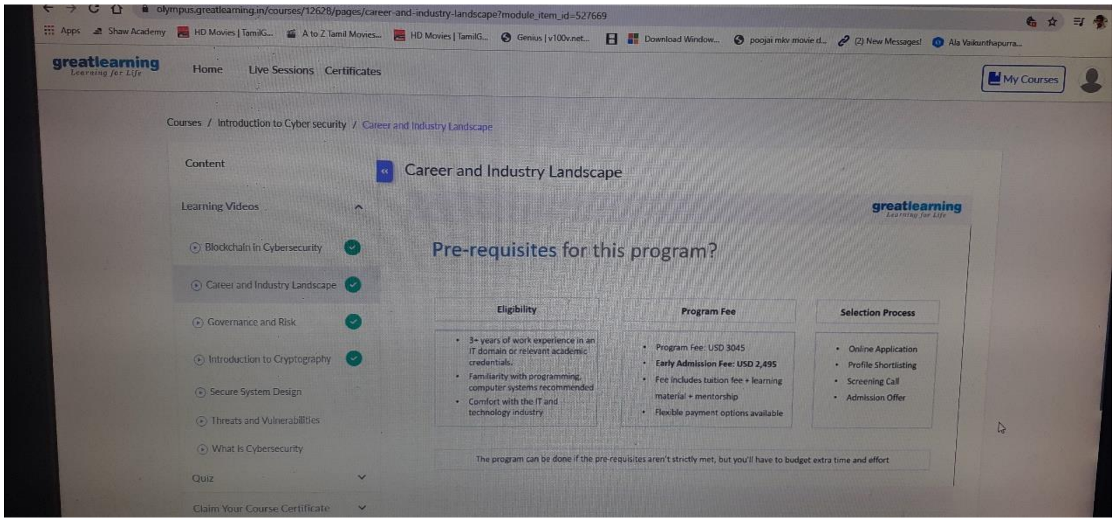
DAILY ASSESSMENT

Date:	**17-6-2020**	**Name:**	**Kavyashree m**
Course:	**Cyber security**	**USN:**	**4al15ec036**
Topic:	**Vulnerabilities & amp; Password Security, Cryptography; Message integrity**	**Semester & Section:**	**8th A**
Github Repository:	**kavya**		

FORENOON SESSION DETAILS



Career and Industry Landscape

Pre-requisites for this program?

Eligibility	Program Fee	Selection Process
• 3+ years of work experience in an IT domain or relevant academic credentials. • Familiarity with programming, computer systems recommended • Comfort with the IT and technology industry	• Program Fee: USD 3045 • **Early Admission Fee: USD 2,495** • Fee includes tuition fee + learning material + mentorship • Flexible payment options available	• Online Application • Profile Shortlisting • Screening Call • Admission Offer

The program can be done if the pre-requisites aren't strictly met, but you'll have to budget extra time and effort

Vulnerabilities &amp

A vulnerability is a hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows an attacker to cause harm to the stakeholders of an application. Stakeholders include the application owner, application users, and other entities that rely on the application.

Examples of vulnerabilities

- Lack of input validation on user input
- Lack of sufficient logging mechanism
- Fail-open error handling
- Not closing the database connection properly

amp-java

Java implementation of AMP (Asynchronous Messaging Protocol) that includes some twisted python features like reactors and deferred. More on AMP can be found at "http://amp-protocol.net/". A good place to start to get an overview of the project are the example clients and servers in the examples directory.

Supported Data Types:

- ➢ AMP Integer = java.lang.Integer or int
- ➢ AMP String = java.nio.ByteBuffer or byte[]
- ➢ AMP Unicode = java.lang.String
- ➢ AMP Boolean = java.lang.Boolean or boolean
- ➢ AMP Float = java.lang.Double or double
- ➢ AMP Decimal = java.math.BigDecimal
- ➢ AMP DateTime = java.util.Calendar
- ➢ AMP ListOf = java.util.ArrayList
- ➢ AMP AmpList = java.util.ArrayList(extends com.twistedmatrix.amp.AmpItem)

Password security

A secure password hash is an encrypted sequence of characters obtained after applying certain algorithms and manipulations on user-provided password, which are generally very weak and easy to guess. There are many such hashing algorithms in Java which can prove really effective for password security.

Simple password security using MD5 algorithm

The MD5 Message-Digest Algorithm is a widely used cryptographic hash function that produces a 128-bit (16-byte) hash value. It’s very simple and straight forward; the basic idea is to map data sets of variable length to data sets of a fixed length.

In order to do this, the input message is split into chunks of 512-bit blocks. A padding is added to the end so that it’s length can be divided by 512. Now, these blocks are processed by the MD5 algorithm, which operates in a 128-bit state, and the result will be a 128-bit hash value. After applying MD5, generated hash is typically a 32-digit hexadecimal number.

Here, the password to be encoded is often called the “message” and the generated hash value is called the message digest or simply “digest”.

Java MD5 Hashing Example

```
public class SimpleMD5Example
{
    public static void main(String[] args)
```

```
{
    String passwordToHash = "password";
    String generatedPassword = null;
    try {
        // Create MessageDigest instance for MD5
        MessageDigest md = MessageDigest.getInstance("MD5");
        //Add password bytes to digest
        md.update(passwordToHash.getBytes());
        //Get the hash's bytes
        byte[] bytes = md.digest();
        //This bytes[] has bytes in decimal format;
        //Convert it to hexadecimal format
        StringBuilder sb = new StringBuilder();
        for(int i=0; i< bytes.length ;i++)
        {
            sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1));
        }
        //Get complete hashed password in hex format
        generatedPassword = sb.toString();
    }
```

```
        catch (NoSuchAlgorithmException e)

        {

            e.printStackTrace();

        }

        System.out.println(generatedPassword);

    }

}
```

Console output:

```
5f4dcc3b5aa765d61d8327deb882cf99
```

Cryptography

The Java cryptography API is provided by what is officially called the Java Cryptograpl
Cryptography Extension is also sometimes referred to via the abbreviation JCE. The Java C
has been part of the Java platform for a long time now. The JCE was initially kept separate
US had some export restrictions on encryption technology. Therefore, the strongest encrypt
included in the standard Java platform. You could obtain these stronger encryption algorith
were a company inside the US, but the rest of the world had to make do with the weaker alg
their own crypto algorithms and plug into JCE).

Java Cryptography Architecture

- The Java Cryptography Architecture (JCA) is the name for the internal design of the
- JCA is structured around some central general purpose classes and interfaces.
- The real functionality behind these interfaces are provided by providers.
- Thus, you may use a Cipher class to encrypt and decrypt some data.

Message integrity

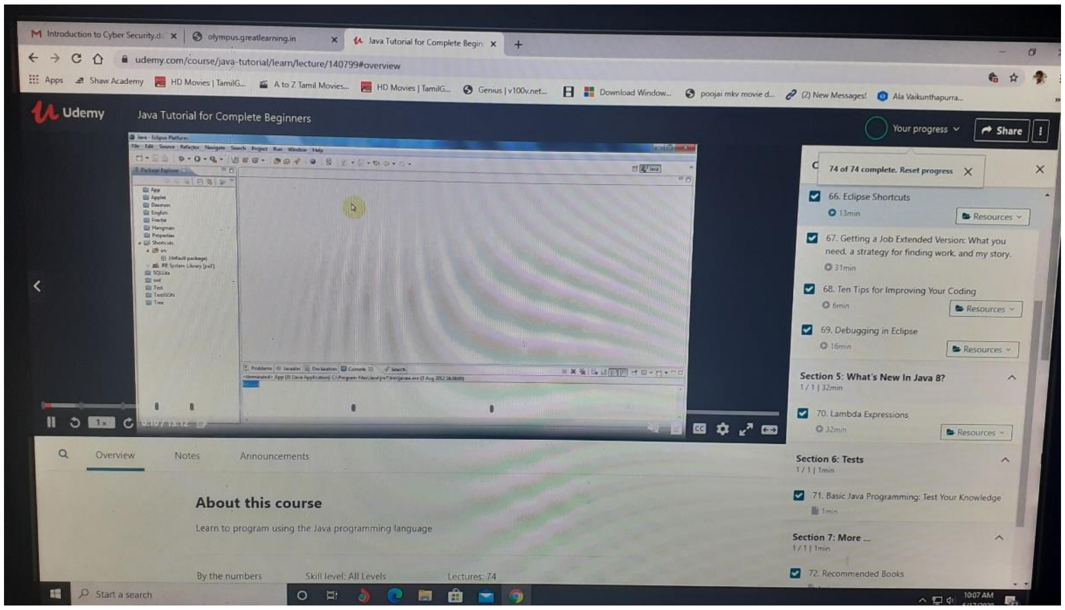
To preserve message integrity, we use message digest or hash.

The attacker would not send the modified message and the new digest to the receive.

AFTERNOON SESSION DETAILS

Date:	**17-6-2020**	**Name:**	**Kavyashree m**
Course:	**Java**	**USN:**	**4al15ec036**
Topic:	**Eclipse shortcuts , debugging in coding , what's new in java 8,lambda expressions**	**Semester & Section:**	**8th A**
Github Repository:	**kavya**		

Image of session


Udemy
Java Tutorial for Complete Beginners
Your progress
Share
74 of 74 complete. Reset progress
66. Eclipse Shortcuts
67. Getting a Job Extended Version: What you need, a strategy for finding work, and my story.
68. Ten Tips for Improving Your Coding
69. Debugging in Eclipse
Section 5: What's New In Java 8?
70. Lambda Expressions
Section 6: Tests
71. Basic Java Programming: Test Your Knowledge
Section 7: More ...
72. Recommended Books
Overview
Notes
Announcements
About this course
Learn to program using the Java programming language

Eclipse shortcut

File Navigation – Eclipse Shortcuts

- CTRL SHIFT R – Open a resource. You need not know the path and just part of the file name is enough.
- CTRL E – Open a file (editor) from within the list of all open files.

- CTRL PAGE UP or PAGE DOWN – Navigate to previous or next file from within the list of all open files.
- ALT <- or ALT -> – Go to previous or next edit positions from editor history list.

Java Editing – Eclipse Shortcuts

- CTRL SPACE – Type assist
- CTRL SHIFT F – Format code.
- CTRL O – List all methods of the class and again CTRL O lists including inherited methods.
- CTRL SHIFT O – Organize imports.
- CTRL SHIFT U – Find reference in file.
- CTRL / – Comment a line.
- F3 – Go to the declaration of the variable.
- F4 – Show type hierarchy of on a class.
- CTRL T – Show inheritance tree of current token.
- SHIFT F2 – Show Javadoc for current element.
- ALT SHIFT Z – Enclose block in try-catch.

General Editing – Eclipse Shortcuts

- F12 – Focus on current editor.
- CTRL L – Go to line number.
- CTRL D – Delete a line.

- CTRL <- or -> – Move one element left or right.
- CTRL M – Maximize editor.
- CTRL SHIFT P – Go to the matching parenthesis.

Debug, Run – Eclipse Shortcuts

- CTRL. or, – Navigate to next or previous error.
- F5 – Step into.
- F6 – Step over.
- F8 – Resume
- CTRL Q – Inspect.
- CTRL F11 – Run last run program.
- CTRL 1 – Quick fix code.

Search – Eclipse Shortcuts

- CTRL SHIFT G – Search for current cursor positioned word reference in workspace
- CTRL H – Java search in workspace.

Debugging in code

Debugging is the routine process of locating and removing bugs, errors or abnormalities from programs. It’s a must have skill for any Java developer because it helps to find subtle bug that are not visible during code reviews or that only happens when a specific condition occurs. The Eclipse Java IDE provides many debugging tools and views grouped in the Debug Perspective to help the you as a developer debug effectively and efficiently.

There are many improvements included in the latest Eclipse Java Development Tools (JDT) release included in the Eclipse Oxygen Simultaneous Release. This article will start with a beginner’s guide to start you with debugging. In the second part of the article, you will find a more advanced guide to debugging and you’ll discover what’s new for debugging in Eclipse Oxygen.

what’s new in java 8

Java 8 is here, and, with it, come lambdas. Although long overdue, lambdas are a remarkable new feature that could make us rethink our programming styles and strategies. In particular, they offer exciting new possibilities for functional programming.

While lambdas are the most prominent addition to Java 8, there are many other new features, such as functional interfaces, virtual methods, class and method references, new time and date API, JavaScript support, and so on. In this post, I will focus mostly on lambdas and their associated features, as understanding this feature is a must for any Java programmer on-boarding to Java 8.

Lambda expressions

Lambdas are succinctly expressed single method classes that represent behavior. They can either be assigned to a variable or passed around to other methods just like we pass data as arguments.

You’d think we’d need a new function type to represent this sort of expression. Instead, Java designers cleverly used existing interfaces with one single abstract method as the lambda’s type.

Before we go into detail, let’s look at a few examples.

Example Lambda Expressions

Here are a few examples of lambda expressions:

```
// Concatenating strings

(String s1, String s2) -> s1+s2;



// Squaring up two integers

(i1, i2) -> i1*i2;



// Summing up the trades quantity

(Trade t1, Trade t2) -> {

  t1. setQuantity(t1. getQuantity() + t2.getQuantity());

  return t1;

};



// Expecting no arguments and invoking another method

() -> doSomething();
```

Lambda Syntax

}

There is a special syntax to create and represent a lambda expression. Just like a normal Java method, a lambda expression has input arguments, a body, and an optional return value. This is illustrated here below:

input arguments -> body

Example

Our requirement is to create business logic for merging two trades issued by the same issuer. This example should solve the requirement by using both pre-Java 8 and Java 8 approaches.

Pre-Java 8 Implementation

Prior to Java 8, we were expected to implement the interface with concrete definitions, as shown in the test class below:

```
public void testPreJava8() {
  IAddable<Trade> tradeMerger = new IAddable<Trade>() {
  @Override
    public Trade add(Trade t1, Trade t2) {
      t1.setQuantity(t1.getQuantity() + t2.getQuantity());
      return t1;
    }
  };
}
```

Here, we created a concrete class implementing the interface and invoked the add method on the instantiated object. We used an anonymous strategy for creating the code shown above.

Although the actual logic of merging the trades is our core logic, we are forced to do some additional tasks such as implementing the interface with a class identity, overriding the abstract method, creating an instance of it and finally doing something with the instance. This “excess baggage” has always attracted critics and made developers uneasy—it’s a lot of boiler plate code and meaningless implementations and instantiations for a piece of business logic.

Once we have the instance of the class, we follow the usual process of invoking the respective methods on the instance, as shown below:

```
IAddable addable = ....;


Trade t1 = new Trade(1, "GOOG", 12000, "NEW");

Trade t2 = new Trade(2, "GOOG", 24000, "NEW");


// using the conventional anonymous class..

Trade mergedTrade = tradeMerger.add(t1,t2);
```

The business logic is intertwined with the technical garbage. The core logic is very much tied to the implementing class. For example, instead of returning a merged trade as in the above case, if I may have to compare and return the big trade, I have to sigh a bit, grab a coffee, sneeze loudly, moan, groan and roll up my sleeves and get ready to rewrite the code logic.

Note also that all our test cases will start failing too once we change the logic!

What if I have to support dozens of such requirements? Well, either I have to hack the current method to create the additional logic path using a control statement (if-else or switch) or create a new class for each piece of logic. Tightly coupling your business logic to its implementing class is asking for trouble—especially if we have fickle-minded business analysts and project managers. Surely there must be a better way to make it work.

Java 8 Implementation

Creating multiple behaviors using anonymous classes is not impossible, but it’s not ideal. We can simplify this problem by using lambda functions to represent various behaviors. So, for example, we can write a lambda expression to sum up the trade’s quantities, another for returning a large trade, another expression for encrypting trade data, etc. All we do is write a lambda expression for each of the requirements and send it over to the class that expects the lambda.

For example, take a look at this list of lambda expressions for our requirements:

```
// Summing up the trades quantity

IAddable<Trade> aggregatedQty = (t1, t2) -> {

  t1.setQuantity(t1.getQuantity() + t2.getQuantity());

  return t1;

};
```

```
// Return a large trade

IAddable<Trade> largeTrade = (t1, t2) -> {

 if (t1.getQuantity() > 1000000)

   return t1;

 else

   return t2;

};


// Encrypting the trades (Lambda uses an existing method)

IAddable<Trade> encryptTrade = (t1, t2) -> encrypt(t1,t2);
```

As you can see, we have declared lambdas for each our respective functionalities. The method can now be modeled to expect an expression, as shown below:

```
//A generic method with an expected lambda

public void applyBehaviour(IAddable addable, Trade t1, Trade t2){

  addable.add(t1, t2);

}
```

Note that the method is generic enough that it can apply functionality to any two trades with the given behavior using the given lambda expression (the IAddable interface).

Now, the client has the control of creating the behaviors and pass it on to the remote server for application of them. This way, the client cares about what to do, while the server cares about how to do it. As long as the interface is designed to accept the lambda expression, the client can create a number of those expressions according to its requirements and invoke the method.

Before we sum up, let’s take an existing Runnable interface, which comes with lambda support, and see how it can be used.

Runnable Functional interface

The most popular Runnable interface has one method, run, which takes no arguments and returns nothing. Perhaps the reasoning behind this is that a piece of logic is to be run in a separate thread to speed up the process.

The new definition of Runnable interface, followed by an example implementation using an anonymous class, is given below:

```
// The functional interface

@Functional Interface

public interface Runnable {

  public void run ();

}


// example implementation

new Thread (new Runnable () {
  @Override
  public void run () {
    sendAnEmail ();
  }
}). start ();
```

As you can see the above way of creating and using the anonymous class is very verbose—and unsightly too. Apart from sendAnEmail () in the run method, everything else is redundant boilerplate code.

The same Runnable can now be re-written to use a lambda expression, as shown below:

```
// The constructor now takes in a lambda
new Thread (() -> sendAnEmail ()). start ();
```

The lambda expression () → sendAnEmail () highlighted above is passed to the constructor of the thread. Note that this expression is effectively a piece of code (an instance of Runnable) that carries certain behavior (make sure to always send an email in a new thread).

Looking at the expression, we can deduce the type of the lambda—in this case it is Runnable, as we all know that the thread constructor accepts a Runnable. If you've noticed the redefined interface definition, Runnable is now a functional interface and hence tagged with the @Functional Interface annotation. Now the lambda expression can be assigned to a class's variable as Runnable r = () → sendAnEmail () just like how we declare and assign variables.

This is the power of lambdas—they allow us to pass a behavior to a method, assigned to a variable (free floating) from another method, just like we do when passing arguments that carry data.

Let’s suppose we have a server side class called AsyncManager whose sole purpose is to execute requests on different threads. It has a single method, run A sync, which accepts a Runnable, as shown below:

```
public class AsyncManager {
  public void run sync (Runnable r) {
    new Thread(r);
  }
}
```

The client now has the ability to create a plethora of lambda expressions based on his requirements. For example, see the various lambda expressions that can be passed on to the server-side class:

```
  manager.runAsync(() -> System.out.println("Running in Async mode"));
  manager.runAsync(() -> sendAnEmail());
  manager.runAsync(() -> {
    persistToDatabase();
    goToMoon();
    returnFromMars();
    sendAnEmail ();

  });
```