length record called "`TimeResponseFL`" and simulate a response from a separate machine, such as a server. In this case, we're going to simulate a message from a server that's in a fixed-length format. So, we're using fixed-length strings. That's it for the record.

Let's create the library. Right-click on the "`eglprogram`" project and select the New | Source File option. The library is called "`TimestampLib`".

Listing: `lab/lesson12/TimestampLib.egl`

```
package lab.lesson12;

import lab.lesson06.BatchLib;

library TimestampLib

   function demo()
      showServerTime();
   end

   private function showServerTime()
      BatchLib.startFunction("show server time");
      if(true)
         response TimeResponseFL = getTimeFromServer();
         SysLib.writeStdout("[response] succeeded=" :: response.succeeded ::
                     " statusCode=" :: response.statusCode ::
                     " message=" :: response.message :: " ts=" ::
                     response.ts);
      end

      response TimeResponse = getServerTime();
      SysLib.writeStdout("[response] succeeded=" :: response.succeeded ::
                  " statusCode=" :: response.statusCode :: " message=" ::
                  response.message :: " ts=" :: response.ts);

      BatchLib.endFunction();
   end

   private function getServerTime() returns(TimeResponse)
      response TimeResponseFL = getTimeFromServer();
      result TimeResponse{};
      //       if(response.succeeded == "T")
      //          result.succeeded = true;
      //       else
      //          result.succeeded = false;
      //       end
      result.succeeded =(response.succeeded == "T");
      result.statusCode = response.statusCode;
      result.message = response.message;
      result.ts = response.ts;
      return(result);
   end

   private function getTimeFromServer() returns(TimeResponseFL)
      result TimeResponseFL{};
      result.succeeded = "T";
      result.statusCode = "000";
      result.message = StringLib.spaces(20);
```

```
        result.ts = "2001-02-03 01:02:03.45678";
        return(result);
    end
end
```

In the library, we're going to do the usual decoration and create a function called "`demo`" and use the "`BatchLib`" library from lesson 6.

The "`demo`" function is going to call a function called "`showServerTime`".

We're going to skip the contents of the "`showServerTime`" function for right now.

We'll create a new function called "`getServerTime`". It's going to return the EGL record, the non-fixed-length record.

And then, we're going to create another function called "`getTimeFromServer`", which returns the fixed-length record. This is the simulated response from a server right here. We'll fill this out so that "`succeeded`" is equal to "`T`" for true; "`statusCode`" is "`000`"; the message is spaces indicating that there is no error message. The timestamp from our simulated server is going to be filled out like this "`2001-02-03 01:02:03.45678`". The "`getTimeFromServer`" returns the result.

The "`getServerTime`" function is going to convert the response from the server to an EGL-type record, which is not fixed-length, so that we can work with it in EGL. First, if "`response.succeeded`" is equal to "`T`", then we're going to set the "`result.succeeded`" equal to "`true`", because in our EGL record the "`succeeded`" field is a boolean (true or false) type. We're just going to copy the "`statusCode`" and "`message`" from "`response`" to "`result`" verbatim with an assignment statement. On the timestamp, "`result.ts`" is a timestamp type, whereas "`response.ts`" is a string. In that process, it will convert from a string to a timestamp. It returns the result.

Extra credit: The "`statusCode`" field in the "`TimeResponse`" record could have been an "`int`" type and conversion to an integer could have been implied. Change the type from "`string`" to "`int`", and see what happens.

Extra credit: The "`message`" field in the "`TimeResponse`" record is a "`string`" and, therefore, is not a fixed-length field. Use the "`clip`" function of a "`string`" to trim spaces. In the assignment statement, change "`response.message`" to "`response.message.clip()`", and see what happens.

In the "`showServerTime`" function, we're going to get the fixed-length record simulating a record coming back from the server. I'll type "`response`" and a dot. Content Assist pops up a list of fields. I'll fill out the rest of this. So, we're going to print out all of the different fields of the record to the console. And then, I'll use EGL Source and Format to make that fit nice on the screen.

I'm going to do the same thing in "`showServerTime`" in a second call. This time, I'm going to get the server time in the EGL normal format. It's going to be not fixed-length anymore. It's going to use EGL types.

An interesting thing is, since I named the record variables the same and they have the same names of fields, I can copy the line to print it out to the console.

We also need a program. So, let me fill out a program now. Right-click on the "`lab.lesson12`" package and select the New | Source File option. The name of the program is "`Lesson12`".

Listing: **`lab/lesson12/Lesson12.egl`**

```
package lab.lesson12;

import lab.lesson06.BatchLib;

program Lesson12

  function main()
    BatchLib.startProgram("Lesson 12");
    TimestampLib.demo();
    BatchLib.endProgram();
  end
end
```

I'll decorate it with the usual stuff. I'll use the "`BatchLib`" library from lesson 6. The "`main`" function is going to call "`TimestampLib.demo()`". And, yes, we've had a library called "`TimestampLib`" before; but this time, it's going to be a lesson 12 library.

Let's take this for a test drive. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option.

I'll click on the Console icon. What do we have?

Listing: Lesson 12 Console

```
START Lesson 12 - 2013-02-01 20:23:24.2440
START show server time
[response] succeeded=T statusCode=000 message=             ts=2001-02-03 01:02:03.45678
[response] succeeded=true statusCode=000 message=            ts=2001-02-03 01:02:03.45600
END show server time
END Lesson 12 - 2013-02-01 20:23:24.7880
```

We have the "`showServerTime`" is being called; "`succeeded`" field is equal to "`T`" in the fixed-length record; it is equal to "`true`" in the EGL record. The "`statusCode`" fields are the same. The "`message`" field is copied. As far as the timestamp, the first timestamp here is a string that I typed in; the second one is the timestamp type. It has been converted. In that conversion, it looks like right now, the last two digits are a little bit off. I'm not sure why at this point; but, that should have converted all five digits.

Now, in "`getServerTime`", this if-else statement should have been a single line of code because, in either case, we're going to assign "`result.succeeded`". So, a better way to write this is "`result.succeeded =`" and then use the boolean expression "`(response.succeeded == "T")`". So, if it's equal, it will be true; if it's not equal, it will be false. Those two blocks of code are equivalent. The second one is more specific.

Run the program again. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option.

This has been another lesson in *Essential EGL+Batch*. Thanks for watching!

Lesson 13 Nested Record

Create and use a nested record, a record within a record.

Transcript: Lesson 13 Nested Record

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

In today's lesson, we're going to deal with a number of new concepts; but, mostly we're going to extend the idea of a record and use a record within a record.

To get started, we're going to create the usual package which is called "`lab.lesson13`". We'll start with a new source file called "`ApplicationInfo`".

Listing: `lab/lesson13/ApplicationInfo.egl`

```
package lab.lesson13;

record ApplicationInfo
   name string?;
   version VersionInfo?;
   modified date?;
   notes string[];
end

record VersionInfo
   major int?;
   minor int?;
   micro int?;
end
```

This is where we're going to define a couple of records. One is called "`ApplicationInfo`". The other is called "`VersionInfo`". And, yes, we've used "`VersionInfo`" before; but, this time it will be "`lab.lesson13.VersionInfo`".

Usually, we declare version info with the open and close curly braces; but, this time we're going to use a question mark so that it is nullable. So, "`VersionInfo`" has major, minor and micro as a number like 0-0-0 or 0-1-2.

And then, "ApplicationInfo" has more information.

We're also going to create an application library called "ApplicationLib". Right-click on the "lab.lesson13" package and select the New | Source File option. Type the name of the library in the Name field and press the Finish button.

Listing: lab/lesson13/ApplicationLib.egl

```
package lab.lesson13;

import lab.lesson06.BatchLib;

library ApplicationLib

  function demo()
    showApplicationInfo();
  end

  private function showApplicationInfo()
    BatchLib.startFunction("show application info");
    info ApplicationInfo = getApplicationInfo();
    SysLib.writeStdout(info.name :: " - " :: info.version.major :: "." ::
                info.version.minor :: "." :: info.version.micro ::
                " - " :: info.modified);
    forEach(item string from info.notes)
      SysLib.writeStdout(" - " :: item);
    end
    BatchLib.endFunction();
  end

  private function getApplicationInfo() returns(ApplicationInfo)
    result ApplicationInfo{name = "Application", version = new VersionInfo{major = 0, minor = 1, micro =
2}, modified = "01/01/2001", notes =[
                "bug fixes", "new features", "stability"]};
    return(result);
  end
end
```

Let me do the usual stuff with the application library. We'll create a function called "demo" and a function called "showApplicationInfo".

Now, there's a new function called "getApplicationInfo" that returns an instance of, or returns a record of "ApplicationInfo". I'll fix up the decoration with the "startFunction" and "endFunction" functions using "BatchLib" from lesson 6.

I'll create a new record called "`info`". I'll put between the open and close curly braces a comma-delimited list of settings. Notice that we can also use version, which is another record. We can use the keyword "`new`" and continue to define a record within a record on one line, which is very efficient in EGL. So, we're filling out the application record. With an array of strings, we can continue with open square bracket, and a comma-delimited list of strings. So, we can fill out the notes, what happened in the latest release of our application.

I'll right-click somewhere in the editor, select the EGL Source | Format option to make that nice and pretty.

And then, the last piece is we have to return the "`info`" variable. We've defined it on one line; and then, we return it.

Oh, that's right. It's supposed to be a variable called "`result`", not "`info`", as a convention.

Returning to the "`showApplicationInfo`" function, we're going to create a variable called "`info`" of the type "`ApplicationInfo`", which is a record. In turn, "`info`" has a record inside called "`version`". We're going to print this out on the console.

Like we have shown before, when we do "`info`" and dot, Content Assist brings up a list of fields that are in the record. Notice here, that "`version`" is not a string, a date or a built-in type; but, it is a "`VersionInfo`" record. We select "`version`". When we put in another dot, Content Assist brings up the fields that are inside the "`VersionInfo`".

Let me do that a couple of more times. We'll put in the major version. Then, we'll do the minor version. Also, we'll do the micro version. So, you can access the parts of a record with the dot operator.

Continuing on, we're going to do the modified date. That will be it for that line. It will print the name of the application, the version and then the date it was last modified.

I'll format that to make it nice and pretty.

Now, we're going to introduce a new keyword: "`forEach`". The way that syntax is we do "`forEach`" and an open parenthesis. We define a variable; in this case it is called "`item`". The "`item`" variable is defined so that you can use it within the loop. We need to have a type. In this case, it will be a string. And then, we use the keyword "`from`" to find an array. In this case, it's "`info.notes`". If we take a look at "`notes`", remember that it's a string array. So, we can iterate through the string array with this syntax: "`forEach(item string from info.notes)`".

Within the loop, "`item`" takes on the value of each of the elements in the "`notes`" array.

Let's go ahead and create the program for this lesson, lesson 13. I'll drop in the program here.

Listing: `lab/lesson13/Lesson13.egl`

```
package lab.lesson13;

import lab.lesson06.BatchLib;

program Lesson13

   function main()
      BatchLib.startProgram("Lesson 13");
      ApplicationLib.demo();
      BatchLib.endProgram();
   end
end
```

Our program is going to call the "`ApplicationLib.demo`" function.

So, let's take it for a test drive and see what happens. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option. Click the Console icon.

Listing: Lesson 13 Console

```
START Lesson 13 - 2013-02-01 20:42:39.9970
START show application info
Application - 0.1.2 - 01/01/2001
 - bug fixes
 - new features
 - stability
END show application info
END Lesson 13 - 2013-02-01 20:42:40.5230
```

This has been another lesson in *Essential EGL+Batch*. Thanks for watching!

Lesson 14 Record Array

Work with an array of records. Use a fixed-length record to generate a report.

Transcript: Lesson 14 Record Array

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

In today's lesson, we're going to deal with an array of records. This is lesson 14. So, we'll create a package called "`lab.lesson14`".

The first thing we want to do is to create a new source file called "`SampleInfo`".

Listing: `lab/lesson14/SampleInfo.egl`

```
package lab.lesson14;

record SampleInfo
   name string?;
   sides int?;
   price float?;
end

record SampleInfoFL
   index string(11);
   name string(20);
   sides string(20);
   price string(20);
end
```

We'll put in a couple of records in here: "`SampleInfo`" and "`SampleInfoFL`" for fixed length.

"`SampleInfo`" will have three fields: "`name`", "`sides`" and "`price`".

"`SampleInfoFL`" is going to have four fields, but three are going to be the same names: "`name`", "`sides`" and "`price`". "`SampleInfoFL`" is going to be used to print the information to the console so we're going to change the types to all strings. That's it for "`SampleInfo`".

Let's create a library called "`SampleLib`". We're using New | Source File to create a library.

Listing: `lab/lesson14/SampleLib.egl`

```
package lab.lesson14;

import lab.lesson06.BatchLib;

library SampleLib

   function demo()
      arrayOfRecords();
   end

   private function arrayOfRecords()
      BatchLib.startFunction("array of records");
      list SampleInfo[] =[
                  new SampleInfo{name = "soup", sides = 0, price = 1.0},
                  new SampleInfo{name = "chicken", sides = 2, price = 2.0},
                  new SampleInfo{name = "fish", sides = 2, price = 2.5},
                  new SampleInfo{name = "vegetable", sides = 4, price = 1.5}
            ];

      displaySampleInfoArray(list);
      displaySampleInfoArrayFL(list);

      updateArray(list);
      displaySampleInfoArray(list);
      displaySampleInfoArrayFL(list);

      BatchLib.endFunction();
   end

   private function updateArray(list SampleInfo[] inOut)
      forEach(item SampleInfo from list)
         item.name = item.name + " plus 1";
         item.sides += 1;
         item.price += 0.25;
      end

      item SampleInfo{name = "soup and salad", sides = 0, price = 1.75};
      list.appendElement(item);
   end

   private function displaySampleInfoArray(list SampleInfo[] in)
      const WIDTH int = 20;
      SysLib.writeStdout(" Index---------------    Name----------------   Sides---------------    Price---------------");
      iMax int = list.getSize();
      for(i int from 1 to iMax by 1)
         s string = StringLib.format(i, "       ##&         ");
         t string = list[i].name :: StringLib.spaces(WIDTH -
                                  list[i].name.length());
         u string = StringLib.format(list[i].sides, "            ##&");
         w string = StringLib.format(list[i].price, "          ##&.&&");
         SysLib.writeStdout("  " :: s :: "   " :: t :: "   " :: u ::
                     "   " :: w);
      end
   end

   private function displaySampleInfoArrayFL(list SampleInfo[] in)
      const SPACE string(20) = StringLib.spaces(20);
      const FILL string(20) = "--------------------";
      head SampleInfoFL{};
      head.index = "Index" :: FILL;
      head.name = "Name" :: FILL;
      head.sides = "Sides" :: FILL;
      head.price = "Price" :: FILL;
      SysLib.writeStdout("  " :: head.index :: "   " :: head.name ::
                  "   " :: head.sides :: "   " :: head.price);
      iMax int = list.getSize();
      for(i int from 1 to iMax by 1)
         item SampleInfo = list[i];
         line SampleInfoFL{};
         line.index = StringLib.format(i, "      ##&    ") :: SPACE;
         line.name = list[i].name :: SPACE;
         line.sides = StringLib.format(list[i].sides,
               "             ##&") :: SPACE;
         line.price = StringLib.format(list[i].price,
               "          ##&.&&") :: SPACE;
         SysLib.writeStdout("  " :: line.index :: "   " :: line.name ::
                     "   " :: line.sides :: "   " :: line.price);
      end
      SysLib.writeStdout("  ------------------------------------------------------------------------------------------");
   end
end
```

This is our usual library. A function called “`demo`” calls a function called “`arrayOfRecords`”.

Inside of a function called “`arrayOfRecords`”, I’m creating a variable called “`list`”. It’s going to be a “`SampleInfo`” array. So, it’s going to be an array of records. To populate the record, in the square brackets with a “`new`” keyword, we’ll populate each record. Within the curly braces, we can set the value of each field. I’m going to do that several more times.

I’m going to use the EGL Source | Format option to make it look good. EGL nicely aligns them up based upon the “`new`” keyword.

Next, we’re going to display the “`SampleInfo`” array. So, we’ll be able to see what the array looks like at that point. I’ll paste that in. The “`displaySampleInfoArray`” function is using none of the fixed length strings. It’s not using a record format to display the header. It’s using all of the equations.

Let’s create a program called Lesson14.

Listing: `lab/lesson14/Lesson14.egl`

```
package lab.lesson14;

import lab.lesson06.BatchLib;

program Lesson14

  function main()
    BatchLib.startProgram("Lesson 14");
    SampleLib.demo();
    BatchLib.endProgram();
  end
end
```

And, let’s put in the usual stuff so that the “`main`” function calls the “`SampleLib.demo`” function.

We’ll take this for a test drive. I’ll click on the Console icon and this is what it looks like.

Listing: Lesson 14 First Console

```
START Lesson 14 - 2013-02-01 21:09:51.8690
START array of records
 Index---------------   Name----------------   Sides---------------   Price---------------
          1            soup                              0                 1.00
          2            chicken                             2                 2.00
          3            fish                            2                 2.50
          4            vegetable                           4                 1.50
END array of records
END Lesson 14 - 2013-02-01 21:09:52.4140
```

So, we’re able to have four records in our array. It’s printed out using all this.

Now, let’s do a more EGL way of doing things with the “`displaySampleInfoArrayFL`” function. We’ll use a fixed length record to display the data in our array. It looks something like that. We’re using a record for the header. We're using the same record for each line.

What does that look like? Let's see. I'll click on the Console icon and bring up the Console view. That's the array. We'll get back to that in a second, about formatting it.

Now, we want to update the array. After we have our initial list, we're going to update the array. After we update the array, we're going to print out the list.

In "`updateArray`", what we're going to do is iterate through the list. It started out with code that iterates by counting up; we've seen that code before. What we're going to do is change this, convert this to the "`forEach`" syntax. That happens in the real world where you really should refactor the code to use "for each" rather than "count up".

This is adding 1 to the sides and increasing the price. It's also adding another record to the end of the list.

So, let's take this for a test drive and see what we have so far.

In the second list, it's going to have one additional record.

We notice that the columns don't line up quite right. Well, it's because, in the "`SampleInfoFL`" record, this really should be fixed length strings. I'm going to change them all to 20 characters. That forces the columns to line up. That forces the header. This really does force it. The widest possible column is limited for both the header and detail lines, so they will line up. For maintainability, it's easy to go and change the record, to make a column more narrow.

Listing: Lesson 14 Console

```
START Lesson 14 - 2013-02-01 21:17:05.6340
START array of records
 Index---------------   Name---------------   Sides---------------   Price---------------
          1            soup                            0                 1.00
          2            chicken                          2                 2.00
          3            fish                           2                 2.50
          4            vegetable                        4                 1.50
 Index------   Name---------------   Sides---------------   Price---------------
          1    soup                            0                 1.00
          2    chicken                          2                 2.00
          3    fish                           2                 2.50
          4    vegetable                        4                 1.50
 ----------------------------------------------------------------------------------------
 Index---------------   Name---------------   Sides---------------   Price---------------
```

```
        1          soup plus 1                    1              1.25
        2          chicken plus 1                  3              2.25
        3          fish plus 1                   3              2.75
        4          vegetable plus 1                 5              1.75
        5          soup and salad                  0              1.75
 Index------   Name----------------   Sides---------------   Price---------------
        1   soup plus 1                    1              1.25
        2   chicken plus 1                  3              2.25
        3   fish plus 1                   3              2.75
        4   vegetable plus 1                 5              1.75
        5   soup and salad                  0              1.75
 ------------------------------------------------------------------------------------------
END array of records
END Lesson 14 - 2013-02-01 21:17:06.2490
```

This is another lesson in *Essential EGL+Batch*. Thanks for watching!

Lesson 15 Exception

Handle an exception with `try/onException`. Throw an exception with `throw`.

Transcript: Lesson 15 Exception

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

In today's lesson, which is lesson 15, we're going to talk about exceptions and how exceptions work inside of EGL. We're generating EGL to Java so we need to know a little bit about how exceptions are handled in Java. But, don't let that scare you because we'll work through it; we'll figure it out.

The first thing I want to do is create a package called "`lab.lesson15`". Inside "`lab.lesson15`", I'm going to create a library called "`ExceptionLib`".

Listing: `lab/lesson15/ExceptionLib.egl`

```
package lab.lesson15;

import lab.lesson06.BatchLib;

library ExceptionLib

  function demo()
    // unhandledException();
    shallowException();
    deepException();
  end

  private function shallowException()
    BatchLib.startFunction("shallow exception");
    try
      e AnyException{message = "uh oh"};
      throw e;
    onException(e AnyException)
      SysLib.writeStdout("exception.message=" :: e.message);
    end
    BatchLib.endFunction();
  end

  private function deepException()
```

```
      BatchLib.startFunction("deep exception");
      try
         digDeep();
      onException(e AnyException)
         SysLib.writeStdout("exception.message=" :: e.message);
      end
      BatchLib.endFunction();
   end

   private function digDeep()
      digDeeper();
   end

   private function digDeeper()
      digDeepest();
   end

   private function digDeepest()
      e AnyException{message = "oh no"};
      throw e;
   end

   private function unhandledException()
      e AnyException{message = "doh!"};
      throw e;
   end
end
```

I'm going to create a function called "`unhandledException`" and build out an exception. The syntax for that is "`e`", which is the variable for the exception, "`AnyException`". And then, inside the curly braces, we're going to put message equals "Doh!".

To throw an exception, the syntax is "`throw`" and the name of an exception variable, such as "`e`". It is not necessary to use parentheses in a "`throw`" statement.

We're going to run and see what happens when an exception is not handled at all in the EGL code. So, we need a program.

Listing: `lab/lesson15/Lesson15.egl`

```
package lab.lesson15;

import lab.lesson06.BatchLib;

program Lesson15
```

```
  function main()
     BatchLib.startProgram("Lesson 15");
     ExceptionLib.demo();
     BatchLib.endProgram();
  end
end
```

We'll put in a program, called "`Lesson15`", that calls the "`ExceptionLib.demo`" function. This time we are going to take it for a test drive, it's going to be unhandled. We right-click somewhere in the editor and select the Run As | EGL Java Main Application option.

I'll press the Console icon and bring up the console so we can see it.

Listing: Lesson 15 Console 1

```
START Lesson 15 - 2013-02-01 21:22:25.2630
doh!
eglx.lang.AnyException doh!
        at eglx.lang.AnyException.fillInStackTrace(AnyException.java:186)
        at java.lang.Throwable.<init>(Throwable.java:198)
        at java.lang.Exception.<init>(Exception.java:46)
        at java.lang.RuntimeException.<init>(RuntimeException.java:49)
        at eglx.lang.AnyException.<init>(AnyException.java:31)
        at lab.lesson15.ExceptionLib.unhandledException(ExceptionLib.java:90)
        at lab.lesson15.ExceptionLib.demo(ExceptionLib.java:32)
        at lab.lesson15.Lesson15.main(Lesson15.java:50)
        at org.eclipse.edt.javart.resources.RunUnitBase.start(RunUnitBase.java:238)
        at lab.lesson15.Lesson15.main(Lesson15.java:20)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:616)
        at org.eclipse.edt.javart.ide.MainProgramLauncher.main(MainProgramLauncher.java:78)
```

It looks like, well, a bunch of stuff. And, it really is. It's a bunch of stuff from the Java runtime. All of that, that entire highlighted section there, that is selected in from the Java runtime. It doesn't mean too much to us at the moment. But, that unwieldy message really shouldn't show up in your production quality, professional EGL code.

To prevent an unhandled exception from happening, there is a syntax in EGL that is the `try/onException` construct. If an exception occurs somewhere inside the `try` block, it will go and run the code in the `onException` block. In this case, we just want to print out an exception.

This is a very shallow exception because an exception is *thrown*—that's the verbiage for that—the exception is thrown in the same method. The exception has a "messageID" and a "message". We are just going to populate the message part. Both of these are strings, by the way.

We take the program for another test drive. Instead of the unwieldy Java message that comes up, we've reduced that to a proper, simple EGL message. Whatever we type in our EGL program, that's what is going to be displayed on the console.

Let's do a little bit more normal exception. Usually, what happens is we're deep in our code somewhere when something goes wrong. The purpose of the exception is, instead of a return value, returning an error on each function, an exception can be thrown so that, in a chain of functions, it goes all the way back up the chain, like "digDeep" calls "digDeeper", "digDeeper" calls et cetera.

Now, let's look at this when it's unhandled. We take the program for another test drive.

Listing: Lesson 15 Console 2

```
START Lesson 15 - 2013-02-01 21:27:24.4020
START shallow exception
exception.message=uh oh
END shallow exception
START deep exception
oh no
eglx.lang.AnyException oh no
        at eglx.lang.AnyException.fillInStackTrace(AnyException.java:186)
        at java.lang.Throwable.<init>(Throwable.java:198)
        at java.lang.Exception.<init>(Exception.java:46)
        at java.lang.RuntimeException.<init>(RuntimeException.java:49)
        at eglx.lang.AnyException.<init>(AnyException.java:31)
        at lab.lesson15.ExceptionLib.digDeepest(ExceptionLib.java:69)
        at lab.lesson15.ExceptionLib.digDeeper(ExceptionLib.java:66)
        at lab.lesson15.ExceptionLib.digDeep(ExceptionLib.java:63)
        at lab.lesson15.ExceptionLib.deepException(ExceptionLib.java:59)
        at lab.lesson15.ExceptionLib.demo(ExceptionLib.java:33)
        at lab.lesson15.Lesson15.main(Lesson15.java:50)
        at org.eclipse.edt.javart.resources.RunUnitBase.start(RunUnitBase.java:238)
        at lab.lesson15.Lesson15.main(Lesson15.java:20)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:616)
        at org.eclipse.edt.javart.ide.MainProgramLauncher.main(MainProgramLauncher.java:78)
```

We'll notice that in the Java code, it has a list of our functions that have been called: "`demo`", "`deepException`", "`digDeep`", "`digDeeper`" and "`digDeepest`". But, that's still the Java way of doing things, an unhandled exception.

So, let's go back and make this a handled exception. We'll wrap the "`digDeep`" function in a `try/onException` block. And again, the purpose of all this is so that you don't have to deal with errors in the signature of each function. There is a system-side, language-wide, automatic return value from every function, which is the exception.

Listing: Lesson 15 Console 3

```
START Lesson 15 - 2013-02-01 21:28:12.9620
START shallow exception
exception.message=uh oh
END shallow exception
START deep exception
exception.message=oh no
END deep exception
END Lesson 15 - 2013-02-01 21:28:13.4940
```

This is another lesson in *Essential EGL+Batch*. Thanks for watching!

Lesson 16 Log Exception

When it is thrown, log an exception on the console.

Transcript: Lesson 16 Log Exception

Welcome to *Essential EGL+Batch*. This video series features EGL Development Tools version 0.8.

In today's lesson, which is lesson 16, we're going to borrow some code from lesson 6. We're going to copy the "`BatchLib.egl`" file. We're going to fix it up with being able to log an exception. So, we need to create a new package for today, which is "`lab.lesson16`". In the "`lab.lesson16`" package, I'm going to paste in the "`BatchLib`" library from lesson 6.

Let's create a new program called "`Lesson16`".

Listing: `lab/lesson16/Lesson16.egl`

```
package lab.lesson16;

import lab.lesson15.ExceptionLib;

program Lesson16

  function main()
    try
      BatchLib.startProgram("Lesson 16");
      ExceptionLib.demo();
    onException(exception AnyException)
      BatchLib.logException(exception);
    end
    BatchLib.endProgram();
  end
end
```

This lesson 16 program is a good template for production-quality EGL programs because it always handles an exception. This is a continuation also of lesson 15 on exceptions. We'll go into a little bit more detail about what an exception is in EGL.

The purpose of this lesson is to create a function called "`logException`". It takes an "`AnyException`" as a parameter. Like lesson 15, for the moment, we're just going to print the exception message. We'll get back to this later.

Let's take this for a test drive and see what happens.

Right now in the code, it's not ever throwing an unhandled exception. So, we need to go to the "`ExceptionLib.demo`" function and make it throw an unhandled exception. By taking out the comment symbol, which is a slash-slash, now it will throw an unhandled exception. We can go to that function and take a look at that, a reminder from the last lesson.

We create an exception with the message of "Doh!" And then, the throw statement raises that exception, or throws that exception where it can be caught in a `onException` block.

The syntax for handling an exception is this `try/onException` block of code. What happens is that the code running in the try part, if it ever throws an exception, EGL is going to execute the code that's in the `onException` part. If no exception is thrown in the `try` part, then the code in the `onException` part is never used. The `onException` part is passing the exception that is thrown.

Let's take this for a test drive, now that we have an unhandled exception, and see what happens.

The "`ExceptionLib.demo`" function comes to a halt. The exception is thrown back to our program, where it's logged.

Listing: `lab/lesson16/BatchLib.egl` (Final)

```
package lab.lesson16;

import lab.lesson05.TimestampLib;

library BatchLib
   private const PREFIX_START string = "START ";
   private const PREFIX_END string = "END ";

   private functionName string?;
   private programName string?;

   function startFunction(name string in)
      functionName = name;
      message string = PREFIX_START :: functionName;
      SysLib.writeStdout(message);
   end

   function endFunction()
      if(functionName == null)
         return;
      else
         message string = PREFIX_END :: functionName;
         SysLib.writeStdout(message);
         functionName = null;
      end
   end

   function startProgram(name string in)
      programName = name;
      ts timestamp = TimestampLib.getTimestamp();
      s string = TimestampLib.formatTimestamp(ts);
      message string = PREFIX_START :: programName :: " - " :: s;
      SysLib.writeStdout(message);
   end

   function endProgram()
      if(programName == null)
         return;
      end

      ts timestamp = TimestampLib.getTimestamp();
      s string = TimestampLib.formatTimestamp(ts);
      message string = PREFIX_END :: programName :: " - " :: s;
      SysLib.writeStdout(message);
   end

   function logException(e AnyException)
      // SysLib.writeStdout("exception.message=" :: e.message);
      message string = "ERROR";
      if(e.messageId.clip() != "")
         message ::= " " :: e.messageId;
      end
      if(functionName != null)
         message ::= " in " :: functionName;
      else
         if(programName != null)
            message ::= " in " :: programName;
         end
      end
      message ::= ": " :: e.message;
      SysLib.writeStdout(message);
   end
end
```

As I said, we're going to get back to this "`logException`" function and make this do a little bit more work for us. There's a lot of opportunity to customize the “`logException`” function, make it do all kinds of things, including writing exception messages to a log file. In this function, what we want to do as an example of what you could do in the “`logException`” function, we want to create a message and print it out on the console.

This is an introduction of the “`clip`” function. What “`clip`” does is it removes all of the spaces from the beginning and end of a string. For “`messageID`”, if it is not used, it is going to equal to blank. So, this expression, where it is not equal to blank, is where the “`messageID`” has been used. We're going to add that to the message.

The message starts out with the phrase “ERROR”. If there's a “`messageID`”, it's going to be added (a space and the “`messageID`”).

With the function name, if the function name that's in the library—we'll take a look at that as a reminder with the Open on Selection option. This is a library variable that we defined.

We're going to use the concatenation operator, which is colon-colon-equals, to add something to the message. In this case, it's going to be space, “in”, space and the function name.

Then, we're going to do the same thing with the program name. If there is no function name available, we're going to add the program name. So, we'll have a nice prefix to our message.

And then, the last thing is the actual exception message. We're going to add a colon, a space and then the exception message. So, that will make it nice and readable, legible when we log it to the console.

Now that we have have a pretty decent “`logException`”—and again, this is just an example of what you can have in the “`logException`” function—we're going to run the program again, click on the Console icon and see what happens here.

Listing: Lesson 16 Console

```
START Lesson 16 - 2013-02-07 18:22:21.6620
ERROR in Lesson 16: doh!
END Lesson 16 - 2013-02-07 18:22:22.1890
```

Now, it says there was an error in lesson 16 and then the exception message. That's a little bit more specific and helps us track down what the error is.

This is another lesson in *Essential EGL+Batch*. Thanks for watching!

Lesson 17 System Property

When EGL generates to Java, the standard Java system properties, such as "`os.name`" and "`user.name`", are available to your EGL program.

Transcript: Lesson 17 System Property

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

This is lesson 17. First, we want to create a lesson 17 package. We create a new package called "`lab.lesson17`". This is part of the same "`eglprogram`" project that we've been using all along.

In the "`lab.lesson17`" package, we want to create a new source file for a library. The name of the library is going to be "`PropertyLib`".

Listing: `lab/lesson17/PropertyLib.egl`

```
package lab.lesson17;

import lab.lesson16.BatchLib;

library PropertyLib
   private const OS_KEYS string[] =["os.arch", "os.name", "os.version"];
   private const USER_KEYS string[] =["user.country", "user.dir", "user.home",
              "user.language", "user.name", "user.timezone",
              "user.zoneinfo.dir"];
   private const SEPARATOR_KEYS string[] =["file.encoding", "file.separator",
              "line.separator", "path.separator"];
   private const JAVA_KEYS string[] =["java.class.path", "java.class.version",
              "java.endorsed.dirs", "java.ext.dirs", "java.home",
              "java.library.path", "java.runtime.name",
              "java.runtime.version", "java.vendor", "java.version"];
   private const JAVAVM_KEYS string[] =["java.vm.info", "java.vm.name",
              "java.vm.vendor", "java.vm.version"];
   private pathSeparator string = ":";

   function demo()
      showProperties();
   end

   private function showProperties()
      BatchLib.startFunction("show properties");
      pathSeparator = SysLib.getProperty("path.separator");
      const KEYS_LIST string[][];
      KEYS_LIST.appendElement(OS_KEYS);
```

```
        KEYS_LIST.appendElement(USER_KEYS);
        KEYS_LIST.appendElement(SEPARATOR_KEYS);
        KEYS_LIST.appendElement(JAVA_KEYS);
        KEYS_LIST.appendElement(JAVAVM_KEYS);
        forEach(list string[] from KEYS_LIST)
            showProperties(list);
        end

        BatchLib.endFunction();
    end

    function showProperties(list string[] in)
        values string[] = getProperties(list);
        iMax int = list.getSize();
        for(i int from 1 to iMax by 1)
            // if(list[i] == "java.class.path" or list[i] == "java.library.path" or list[i] == "java.ext.dirs" or list[i] ==
"sun.boot.class.path")
            case(list[i])
                when("java.class.path", "java.library.path", "java.ext.dirs", "sun.boot.class.path")
                    showSpecialProperty(list[i], values[i]);
                otherwise
                    SysLib.writeStdout(list[i] :: "=" :: values[i]);
            end
        end
    end

    private function showSpecialProperty(key string? in, value string in)
        s string;
        if(key == null)
            s = "  ";
        else
            s = key :: "=";
        end
        pos int = value.indexOfPattern(pathSeparator);
        if(pos == 0)
            SysLib.writeStdout(s :: value);
            return;
        end

        prefix string;
        if(pos == 1)
            prefix = "";
        else
            prefix = value[1 : pos - 1];
        end
        SysLib.writeStdout(s :: prefix :: " \\");
        suffix string = value[pos + pathSeparator.length() : value.length()];
        showSpecialProperty(null, suffix);
    end

    private function getProperties(keys string[] in) returns(string[])
        result string[]{};
        forEach(key string from keys)
            value string = SysLib.getProperty(key);
            result.appendElement(value);
        end
        return(result);
    end
end
```

First, we have some names of Java system properties. We create the usual "demo" function, which calls the "showProperties" function. We'll be using the "BatchLib" library from lesson 16.

Now, this is a new concept: We're going to create an array of string arrays. So, "KEYS_LIST" is an array of string arrays. So, we have a string, and then an array and an array. Notice that the set of square brackets are there twice.

What we're going to do is take the "OS_KEYS" string array and add that to our "KEYS_LIST". The "OS_KEYS" contains the name of properties related to the operating system. We append an element of "OS_KEYS". That takes the entire string array and makes it one element in "KEYS_LIST".

We'll do the same thing for "USER_KEYS", which are user-related properties in Java system properties.

We can iterate through the list of lists of properties by using "for each list string array from keys-list", or literally, "forEach(list string[] from KEYS_LIST)".

What we want to do is be able to get a list of values based on a list of keys. So, we'll create a function called "getProperties". In the "getProperties" function, one of the key elements here is that we're going to be using a built-in EGL function called "getProperty". It takes the name of a Java system property and returns its value.

In the "getProperties" function, we pass a parameter of a string array and return a string array. By convention, we're going to use the "result" as the name of our string array that we return. We have to iterate through the list of keys. So, we'll have "for each key string from keys", or literally, "forEach(key string from keys)". (The parameter really should be called "keys"; I'll fix that up.)

For each key, we want to get the value. We'll use "SysLib.getProperty" and pass the parameter "key". Let me do that again. We have a "SysLib" library that's built-in to EGL. It has a bunch of functions. One of those functions is "`getProperty`". When you generate from EGL to Java, you're able to get Java system properties. Once we have that value, we want to add it to the list that we return; we want to put it in the "`result`".

The "`getProperties`" function is finished. It takes a list of system properties and returns a list of their values.

Back to the "`showProperties`" function, we would like to iterate through the list of keys using "`forEach`"; but, since we are dealing with two different arrays, it is better to use the normal "count up" so that we can get "`list[i]`" and "`values[i]`".

Let's take this to the next part, which is getting a program. The program name is "`Lesson17`".

Listing: `lab/lesson17/Lesson17.egl`

```
package lab.lesson17;

import lab.lesson16.BatchLib;

program Lesson17

  function main()
    try
      BatchLib.startProgram("Lesson 17");
      PropertyLib.demo();
    onException(e AnyException)
      BatchLib.logException(e);
    end
    BatchLib.endProgram();
  end
end
```

The program is the typical program like we explained in lesson 16. The essential part here is that we're going to use "`PropertyLib.demo`" to call our new "`PropertyLib`" library.

Let's take this for a test drive. Right-click somewhere in the editor and select the Run As | EGL Java Main Application.

I'll press on the Console icon. This shows system properties out of the Java

Virtual Machine.

Now, let's go back and, of course there are a lot of system properties, so we want to add more system properties. That includes separators. There's platform-specific separators. There's keys related to the Java environment. There's also Java Virtual Machine properties.

We want to add all of those properties to "`KEYS_LIST`". With a list of all those properties, we'll be able to take it for a test drive again. That will give us much more information out of the Java system properties.

Some of those properties that we're going to see are paths, such as "`java.library.path`". A path has a different delimiter based upon the platform that you're running on. So, we can run this on Windows and the path separator will be a semi-colon (;). We can run the same program without recompiling and the path separator will be a colon (:) on Linux. The "`SysLib.getProperty("path.separator")`" will give us the platform-specific separator. By default, I'm putting in a colon; but, right there, we're going to get it from the Java Virtual Machine.

What we're going to do is use that path separator as a delimiter to show the special properties, or properties that have a delimiter in them. Given a key and a value, we're going to construct a message to put on the console. The "`indexOfPattern`" function gives us the position of a pattern in a string, if it exists. If it doesn't exist, the position comes back as zero (0), so there's no separation that needs to be done; we just write out the name and value to the console.

What if there is a separator? If the separator is the very first character in the string, the position will be one (1). That means there is no value, except for blank; the value is blank. Otherwise, we'll do a calculation to get the prefix and then we'll show the prefix on the console along with a backslash to show that it's a continuation. And then, we pass the suffix, after calculating that, right back to the "`showSpecialProperties`" function. So, it's a recursive algorithm to be able to get all of the parts of a path.

Normally, what we would want to do, or instinctively what we try to do is say, if "`list[i]`" is equal to and then some value—because these are all of the special properties that require a delimiter—or "`list[i]`" is equal to some other value. And, we keep using "`list[i]`" as the thing we're comparing to. This is how you would write it out as an if statement. But, we're going to show a better way to do this, a more efficient way to do this in EGL, with the case statement. So, we don't need this if statement. We're going to rewrite that as a case statement. What we do is we take the "`list[i]`" once and put it in the case statement as the value we're comparing to and then we put a comma-delimited list in the when part, using the "`when`" keyword. So, if "`list[i]`" is equal to any one of these in the list under when, it will call the "`showSpecialProperties`" function. "`otherwise`" is another keyword. If it doesn't match that list, we can write the property as normal in system out.

Let's take that for a test drive.

This has been another lesson in *Essential EGL+Batch*. Thanks for watching!

Lesson 18 Custom Property

Define a custom property on the command line and get a custom property in an EGL program.

Transcript: Lesson 18 Custom Property

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

This is lesson 18. Now, we've already created a package in our "`eglprogram`" project called "`lab.lesson18`". So, you should create a package.

And in that package, we're going to create a new source file called "`CustomPropertyInfo`".

Listing: `lab/lesson18/CustomPropertyInfo.egl`

```
package lab.lesson18;

record CustomPropertyInfo
   name string?;
   value string?;
end
```

In the "`CustomPropertyInfo`" source file, we're going to create a record for custom properties. There'll be two fields: name and value. That's all there is for "`CustomPropertyInfo`". We'll be using that record to get custom properties.

We need to create a library called "`CustomPropertyLib`". This library will enable us to work with custom properties.

Listing: `lab/lesson18/CustomPropertyLib.egl`

```
package lab.lesson18;

import lab.lesson16.BatchLib;

library CustomPropertyLib
```

```
    function demo()
        showCustomProperties();
    end

    private function showCustomProperties()
        BatchLib.startFunction("show custom properties");
        keys string[] =["name1", "name2", "name3"];
        showProperties(keys);
        BatchLib.endFunction();
    end

    private function showProperties(keys string[] in)
        list CustomPropertyInfo[] = getCustomPropertyInfo(keys);
        forEach(item CustomPropertyInfo from list)
            SysLib.writeStdout(item.name :: "=" :: item.value);
        end
    end

    function getCustomPropertyInfo(keys string[] in) returns(CustomPropertyInfo[])
        result CustomPropertyInfo[]{};
        forEach(key string from keys)
            try
                value string = SysLib.getProperty(key);
                item CustomPropertyInfo{name = key, value = value};
                result.appendElement(item);
            onException(exception AnyException)
                SysLib.writeStdout("[WARN] " :: key ::
                                " property is missing.");
            end
        end
        return(result);
    end
end
```

What is a custom property? In lesson 17, we went over Java system properties. Custom properties are very much like system properties, in that you get them from Java in the same exact way. Custom properties are properties that you can set at the command line. You can given them names, whatever name you like. But, you should not ever name a custom property the same as a Java system property because they are in the same scope.

In "`CustomPropertyLib`", we'll create a function called "`demo`", which calls "`showCustomProperties`". We'll set up the usual decoration. We're using the "`BatchLib`" library from lesson 16.

We start out with a list of our custom property names. These will be the keys that we use to get our custom properties out of the Java system properties. And then, we'll get our "`list`", which is an array of "`CustomPropertyInfo`", from a function called "`getCustomProperties`". The idea here is to be able to reuse this function in different projects because it's generic enough. Given a list of property names, it will get those properties out of the Java system properties. It returns an array of "`CustomPropertyInfo`". Those are name-value pairs.

In the "`showCustomProperties`" function, we'll iterate through each item in the list and write that out to the console, first the name and then the value.

Inside of the "`getCustomProperties`" function, we'll use the convention of defining a variable called "`result`". It's going to be an array of "`CustomPropertyInfo`".

We're going to iterate through each key in a list of keys (or each property name in a list of property names). In this lesson, this is something we didn't do, something we probably should have done in the last lesson. In this lesson, it is possible for our custom property not to exist. So, we're going to put the functionality here inside a `try/onException` block.

We're going to create an "`item`" for this name-value pair and put the "`item`" in the "`result`". We're using the same API, which is "`SysLib.getProperty`" that we used in lesson 17.

If the property doesn't exist, an exception will be thrown. So, what we're going to do is print out a warning that a property is missing. That takes care of our custom property library.

We'll create a program. The program name is going to be "`Lesson18`".

Listing: `lab/lesson18/Lesson18.egl`

```
package lab.lesson18;

import lab.lesson16.BatchLib;

program Lesson18

   function main()
```

```
      try
         BatchLib.startProgram("Lesson 18");
         CustomPropertyLib.demo();
      onException(exception AnyException)
         BatchLib.logException(exception);
      end
      BatchLib.endProgram();
   end
end
```

In our program, we're going to do the usual decoration as we explained in lesson 16. The main thing that is going on here is that the "`main`" function is going to call the "`CustomPropertyLib.demo`" function. Because we have the `try/onException` block in our library, we should not have any exceptions come back to our "`main`" function.

We'll take it for a test drive. You'll notice that, hey, all of our custom properties are missing! That's a good thing in a sense because we didn't overlap with any of the existing system properties.

I'm going to go into the "`generatedJava`" folder and right-click on "`Lesson18.java`" and select the Run As | Java Application option. That's going to run the "`Lesson18`" program at full speed and we get exactly the same results; but, it does create a Run Configuration.

So, let's right-click on "`Lesson18.java`" and go into Run Configurations and see that there are lots of different pieces, bits of information and configuration to run a program at the command line from within the IDE. In other words, right-click on "`Lesson18.java`" and select the Run As | Run Configurations option.

On the Arguments tab, under VM arguments, we can put in the names of our custom properties. The syntax is "`-D`", without any space, the name of our custom property, equals (=) and then a value. There is no space in that entire command line option.

In the VM Arguments field, we type "`-Dname3=batch -Dname2=egl -Dname1=essential`". Now, our program, when we run it, is picking up the custom properties that we're defining on the command line.

We want to export this to a Runnable JAR file. We've exported to a

Runnable JAR file before; but, it requires a Launch Configuration. Right-click on the "`eglprogram`" project and select the Export option. Expand the Java folder, if necessary. In the Java folder, select the Runnable JAR File option and press the Next button. The name of the JAR is "`/lab/zip/eglprogram.jar`". If it exists, we press the OK button. That will create a JAR for us. And then, we can use that JAR and a Java Runtime Environment (JRE) to run our program. We don't need an IDE.

I'll switch to the Remote Systems Explorer perspective and launch a terminal. This works for Linux, Windows and Mac OS X.

The command-line syntax is "`java -jar`" and the name of our jar, which is "`eglprogram.jar`". The full command is now "`java -jar eglprogram.jar`". As we can see, we haven't defined any custom properties; so, the program doesn't pick up any.

Let's define one custom property. The option comes before the "`-jar`". The full command is now "`java -Dname1=essential -jar eglprogram.jar`". Then, we have one custom property that matches, which is "`name1`".

We do that again. The full command is now "`java -Dname2=egl -Dname1=essential -jar eglprogram.jar`". Now, we have two defined.

Last, we put in our third custom property. The full command is now "`java -Dname3=batch -Dname2=egl -Dname1=essential -jar eglprogram.jar`". Now, we have all three defined.

This is *Essential EGL+Batch*. Thanks for watching!

Lesson 19 External Type

Define and use an external type in an EGL program. Create a Java class called "`JavaSystem`" to get a complete list of property names for Java system properties.

Background

Let's say that we want to get the names of all Java system properties. At the moment, EGL has no built-in application programming interface (API) for this. In this lesson, we'll create one.

First, we need a Java class. To keep it simple, we're not going to use a class from an external Java archive file (or JAR); we're going to create one within our EGL project. The "`src`" folder is already configured for non-generated Java source code.

Second, we need to create an external type to enable our EGL library to use the Java class. The short name of the external type is the same as the Java class.

This lesson demonstrates an external type for Java. An external type enables your EGL program to use a Java class as if it were a built-in type.

Transcript: Lesson 19 External Type

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

This is lesson 19.

Pay close attention, because, ordinarily we would go directly into the "`EGLSource`" folder, but not today. Today, we're going to look into the Java source folder, the "`src`" folder. This is the Java source code that's hand coded.

What we want to do is create a new package. This is a Java package. Right-click on the "`src`" folder and select the New | Package option. The name of the package is going to be "`lab.lesson19.java`". Again, this is a Java package.

In that Java package, we're going to create a new Java class. Right-click on the "`lab.lesson19.java`" package and select the New | Class option. The name of the class is going to be "`JavaSystem`". There are quite a few options in the New Java Class wizard; but, we'll just take the defaults. Java is quite a bit more complicated and has many more options than EGL. As a result, we have a "`JavaSystem.java`" file. I've pasted in the source code to try to speed this up a little bit; but, we're going to go through this one thing at a time.

Listing: `lab/lesson19/JavaSystem.java`

```
package lab.lesson19.java;

import java.util.List;
import java.util.Properties;
import java.util.Set;
import java.util.Vector;

public class JavaSystem {
   public JavaSystem() {
      super();
   }

   public List<String> getKeys() {
      Properties p = System.getProperties();
      Set<String> h = p.stringPropertyNames();
      List<String> result = new Vector<String>();
      result.addAll(h);
      return result;
   }
}
```

Just like EGL, Java has a package. The package name is "`lab.lesson19.java`". That will come up later when we need a package name.

The classes that come with the Java Runtime include "`List`", "`Properties`", "`Set`" and "`Vector`". We're going to need all of these in order to get the properties from the Java Virtual Machine into EGL.

The name of the class is "`JavaSystem`". That includes everything from the opening curly brace ({) to the closing curly brace (}).

We have the Java constructor for "`JavaSystem`". In Java, the constructor is always the same name as the class.

We have a method called "`getKeys`", which is responsible for getting the property names. For interfacing to EGL, this particular method has no parameters and its return value type is “`List<String>`”, which is in EGL a string array. Or rather, it’s type-compatible with a string array.

In the “`getKeys`” method, we have a variable called “`p`”. Its type is “`Properties`”. The “`getProperties`” method is going to get all of the system properties, including all of the names and all of the values. So, we’ll get that out of the “`System`” class.

What we’re going to do, we’re going to ask the properties for all of the property names, using the “`stringPropertyNames`” method. That returns a set of strings. The type of the variable “`h`” is “`Set<String>`”, which is a set of strings.

We want to return this back to EGL; so, we have a variable by convention called “`result`”. It’s type is the same as the return type, which is “`List<String>`”. We’re going to assign that from a new vector, which is an array of strings in EGL, it is compatible with an array of strings where we add one string at a time. We take all of the property names returned by the “`stringPropertyNames`” method and add those property names to the list.

And finally, we can return the result. That gets the keys from the Java system properties back to our EGL program.

Now, let’s go into EGL. We’ll create a new package, an EGL package, called “`lab.lesson19`”.

In that package, we’re going to create a new source file. Right-click on the “`lab.lesson19`” package and select the New | Source File option. The source file that we’re going to create is going to define an external type. It’s called “`JavaSystem`”. That is on purpose the same name as the Java class so that it’s easy to understand and it takes less work to get it to be an external type.

Listing: `lab/lesson19/JavaSystem.egl`

```
package lab.lesson19;

import eglx.java.JavaObject;

externalType JavaSystem type JavaObject{PackageName = "lab.lesson19.java"}

   constructor();
```

```
  function getKeys() returns(string[]?);
end
```

The package for the external type is “`lab.lesson19`” in EGL.

In EGL, a constructor is simply called “`constructor`”. The function is called “`getKeys`”.

In EGL, “`JavaSystem`” is a type of Java object; so, we write “`type JavaObject`”. That “`JavaObject`” comes from “`eglx.java.JavaObject`” that we have to import when we do a Java external type.

The “`PackageName`” tells EGL where the original Java package is. That’s why it’s important to remember the “`lab.lesson19.java`” package in the Java source code.

Notice that the constructor has no body. Basically, in a Java external type, the body is defined in Java so we don’t define a body, we don’t have any statements at all, we can’t have statements in the constructor in the external type.

The same thing is true for “`getKeys`”. It’s defined somewhere else, externally to EGL. The “`getKeys`” function takes no parameters and returns an array of strings. This is an array of strings that can be null. That’s why there’s an extra question mark there. It has no body; it ends with a semi-colon (;). So, that’s all there is for the external type that enables us to use a Java class as if it were some EGL code.

Now, we’re going to create a library called “`JavaPropertyLib`”.

Listing: `lab/lesson19/JavaPropertyLib.egl`

```
package lab.lesson19;

import lab.lesson16.BatchLib;
import lab.lesson17.PropertyLib;

library JavaPropertyLib

  function demo()
    showProperties();
    end

    private function showProperties()
        BatchLib.startFunction("show properties");
        system JavaSystem{};
        keys string[] = system.getKeys();
        PropertyLib.showProperties(keys);
        BatchLib.endFunction();
    end
end
```

This is the bulk of "`JavaPropertyLib`" with all of the decoration.

In the "`showProperties`" function, we're going to create a variable called "`system`". The type is going to be "`JavaSystem`". By using the open and close curly braces, we're going to create one instance of that external type. At that point, it's going to look to the Java code. It's going to try to find a Java class called "`JavaSystem`" in a package called "`lab.lesson19.java`". So, it's going to find our class that we've defined in the "`src`" folder.

And then, it's going to call the constructor of that external type. It does that automagically. It finds the constructor and invokes it so that we have a real instance of "`JavaSystem`". The constructor, as I said before, is a method with the same name as the class in Java.

When we use that variable, it's just like any other EGL library or record, where you use the dot (Content Assist) to find out what's available. That's declared in the external type. Under "`getKeys`", it returns an array of strings and this array is nullable. At that point, where we call in EGL the "`getKeys`" function, it goes through and actually gets the properties, adds them to the result and we get back a string array in EGL.

What are we going to do with that string array? We're going to create a variable called "`keys`". It's a string array type. We're going to capture the keys from the Java system. And then, we're going to call the "`PropertyLib.showProperties`" function and pass it the array of keys.

We need to do Organize Imports. Right-click somewhere in the editor and select the EGL Source | Organize Imports option.

The "`showProperties`" function still doesn't exist; so, we need to go to the source code for the “PropertyLib” library and take a look. That “PropertyLib” library is from lesson 17. The original “showProperties” function does not take a parameter. What we want to do is refactor this function so that what’s inside the loop we’re going to put into a new function also called “showProperties” and it’s going to take a list of properties to display. We’ll not make this private so it can be used by lesson 19. The parameter is a list of strings.

And then, we paste in the code that used to be in the “forEach” loop. By refactoring it like this, we can take this functionality and use it from within another library. And yet, we haven’t damaged the functionality of the “PropertyLib” library. We’re just moving the code around a little bit. So, it will print special properties; it will print regular properties.

We can take this for a test drive. This is going to be a complete list of all of the properties, all of them, without having to type in any property names.

We’ll go ahead and create a program. Right-click on the “lab.lesson19” package and select the New | Source File option. The name of the program is “Lesson19”.

Listing: lab/lesson19/Lesson19.egl

```
package lab.lesson19;

import lab.lesson16.BatchLib;

program Lesson19

  function main()
    try
      BatchLib.startProgram("Lesson 19");
      JavaPropertyLib.demo();
    onException(exception AnyException)
      BatchLib.logException(exception);
    end
    BatchLib.endProgram();
  end
end
```

We’ll use the “BatchLib” library from lesson 16. The “main” function will call the "`JavaPropertyLib.demo`" function. It has the usual decoration.

We'll take it for a test drive. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option.

In this lesson, we're using a "`JavaSystem`" class to get system property names. This is another lesson from *Essential EGL+Batch*. Thanks for watching!

Lesson 20 Delegate Field

A delegate can be a field in a record. Use demo functions from lessons 4 thru 19.

Transcript: Lesson 20 Delegate Field

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

This is lesson 20. Lesson 20 is a bonus lesson because it's going to be quick and easy.

We've already created a new package called "`lab.lesson20`"; You need to do the same thing.

We create a new source file called "`FunctionInfo`". Right-click on the "`lab.lesson20`" package and select the New | Source File option.

Listing: `lab/lesson01/FunctionInfo.egl`

```
package lab.lesson20;

delegate onDemo();

record FunctionInfo
   name string?;
   demo onDemo?;
end
```

Inside the "`FunctionInfo`" source file, we'll start by creating a delegate. As you may remember from a previous lesson, a delegate is a reference to a function. The "`onDemo`" delegate is going to be a function that has no parameters and has no return type, so it returns nothing.

We create a record called "`FunctionInfo`". It's going to have a field called "`name`". And its going to have a field called "`demo`", which is of the "`onDemo`" type. So, you can put a reference to a function in a record with no problem at all.

That's all there is to "`FunctionInfo`".

We're going to create a new program in the "`lab.lesson20`" package called "`Lesson20`". Right-click on the "`lab.lesson20`" package and select the New | Source File option. The program is going to look something like the following listing. We'll go over it in some detail.

Listing: `lab/lesson20/Lesson20.egl`

```
package lab.lesson20;

import lab.lesson04.DateLib;
import lab.lesson05.TimestampLib;
import lab.lesson07.CharLib;
import lab.lesson08.IntLib;
import lab.lesson09.FloatLib;
import lab.lesson11.VersionLib;
import lab.lesson13.ApplicationLib;
import lab.lesson14.SampleLib;
import lab.lesson15.ExceptionLib;
import lab.lesson16.BatchLib;
import lab.lesson17.PropertyLib;
import lab.lesson18.CustomPropertyLib;
import lab.lesson19.JavaPropertyLib;

program Lesson20
   const FILLER string = "-------------------------------------";

   function main()
      try
         BatchLib.startProgram("Lesson 20");
         list FunctionInfo[] =[
                    new FunctionInfo{name = "DateLib", demo = DateLib.demo},
                    new FunctionInfo{name = "TimestampLib", demo = TimestampLib.demo},
                    new FunctionInfo{name = "CharLib", demo = CharLib.demo},
                    new FunctionInfo{name = "IntLib", demo = IntLib.demo},
                    new FunctionInfo{name = "FloatLib", demo = FloatLib.demo},
                    new FunctionInfo{name = "VersionLib", demo = VersionLib.demo},
                    new FunctionInfo{name = "TimestampLib", demo = lab.lesson12.TimestampLib.demo},
                    new FunctionInfo{name = "ApplicationLib", demo = ApplicationLib.demo},
                    new FunctionInfo{name = "SampleLib", demo = SampleLib.demo},
                    new FunctionInfo{name = "ExceptionLib", demo = ExceptionLib.demo},
                    new FunctionInfo{name = "PropertyLib", demo = PropertyLib.demo},
                    new FunctionInfo{name = "CustomPropertyLib", demo = CustomPropertyLib.demo},
                    new FunctionInfo{name = "JavaPropertyLib", demo = JavaPropertyLib.demo}
              ];
         forEach(item FunctionInfo from list)
            SysLib.writeStdout("");
            title string(20) = " " :: item.name :: " " :: FILLER;
            SysLib.writeStdout(FILLER :: title :: FILLER);
            SysLib.writeStdout("");
            item.demo();
            SysLib.writeStdout("");
            SysLib.writeStdout("");
         end
```

```
    onException(e AnyException)
      BatchLib.logException(e);
    end
    BatchLib.endProgram();
  end
end
```

First, we have a library variable called "FILLER", which value is a bunch of hyphens. I don't remember exactly how many.

We have our usual function called "main". Everything is happening in this function. We use the regular decoration, which is using the "startProgram" and "endProgram" functions from the "BatchLib" library.

We create a variable called "list". That's of the type "FunctionInfo"; it's an array of "FunctionInfo". We define it inline. We're going to use the keyword "new" to create new records for that array. Within curly braces ({}), we're going to set the field values. For the first record, the "name" field is going to be equal to "DateLib". That's the name of the function, where the function comes from. And the "demo" field is going to be set to the "DateLib.demo" function. So, "demo" is a reference to the "DateLib.demo" function. It can be called as if "demo" were calling that function directly.

This is a comma-delimited list of all of the different functions that we want to call. These are the demo functions of all these different libraries. If you haven't yet gone over a particular lesson, you can comment out the library or not type it in.

What we want to do is be able to go and do EGL Source | Organize imports, selecting the "TimestampLib" library from lesson 5 and the "BatchLib" library from lesson 16. Right-click somewhere in the editor and select the EGL Source | Organize Imports option.

So, we have a library from lessons 4, 5, 7, 8, 9, 11, 13, 14, 15, 17, 18 and 19, all of the work that we've done up until now.

What we have to do because we have two libraries that are both called "TimestampLib", for lesson 12, we're going to use the fully qualified name, which is "lab.lesson12.TimestampLib". And then, it's function is ".demo". So, there is no collision of namespace when we can use the fully qualified name.

For each "`FunctionInfo`" record that we have—so we go through the list of "`FunctionInfo`" which is an array of "`FunctionInfo`" records—we're going to print out a blank line.

Then, we're going to calculate a title. The title is going to be a fixed length string. We'll put in a space, the name of the function, a space and then the filler. That's going to be the title. We're going to print out to the console, the filler, the title and more filler so that it will always be the same length.

Then, we print another blank line.

And then, we're going to call "`item.demo`". That means that the reference to a function that's in a record, we're going to call that function. As you may remember, the "`demo`" field is a delegate, the "`onDemo`" type.

We're going to print two blank lines at the end.

Let me go through this one more time about the imports, go over it a little bit more slowly because this is an important thing. Right-click somewhere in the editor and select the EGL Source | Organize Imports option. If there is more than one library with the same short name, it gives a list that we have to choose one. Press the Next button if there is more than one. Press the Finish button if we get all of the way to the end.

We can format our source code. Right-click somewhere in the editor and select the EGL Source | Format option.

Let's take that for a test drive. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option.

This has been a bonus lesson of *Essential EGL+Batch*. Thanks for watching!

Lesson 21 Arguments

Use a Java class and external type to get command-line arguments. Pass arguments from a command-line to an EGL batch program.

Background

An application programming interface (or API) to get command line arguments is missing from the "`SysLib`" library in EGL Development Tools version 0.8. In this lesson, we build the missing functionality. This lesson demonstrates the following:

- ✓ EGL is easily extendable using the Java programming language and the "`JavaObject`" external type, and
- ✓ The Java API for EGL supports additional features, such as type conversion from EGL to Java and from Java to EGL and getting command line arguments from EGL at runtime.

In Rational Business Developer (or RBD), a commercial product from IBM, the "`SysLib`" library defines the following functions related to command line arguments.

- "`GetCmdLineArgCount`". This function returns the number of command line arguments.
- "`GetCmdLineArg`". This function requires an index parameter and returns one command line argument at a time.

This lesson gets command line arguments as an array of strings, instead of mimicking the API from RBD.

Transcript: Lesson 21 Arguments

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

This is lesson 21. In this lesson, we're going to create another Java class. We need to go to the "`src`" folder and expand it. Right-click on the "`lab.lesson19.java`" package, which is a Java package, and select the New | Package option. The name of the new package is “`lab.lesson21.java`”.

We’ll create a new class. Right-click on the “`lab.lesson21.java`” package and select the New | Class option. The name of the class is going to be “`JavaArguments`”.

Listing: `lab/lesson21/JavaArguments.java`

```
package lab.lesson21.java;

import java.util.List;
import java.util.Vector;
import org.eclipse.edt.javart.Runtime;

public class JavaArguments {
   public JavaArguments() {
      super();
   }

   public List<String> getArguments() {
      List<String> result = new Vector<String>();
      String[] list = Runtime.getRunUnit().getStartupInfo().getArgs();
      int iMax = list.length;
      for (int i = 0; i < iMax; i++) {
         result.add(list[i]);
      }
      return result;
   }
}
```

In “`JavaArguments.java`”, we’re going to step through this code. First, the package name is “`lab.lesson21.java`”. We’ll need that in the EGL external type later. From the opening curly brace ({) to the closing curly brace (}), this is a class definition.

First, we’ll create a constructor. For a constructor in Java, the method name is the same name as the class.

Now, to the really good stuff! We need a function that gets command line arguments. So, we’ll create a function called “`getArguments`”. It takes no parameters. It returns “`List<String>`”, a list of strings, which in EGL is a string array. It’s type-compatible with a string array.

By convention, we’re going to create a variable called “`result`”. It’s type is also “`List<String>`”. We’re going to initialize “`result`” to be a new vector. The new vector is also going to be a list of strings.

I’m using right-click and the Source | Organize Imports option to bring in first the “`List`” class and then the “`Vector`” class.

We need to import “`org.eclipse.edt.javart`”, which is the Java runtime for EGL in EGL Development Tools, and then a class called “`Runtime`”. We can get a list of strings from “`Runtime.getRunUnit()`”, which gives the current run unit that’s running in EGL, “`.getStartupInfo()`”, which is the start up information that’s created when you start running an EGL program, and then “`.getArgs()`”, which gives a list of strings, the arguments on the command line.

We’ll count up through the list of arguments, if there are any. This is the for loop in Java. For each string in the list of arguments, we’re going to add that to our list. That will return it as a result.

So, that’s the Java source code.

Again, “`getRunUnit()`” gets the current run unit that we’re on. And “`getStartupInfo().getArgs()`”, gives us the arguments that were passed on the command line.

In the EGL source, we need to create a new package called “`lab.lesson21`”. In the “`lab.lesson21`” package, we’re going to create a new source file called “`JavaArguments`”.

Listing: `lab/lesson21/JavaArguments.egl`

```
package lab.lesson21;

import eglx.java.JavaObject;

externalType JavaArguments type JavaObject{PackageName = "lab.lesson21.java"}

   constructor();

   function getArguments() returns(string[]);
end
```

This is going to be our external type. This external type will match the

"JavaArguments.java" class. The external type is "JavaArguments". The type is "JavaObject"; so, we need to import "eglx.java.JavaObject". Inside the curly braces ({}), we need to set the package name. If we remember, that package name is "lab.lesson21.java".

We'll define a constructor. Since it's an externally defined constructor, it ends with a semi-colon (;).

We also want to define a function called "getArguments" that returns an array of strings. It's externally defined; it ends with a semi-colon.

That's all there is for "JavaArguments".

We want to create a library called "ArgumentsLib". Right-click on the "lab.lesson21" package and select the New | Source File option.

Listing: `lab/lesson21/ArgumentsLib.egl`

```
package lab.lesson21;

library ArgumentsLib
   private const ARGUMENTS string[]{};

   function demo()
      showArguments();
   end

   private function showArguments()
      list string[] = getArguments();
      forEach(item string from list)
         SysLib.writeStdout(item);
      end
   end

   function getArguments() returns(string[])
      arguments JavaArguments{};
      result string[] = arguments.getArguments();
      return(result);
   end
end
```

We put in the usual decoration, where the "demo" function calls the "showArguments" function. In turn, the "showArguments" function gets arguments from the "getArguments" function.

In the "getArguments" function, "arguments" is the variable name and

"JavaArguments" is the type. That creates an instance of "JavaArguments" class. We use the "getArguments" function to get arguments from the command line and return that as a result.

We go on to create a program. The program name is "Lesson21".

Listing: `lab/lesson21/Lesson21.egl`

```
package lab.lesson21;

import lab.lesson16.BatchLib;

program Lesson21

  function main()
    try
      BatchLib.startProgram("Lesson 21");
      ArgumentsLib.demo();
    onException(exception AnyException)
      BatchLib.logException(exception);
    end
    BatchLib.endProgram();
  end
end
```

We'll put in the usual decoration. We'll use the "BatchLib" library from lesson 16.

The most important part of this is that the "main" function calls the "ArgumentsLib.demo" function.

We can take this for a test drive. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option. It will find no command line arguments.

Let's go to the Java source code. This is generated Java source code. Expand the "generatedJava" folder. We should be able to find in "lab.lesson21" package a source file called "Lesson21.java". We right-click on that and select the Run As | Java Application option. That will run our EGL code at full speed. There are still no command line arguments.

Let's do that again. This time, we're going to right-click on "Lesson21.java" and select the Run As | Run Configurations option. On the arguments tab, in the Program arguments field, we're going to put in some command line arguments, such as "`essential egl batch`". Press the **Run** button to run the program with those arguments. So, our EGL program can now print those arguments on the command line.

But, we're not done yet. We want to create a Runnable JAR file. We want to export a Runnable JAR file and use the launch configuration from lesson 21. Right-click on the "`eglprogram`" project and select the Export option. Expand the Java folder, select the Runnable JAR file option and press the **Next** button. Select the launch configuration called "`Lesson21 - eglprogram`". Press the **Finish** button. We'll overwrite the existing JAR file; yes, we want to overwrite it. We'll be able to run this from a command prompt outside of the IDE.

I'll switch to the Remote Systems Explorer perspective and launch a terminal for "`localhost`". I'll change to the export directory. The command is "`cd /lab/zip`".

I'll first run this without any command line arguments. The command is now "`java -jar eglprogram.jar`".

And then, where do we put the command line arguments? It's after the name of the jar. Everything that we type after the jar is a command line argument and will be picked up by our EGL program. The command is now "`java -jar eglprogram.jar essential egl batch`".

This is yet another lesson in *Essential EGL+Batch*. Thanks for watching!

Lesson 22 Connection Properties

Get properties from an open connection to a JDBC-compatible database. Configure a database connection for (1) your Eclipse workbench, (2) your EGL project and (3) your EGL program. Use the Database Development perspective to create a new database connection. Edit the EGL Deployment Descriptor (.egldd) file to add a database (SQL/JDBC) resource. Create a library called "`ConnectionLib`" to connect to a database. This lesson does not describe how to install database software or create a database.

Transcript: Lesson 22 Connection Properties

Welcome to *Essential EGL+Batch*. This video series is based upon EGL Development Tools version 0.8.

This is lesson 22. In this lesson, we want to accomplish three things. First of all, we're going to go to the Database Development perspective and create a connection to a JDBC-compatible database. Then, we're going to create a binding in EGL so that we can connect our EGL program to a JDBC-compatible database. And then, we're going to create a "`ConnectionLib`" library to demonstrate connection properties.

Part One: Database Connection

What we want to do is be able to create a JDBC-compatible database connection. In my case, the database name is "`EGLBATCH`". Host is "`localhost`". Port is "`50001`". I have a user name of "`db2inst1`" because I am using IBM DB2 version 9.7. Your settings may be different.

Press the **Test Connection** button. This is what you want to see. This is the Ping | Succeeded is what you need to see before you continue. Your choice of database really is up to you. I'm not going to go into setting up the database, creating a database, installing the database software.

In this group of lessons, what we're going to do is we're going to use EGL to create a table called "`TABLE1`". It'll have two columns. I'm using Data | Extract Sample Content so show you what the records will look like when we get further along. EGL is being used to insert records into the table, select records from the table, and eventually, to do delete queries. So, this will be covered in the next few lessons.

But, we have to have a database connection. By going this way, through the Database Development perspective, we connected our IDE (Eclipse) to the JDBC-compatible database.

Part Two: Resource Binding

Now, we want to connect EGL project to our JDBC-compatible database. We need to find our EGL Deployment Descriptor (.egldd) for the project, open it up, and then find the Resource Bindings tab. Tabs are along the bottom. We go to Resource Bindings and click the **Add** button. We want an SQL binding. I'll press the **Next** button.

We want a binding that references the workspace connection, the one that we just set up in the Database Development perspective. So, we'll select "`EGLPROGRAM`" and press the **Next** button. It will use the user ID and password that we used to connect to the database. We do not want, in an EGL +Batch program, to use a JNDI (Java Naming and Directory Interface) data source. We uncheck the Connect using a JNDI data source option. So, that's what our "`EGLPROGRAM`" binding looks like in our Resource Bindings tab. It's important to remember the name of the binding. The name of the binding doesn't have to match the name of the connection profile; it doesn't have to match the name of the database. So, it's important to remember the name of the binding.

Part Three: EGL

We go to our EGL source code. This is part three. We're going to use EGL code to connect to the database. So, we create a "`lab.lesson22`" package in EGL. We create a source file called "`ConnectionInfo`", which is just for this lesson.

Listing: `lab/lesson22/ConnectionInfo.egl`

```
package lab.lesson22;

record ConnectionInfo
   list ConnectionItem[];
end

record ConnectionItem
   name string?;
   value string?;
end
```

The “`ConnectionInfo`” record collects together information about the connection, actually using an open connection. There are a number of different properties that are associated with a connection. Each property has a name and a value. The record for a property is called “`ConnectionItem`”. So, that’s it for “`ConnectionInfo`”.

We need to create a library. The library is going to be called “`ConnectionLib`”.

Listing: `lab/lesson22/ConnectionLib.egl`

```
package lab.lesson22;

import eglx.persistence.sql.SQLDataSource;
import eglx.persistence.sql.SQLWarning;
import lab.lesson16.BatchLib;

library ConnectionLib
   private dataSource SQLDataSource? = null;

   function demo()
      showConnectionProperties();
   end

   private function showConnectionProperties()
      BatchLib.startFunction("show connection properties");
      ds SQLDataSource? = getConnection();
      info ConnectionInfo{list =[
                  new ConnectionItem{name = "autocommit", value = ds.getAutoCommit()},
                  new ConnectionItem{name = "transaction isolation", value =
decodeIsolationLevel(ds.getTransactionIsolation())},
                  new ConnectionItem{name = "warnings", value = decodeWarning(ds.getWarnings())},
                  new ConnectionItem{name = "closed", value = ds.isClosed()},
                  new ConnectionItem{name = "readonly", value = ds.isReadOnly()},
                  checkIsolationLevel(SQLDataSource.TRANSACTION_ISOLATION_NONE,
                        ds),
                     checkIsolationLevel(SQLDataSource.TRANSACTION_ISOLATION_READ_COMMITTED,
                         ds),
checkIsolationLevel(SQLDataSource.TRANSACTION_ISOLATION_READ_UNCOMMITTED,
                         ds),
checkIsolationLevel(SQLDataSource.TRANSACTION_ISOLATION_REPEATABLE_READ,
                         ds),
                     checkIsolationLevel(SQLDataSource.TRANSACTION_ISOLATION_SERIALIZABLE,
                         ds)]};
    forEach(item ConnectionItem from info.list)
      SysLib.writeStdout(item.name :: "=" :: item.value);
    end
    BatchLib.endFunction();
  end

  private function checkIsolationLevel(level int in, ds SQLDataSource? in) returns(ConnectionItem)
    s string = "support " :: decodeIsolationLevel(level);
    result ConnectionItem{name = s, value = ds.supportsTransactionIsolationLevel(level)};
    return(result);
  end

  private function decodeIsolationLevel(level int in) returns(string)
    case(level)
      when(SQLDataSource.TRANSACTION_ISOLATION_NONE)
        return("Isolation Level None");
      when(SQLDataSource.TRANSACTION_ISOLATION_READ_COMMITTED)
        return("Isolation Level Read Commmitted");
      when(SQLDataSource.TRANSACTION_ISOLATION_READ_UNCOMMITTED)
        return("Isolation Level Read Uncommitted");
      when(SQLDataSource.TRANSACTION_ISOLATION_REPEATABLE_READ)
        return("Isolation Level Repeatable Read");
      when(SQLDataSource.TRANSACTION_ISOLATION_SERIALIZABLE)
        return("Isolation Level Serializable");
      otherwise
        e AnyException{message = "'" :: level ::
                         "' is not an isolation level."};
        throw e;
    end
  end

  private function decodeWarning(warning SQLWarning? in) returns(string)
    if(warning == null)
      return("");
    end

    result string = warning.message;
    if(warning.nextException != null)
      result += "; " :: decodeWarning(warning.nextException);
    end
    if(warning.nextWarning != null)
      result += "; " :: decodeWarning(warning.nextWarning);
    end
    return(result);
    end

    function getConnection() returns(SQLDataSource?)
        if(dataSource == null)
            ds SQLDataSource?{@Resource{uri = "binding:EGLPROGRAM"}};
            dataSource = ds;
        end
        return(dataSource);
    end
end
```

In the "`ConnectionLib`" library, we going to first work on a function called "`getConnection`" because this function is something that you would continually use in the batch environment. It gets a connection. Basically, a connection is an "`SQLDataSource`". When you type this in and organize imports, you'll notice that it doesn't find the package that "`SQLDataSource`" is in; so, you have to type this in manually at the moment. It's in the "`eglx.persistence.sql`" package.

Next, we would ordinarily set up a result like this and return the result; but, by doing it this way, what we would get is a different connection every time we call the "`getConnection`" function. So, we want to make this part of the library; we want to make this a library variable. I'll call it "`dataSource`". I'm going to explicitly show, that, at the beginning, when we first have the library, the "`dataSource`" library variable is null.

Then, the first time we call the "`getConnection`" function, it will detect that the "`dataSource`" variable has never been used before and it will create one. Right here is how we create one. We use the curly braces, at sign (@) resource, more curly braces, the "`uri`" variable equals "`binding`", colon (:), and remember the binding name? That's the binding name that is used, "`EGLPROGRAM`".

Then, we set the "`dataSource`" library variable equal to "`ds`".

The "`binding:`" scheme gets us all of the information that we set up in the EGL Deployment Descriptor.

I'll paste in the source code for the "`demo`", "`showConnectionProperties`", "`checkIsolationLevel`", "`decodeIsolationLevel`" and "`decodeWarning`" functions.

I'll do Organize Imports again. It'll detect that we need a "`BatchLib`" library. We're going to use the "`BatchLib`" library from lesson 16. But, it does not automatically detect the package for "`SQLWarning`". Again, we have to type this in, at the moment, by hand. It is under the same package, "`eglx.persistence.sql`".

Now, we have the normal library that we usually use, with the normal decoration that we usually use, where the "`demo`" function calls the "`showConnectionProperties`" function. In turn, the "`showConnectionProperties`" function is going to go and get the properties from a live database connection.

First, we have a variable called "`ds`". It's type is "`SQLDataSource`" with a question mark, which means it's nullable. We're going to get that connection from our "`getConnection`" function, which we just defined.

We create a variable called "`info`". That's going to create one instance of "`ConnectionInfo`". And then, the "`list`" is going to have a collection of "`ConnectionItem`"s. These are different values that we can get from the data source, the live data source.

One of the values is auto-commit, which we get with "`ds.getAutoCommit()`". That is a property of the data source, whether it auto-commits or not.

And then, we have transaction isolation, which is another property of the data source. But, transaction isolation is actually a code. That's a machine readable thing; but, we would like to turn that into a human-readable thing. So, "`ds.getTransactionIsolation()`" returns an integer. We'd like to change that into something more like English, a human-readable language. So, we created this function called "`decodeIsolationLevel`", which takes a level and then, returns text based upon the value of that level. These constants are part of "`SQLDataSource`". It takes one of the constants and turns it into text. If it doesn't match one of the predefined constants, it's going to have the value of the "`level`" variable, and a message that says "`is not an isolation level`". It throws an exception. Hopefully, it will work properly and return a string so that we can easily read the isolation level.

And then, a similar thing with the warning. If there is a warning associated with a connection, we want to turn that warning into something human-readable. Warnings are nested so you can have a next exception and next warning. We're going to set up a variable called "`result`", by convention. Then, we're going to add warnings. It's a recursive algorithm, where "`decodeWarning`" calls "`decodeWarning`" until there are no more warnings.

We get the value from the connection whether the connection is closed and whether the connection is read-only.

Those five isolation levels may or may not be supported by your JDBC driver. So, I created a function called "`checkIsolationLevel`", for a given level and data source. You can pass a data source as a parameter and you can return a data source as a return value in EGL Development Tools. The "`checkIsolationLevel`" is going to return a "`ConnectionItem`" record. We format the name, which is going to be "`support`" and then the name of the isolation level. Use this function to determine if the JDBC-compatible database supports that transaction isolation level.

We set up a variable called "`result`". The name is "`s`". The value is going to be "`ds.supportsTransactionIsolationLevel()`". We are passing the given level. That should return a nice result.

Returning to "`showConnectionProperties`", I use that same function for each of the predefined transaction isolation levels, again, using the constants that are defined as part of "`SQLDataSource`".

And then, the next part is for each item in the connection items that are inside of the "`info`" record. So, it's basically "`forEach(item ConnectionItem from info.list)`"; we're going to write the name and value to the console. That will give us a list of connection properties, connection settings, and display them.

Next, we want to create a program called "`Lesson22`". This is just a typical program. The important thing here is that it calls

"`ConnectionLib.demo()`".

Listing: `lab/lesson22/Lesson22.egl`

```
package lab.lesson22;

import lab.lesson16.BatchLib;

program Lesson22

  function main()
    try
      BatchLib.startProgram("Lesson 22");
      ConnectionLib.demo();
    onException(exception AnyException)
      BatchLib.logException(exception);
    end
    BatchLib.endProgram();
  end
end
```

And we'll take this for a test drive. Right-click somewhere in the editor and select the Run As | EGL Java Main Application option. Switch to the Console view.

Listing: Lesson 22 Console

```
START Lesson 22 - 2013-02-23 13:42:19.8020
START show connection properties
autocommit=true
transaction isolation=Isolation Level Read Commmitted
warnings=
closed=false
readonly=false
support Isolation Level None=false
support Isolation Level Read Commmitted=true
support Isolation Level Read Uncommitted=true
support Isolation Level Repeatable Read=true
support Isolation Level Serializable=true
END show connection properties
END Lesson 22 - 2013-02-23 13:42:20.7770
```

This has been another lesson of *Essential EGL+Batch*. Thanks for watching!

Future Lessons

Outline

Learning EGL, featuring EGL Development Tools. The following lessons *might be* considered in the future.

- SQL Runnable JAR
- Decimal type
- **as** operator (conversion from one type to another) convert from float to int, from int to string to display float without everything after the decimal point, round a float, convert to int, convert to string to display rounded value. convert from string to int, convert from string to date, convert from string to timestamp, convert from string to float, convert from int to float **isa** operator (item isa int) check the type of a variable.
- SQL Load
- **any** type (create an array of any)
- Case statement
- External type: writing to a CSV file, reading from a CSV file.
- External type: reading a Java properties file.
- External type: capturing the standard output stream to a file
- External type: writing to a fixed-length file, reading from a fixed-length file.
- External type: deleting a file, does file exist?
- External type: invoking an external command
- Web service consumer: invoking a SOAP web service.
- Web service consumer: invoking a REST web service.
- Web service provider: creating a SOAP web service.
- Web service provider: creating a REST web service.
- EDT and XML. How do I work with an XML document in EGL? What does an XML document give me?
- Creating a PDF document with iText

- Redirecting standard output stream