Lynda.com - Java Essentials Training 2016

**Chapter 4 - History**

Started on 1991 by Sun Microsystem – Green Project

New Portable Programming Language that can run on multiple OS without recompiling

Original Name: Oak – Oak Tree outside the windows of developer’s office

First release named Java (1995)

Original Tag: Write Once Run Everywhere

1996 – First JDK 1.0 (Java Development Kit)

1997 – Object Oriented Nature, JavaBeans, JDBC, RMI (Remote Distributed Systems), Reflection

1998 – J2SE 1.2 (Java 2 Standard Edition); Swing GUI, Collections Framework, JIT Compiler,

2000 – J2SE 1.3; Hotspot JVM, JNDI Interface, Sound API, Debugging Architecture

2002 – J2SE 1.4; Regular Expressions, IPv6 Network, Logging API, XML and XSLT, Security and Cryptography, Java Web Start

2004 – J2SE 1.5; Generics (Strongly data type), Enumerations, Variable Arguments, Foreach enhancements

2006 – Java SE 6; Performance JDBC 4.0

2010 – Oracle buys Sun

2011 – J2SE 7; Strings in switch, Try-catch enhancements,

2014 – Lambda expressions, method references, collections, new date/time API, Nashorn JavaScript engine

Android: check Java 6 and Java 7

**Chapter 5 – Principles of Java**

* Simplicity – consistent, never deviates from the way the language is architected
* Object-Oriented – encapsulation, inheritance, polymorphism
* Familiar with C and C++
* Robust and Secure – object which consists of methods/functions and properties/fields for ease of debugging
* High-performance – just as fast as C Language
* Interpreted at runtime rather than run directly as machine code
* Multi-threaded – can do more than one thing at a time
* Dynamic – change datatypes at runtime as long as its compatible
* Interpreted Language
* Compiles to bytecode instead of machine language
* Compiled application is portable between platforms without recompiling
* Class-based
* Run in a protected virtual machine (JVM)

Java Editions

* Java Platform, Standard Editions (SE)
  + Core language and runtime environment
* Java Platform, Enterprise Editions (EE)
  + Standard for building industrial-strength web applications
* Java Platform, Micro Edition (ME)
  + Micro-controllers, sensors, and mobile devices (e.g. simcards on smart phones)

Java SE Runtime Environment (JRE)

* JRE is the runtime environment
* Free and downloadable from Oracle
* Runs Java Applications on desktop, borwser, and server
* Must be updated regularly to keep up to date
* Mobile OS have own runtimes

Java SE Development Kit (JDK)

* Compiles and package Java Applications
* Includes tools for compilation and packaging
  + Java: runtime
  + Javac: compiler
  + Javadoc: docs builder
  + Jar: archive builder

**Chapter 6 – Java syntax and compilation**

* All code is defined in classes
* Classes are defined in source code files with .java extensions
* Javac command compiles Java code into bytecode
* Java runs bytecode files

Package Declaration: domain name in reverse order (com.example.java)

Class Declaration: each source code must have at least one public class

Main Method: console application must have a main method (public static void(String[] args))

Executable Code: code inside the main method

Case Sensitive: must be unique within the scope; firstname, firstName, FIRSTNAME

Whitespace does not affect interpretation of code (spaces, tabs, and line feeds). Collapsed during compilation

Semicolon – ends the statement

Keywords cannot be used by class or other identifiers

Identifiers must start with alpha characters or underscore

Class names starts with uppercase characters – code convention

Methods and variables start with lowercase character – code convention

Constants are all uppercase – code convention

* Public static final String FIRSTNAME = “John”;

**Chapter 7 – Memory Management**

* Java manages memory automatically. No need to allocate and deallocate memory
* JVM has feature of memory management

Stack and Heap Memory

Primitive Variables can be stored on both stack and heap

Complex variables are always stored on heap

Variables within method expires along the method. After the method executes, the variable is then deallocated

If the variable is set to null, the variable is explicitly dereferenced

Garbage Collector runs on its own thread

**Chapter 26 – Boolean values and expressions**

Primitive variables are translated to equivalent strengths.

Boolean starts at false

Local variables must be initialized.

Negation operation - take any value and reverse the value by adding ! mark.

You can also create Boolean values from expressions

**Chapter 27 – Character Values**

Java distinguished character from string

Character is primitive type. String is complex object that contains many characters

Literals for characters are wrapped in single quotes while strings are wrapped in double quotes

Characters with Unicode escape sequence literal can also be declared

Character.toUpperCase

Character is primitive variables and to manipulate it, you can use Character class

**Chapter 28 – Java Operators**

Java syntax is based on C-style (C,C++,C#,JavaScript)

Different types of operators:

* Equality Operators – compare values to each other
  + == equality; compares the actual values in primitive, reference variables compares whether the two variables points to the same object not necessary whether they have the same values
  + != inequality;
  + >, <, >=, <=
  + instanceof – class membership
    - if (s instanceof java.lang.String)
* Assignment Operators – assign values
  + Int intValue = 10;
  + Type and name on left while value on the right
* Mathematical Operators
  + +, -, \*, /, %
  + Increment(++) and decrement(--)
  + intValue += 5; intValue -= 5;
  + Postfix – intValue ++; evalues the value before executing the math
  + Prefix – ++ intValue; executing the math before evaluating the value
* Logical Operators
  + && - and
  + || - or
  + ?: Ternary
    - String s = condition ? trueValue : false;

Comparing String

* Strings cannot be safely compared using the equals or not equals. Rather use equals method in String class

**Chapter 29 – Object Data Types**

Object

* Instance of class
* Non-primitive variables are references to object
  + When creating a variable and point it on an object. The variable is not the object itself but references the object
* Can have multiple references

Instantiation

Instance variable – is not the member of class itself but the instance of the class

String

* Is an object
* Member of String class and member of java.lang
* Example and means the same thing
  + String string1 = new String(“Hello!”);
  + String string1 = “Hello!”;
  + This creates a string variable
* Array of characters
* Immutable
  + Once instantiated and assign, you cannot change it but derefences the object and creates a new object with new value

When appended a number, they’re automatically turned to string during the compilation process

**Chapter 31 – Converting Primitive values to String**

10\_000\_000 – doesn’t affect the compiler read it. Introduced in Java 7.

**Chapter 32 – Building a string from multiple values**

Instead of manually concatenating strings, use StringBuilder which belongs to java.lang package

StringBuilder:

* One object instead of multiple object
* each method called returns reference to String Object
* length(); method of the StringBuilder class that returns the length of the string
* delete(); deletes the content of the initialized StringBuilder
  + has 2 arguments, the starting character and the ending character
  + when the builder object is passed in, the toString() is automatically called

Scanner Class

* Belongs to java.util
* Collects data user provides

**Chapter 33 – Comparing string values**

Strings are immutable – once created, the value can’t change

Interning – finds the original value, finds the match, and interns

When comparing strings, to get the accurate output, use equals or equalsIgnoreCase and not ==

**Chapter 34 – Formatting numeric values as string**

**Chapter 35 – Working with Date and Time**

There are two complete datetime API in Java; Original mid 90’s and Java 8

Enterprise Editions and Java 8 – use Java 8

Android – use older

Date Class

* Belongs to java.util

Old API

* Month starts (0 January -11 December)

New API

* Month starts (1 January – 12 December)

Format method looks for class temporalAccessor – superclass or ancestor of specific class to be used

**Chapter 39 – Undestanding syntax errors vs runtime exceptions**

Syntax Errors

* Deal before compile and run the application

Runtime Exceptions

* Exceptions that occur while the application is running

**Chapter 40 – Debugging with IntelliJ IDEA**

Creating breakpoints:

1. Click once to the right of the line number
2. A breakpoint icon will show (red circle)
3. Useful information will show when the mouse is hovered over the breakpoint

Run debug mode:

1. Options:
   1. Menu: Run > Debug ‘Main’
   2. Click the Debug Icon (Icon with Bug)
2. A debugger tab and console tab will show at the bottom
3. To step through debugging one line at a time, click the ‘Step Over’ icon. As the variables are set, you can see it in couple of ways
   1. The value will display in the right of the code
   2. Move the cursor over the variable, the value will show in a moment
   3. Add the variable in the watches window.
      1. Select the variable by double clicking
      2. Right click
      3. Choose ‘Add to watches’
      4. The watches window is shown in the bottom. The changes will show there
4. Click the ‘Step Over’ icon again to step over each line of code
5. Add another breakpoint on printline
6. Set the condition to i==7 (Only suspend the code when the condition is true)

**Chapter 41 – Handling exception with try catch**

To surround a block of code with try catch:

1. Select the code to be surrounded with try catch
2. Menu: Code > Surround with
3. Choose try catch
4. The code will be wrapped in try catch

By default, the object that’s passed in the catch block is typed as an exception.

**Chapter 42 – Creating multiple catch blocks**

Note: Identify all the different exceptions the code might generate and write catch block for each exception to make the code more robust and more likely to survive when the application is running

**Chapter 43 – Throwing custom exceptions**

Generate custom exceptions and custom messages

**Chapter 44 – Programming conditional logic**

Conditional code is commonly called “if then” or “If else”

* If statements can have multiple else if but can only have 1 else

**Chapter 45 – Using switch statements**

Don’t need pair of braces in ‘case’ instead place the statements after the colon

Break

* finished the whole statement and want to go to the end
* Include in all the case

Java 7 – can now work with strings

**Chapter 47 – Creating reusable code with methods**

Methods

* Called as functions or sub-routines
* Declared as members of a class
* E.g. main method
  + Signature of method to have to start the application on console
  + Not all java applications use this (android, java ee)
  + <visibility> <type> <return data-type> <name> (<argument-type> <argument-name>){
    - Executable….
  + }

**Chapter 48 – Overloading methods with different signatures**

e.g.

static double addValues(String s1, String s2);

static double addValues(String s1, String s2, String s3);

static double addValues(String … values)

* … means the user can pass as indefinite number of values

**Chapter 49 – Passing arguments by reference or by value**

is arguments passed in to methods by copy or by reference?

* By copy – passing a variable in to a function or method and within the method a copy of variable is received. *It’s not attached in to the original value*
* By References – referring to the original object that existed outside the object and if there’s any changes, it will affect what’s passed

Java passes arguments by copy

Passing Primitives Value

Changes inside the method won’t change the original

**public static void** main(String[] args) {  
 **int** original = 10;  
 System.***out***.println(**"Original Value: "** + original);  
 *increment*(original);  
 System.***out***.println(**"Original after: "** + original);  
}  
  
**public static void** increment(**int** inMethod) {  
 inMethod++;  
 System.***out***.println(**"In method: "** + inMethod);  
}

output:

Original Value: 10

In method: 11

Original after: 10

Primitive Values Wrapped in Complex Object

Making a copy of array passed in but the new array is still referencing the original values outside the scope

**public static void** main(String[] args) {  
 **int**[] original = {10, 20, 30};  
 System.***out***.println(**"Original Value: "** + original[0]);  
 *increment*(original);  
 System.***out***.println(**"Original after: "** + original[0]);  
}  
  
**public static void** increment(**int**[] inMethod) {  
 inMethod[0]++;  
 System.***out***.println(**"In method: "** + inMethod[0]);  
}

output:

Original Value: 10

In method: 11

Original after: 11

Object Variables are References

* Reference variables point to a location(object) in memory
* When variable is passed in the function, a new variable is made but pointing to the original object
* Have now two references pointing to the same object

In most complex object, changing the object in the method would change the object outside but string values are immutable. Once declared, they cannot change.

When string is passed in to a method, a copy of string is created within the method and detached from the original value outside the method.

**Chapter 52 - Using simple arrays**

* Brackets in array can be placed after the data type or after the variable name but not both. Preferably after the data type.
* Once array size has been set, it cannot be changed

Arrays.sort

* Belongs to java.util.Arrays;
* Sort the array

System.arraycopy

* Copy contents of an array into another
* Parameters: source, source-starting-index-position, destination-array, destination-starting-index-position, and length

**Chapter 53 - Using two-dimensional array**

String[][] states = new String[3][2];

* 3; outer array
* 2; inner array – each from outer array can have 2 items in inner array

States.length – length of the outer array

**Chapter 54 – Managing resizable arrays with ArrayList**

The Java Collections framework is a set of interfaces and classes that make it easy to manage data in applications.

Classes Categories Interfaces:

* Set
* List
* Deck
* Map

Commonly used categories:

ArrayList

* Implements list

Hashmap

* Implements map

List Interface

* Member of java.util
* Can contain as many items as you want
* Good practice to declare the type of item it will contain by using generic notation
  + Diamond operator <> - declare the type of items to be added to list. Use the helper class

e.g.

List<String> list = new ArrayList<>()

* List: list interface
* String: generic notation enclosed in diamond operator.
* list: variable name
* ArrayList: Implementing Class/Concrete Implementation of Class
  + Member of java.util
* <>() – nullable values
  + <>: contains the items data type
  + (): integer contains the item quantity the arraylist will contain

List.get();

* Returns the value of key given

List.add();

* Add new items in the list

List.indexOf();

* Returns the key of given value

List.remove();

* Removes the item from the list

**Chapter 55 – Managing unordered data with HashMap**

Hashmap

* implementation of interface named map
* Each item in map is key – value pair. The datatypes is anything you want
* Frequently, key is string and second value is number, string or any sort of complex object

e.g.

Map<String, String> map = new HashMap<>();

* Map; map interface
* <String, String>; key value pair
* Map; variable name
* New HashMap<>(); concrete implementation

map.put(“California”, “Sacramento”);

* Inserting values in map. Key – value pair

map.get(“Oregon”);

* Fetch the value of map

map.remove(“California”);

* Remove the key-value pair matching the given key

**Chapter 56 – Looping through collections with iterators and foreach**

Iterator Object

* Commonly way to loop to collection

ArrayList iterator

* Create an iterator object
  + Iterator<String> iterator = list.iterator();  
    **while**(iterator.hasNext()) {  
     System.***out***.println(iterator.next());  
    }
* Get the object, create a loop, prints
* foreach
  + **for**(String value : list) {  
     System.***out***.println(value);  
    }
* Java 8 method reference
  + list.forEach(System.***out***::println);

Hashmap Iterator

* iterator
  + Set<String> keys = map.keySet();  
    Iterator<String> iterator1 = keys.iterator();  
    **while**(iterator1.hasNext()) {  
     String key = iterator1.next();  
     System.***out***.println(**"The capitol of "** + key + **" is "** + map.get(key));  
    }
* Foreach
  + **for**(String key : keys) {  
     System.***out***.println(**"The capitol of "** + key + **" is "** + map.get(key));  
    }

**Chapter 57 – Encapsulation**

* packaging complex functionality for ease of use
  + break the code into individual classes and grouping the functionality
* Access to individual functions can be restricted
  + Each class can be placed in particular classes
  + Marked with public, private, protected
* Hide complex functionality in method
* True nature of encapsulated data may also be hidden
  + Hiding the mechanism, the data is stored
  + E.g. using xml to store data in file at first then change to json format. If encapsulation is properly implemented, other parts can still call the method but reduce the risks of breaking the application

Benefits

* Breaking functionality into smaller, maintainable units
* Grouping functionality and data together
  + Wrapping in a java class
* Supporting testing of software at a granular level
  + Unit testing

**Chapter 61 – Storing data in instance variables**

Model class

* common Java vocabulary
* defines data model

**Chapter 62 – Declaring instance and static methods**

Instance method

* method without the static keyword
* class that encapsulates data typically have setters and getters

getters – return value

setters

* receive argument
* this; separates the variable member of instance and the arguments

**Chapter 63 – Constructors**

When instance of class Is created, the constructor is typically called

e.g.

Constructor is called three (3) times

Olive olive1 = **new** Olive();  
olives.add(olive1);  
Olive olive2 = **new** Olive();  
olive2.setOil(1);  
olives.add(olive2);  
Olive olive3 = **new** Olive();  
olive3.setOil(4);

* You can have many constructors but make sure that the signature or arguments/data types differ from each other

**Chapter 64 – Using static variables as constants**

Java developers create constants using final keyword

* Final; variables value can only be set once

Can help eliminate the errors that literal strings prone

**Chapter 65 – Enum**

Can be created in separate file or nested enum

Instead of class, Enum uses enum

Every java class and enum has toString();

Advantages:

If you decide to add value later on, you can add it in enum

**Chapter 67 – Understanding inheritance and polymorphism**

Inheritance

* Relationship between classes in which you can extend or inherit one class to another. Not all OOP are the same. C++ can have multiple inheritance while in Java, only one inheritance is permitted. Each class can only inherit one class
* Parent/Child
* Base/Derived
* Superclass/Subclass – preferred java vocabulary

Polymorphism

* Object can be addressed as supertype or subtype
* E.g. List and ArrayList; list is interface while ArrayList implements interface
* Write methods that can accepts supertype as arguments and take an object that’s native instance of subtype
* And pass into the method
* Increase code flexibility and reusability

Contract-based Programming

* You can think of a method that accepts supertype as arguments as contract of a sword
* As long as the object extends this superclass, I’ll accept it. That’s a guarantee that the subclass will implement the functionality that I’m looking for
* Inheritance-model of Java

e.g.

Olive class extends the Object class that’s why it has a toString() method. Olive is also superclass and has subclasses in the example

Superclass

* Doesn’t need special code
* If a class isn’t final, it can be extended
* All fields and methods are inherited unless marked as private
  + Common practice:
    - Mark fields as private and add public setter/getter methods
    - Mark method as protected or public to be called in other places in application