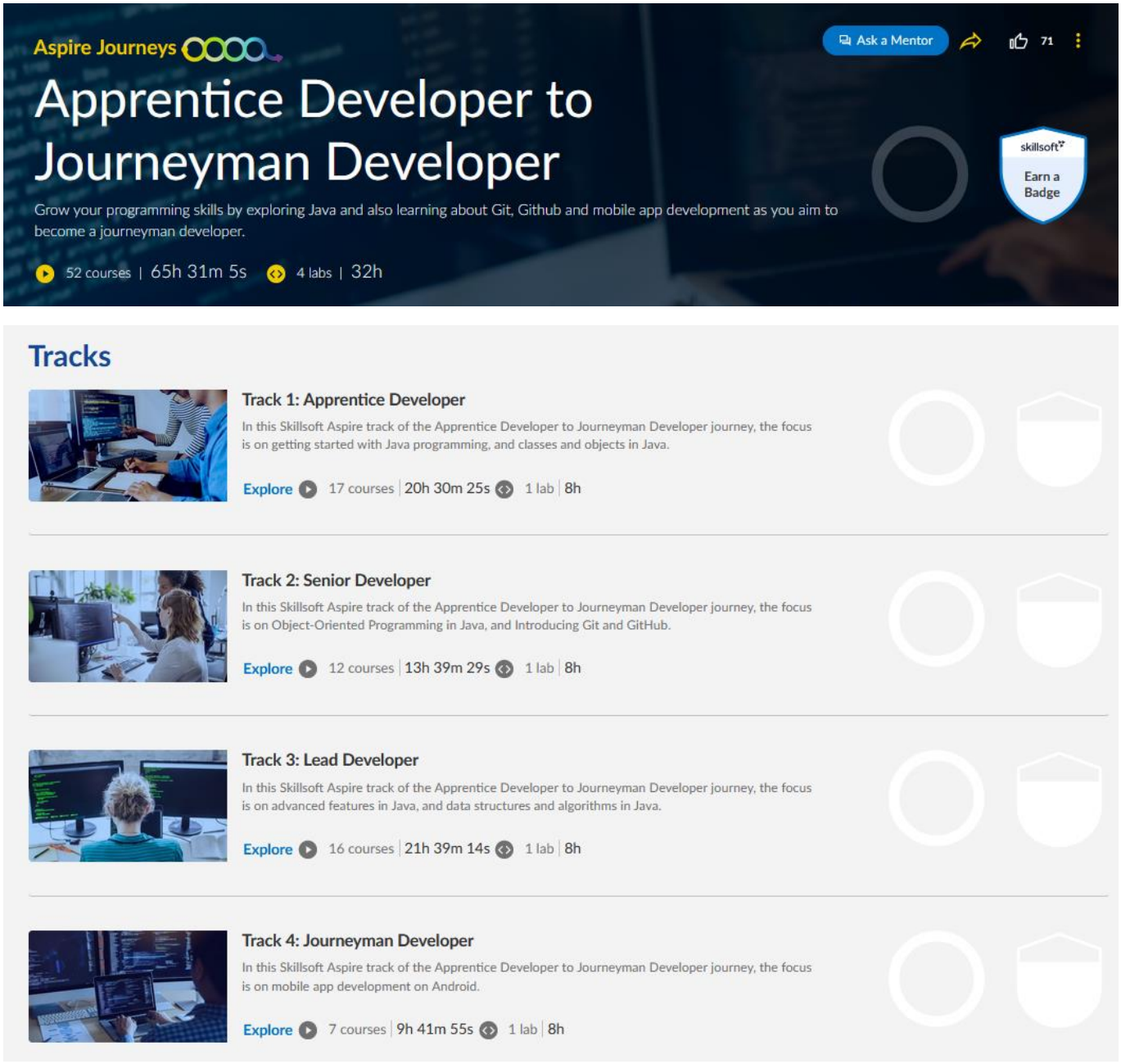
# Apprentice Developer to Journeyman Developer

## SKILLSOFT ASPIRE JOURNEY

skillsoft | percipio




## PREREQUISITES

In order to fully profit from the potential of this Aspire Journey we recommend the following prerequisite skills:

- Familiar with programming
- Familiar with mobile application development



Java Programming: Introduction

Objectives

- list the reasons for Java's popularity
- install Java on a Unix or Mac system
- install the Eclipse IDE on a Mac-based system
- install the IntelliJ IDE on a Mac-based system
- install Java on a Windows-based system
- install the Eclipse IDE on a Windows-based system
- install the IntelliJ IDE on a Windows-based system
- customize IntelliJ to incorporate your individual preferences
- use IntelliJ to examine the contents of .class files of compiled bytecode
- utilize both single-line as well as multi-line comments in Java


Java Programming: Working with Primitive Data Types

Objectives

- identify differences between primitive and object types in Java
- create and use variables of the primitive data types - byte, boolean, short, int, long, float, double, and char
- work with variables of double, char, and String types in Java
- create arrays of both primitive and object types, print their contents, and access individual elements
- create and use enums (enumerated data types) in Java
- initialize and modify variables of primitive types
- apply arithmetic operators to variables of primitive types
- describe the += self-assignment operator
- describe and use the -=, *=, %= and %= self-assignment operators


Java Programming: Arithmetic & Logical Operations

Objectives

- describe the rules of boolean arithmetic and implement them in Java
- implement comparison operations using primitive types
- use the logical AND, logical OR, and logical negation operators
- use parentheses to alter the order of precedence of execution in arithmetic and logical expressions
- identify use-cases for the unary increment and decrement operators
- debug complex constructs built using the unary increment and decrement operators
- perform conversions between variables of primitive types


Java Programming: Working with Strings & Wrapper Objects

Objectives

- compare objects for semantic equality
- create String objects using different input arguments and forms of memory allocation
- differentiate between equality tests based on the == operator and .equals
- initialize local variables of primitive as well as non-primitive types
- use the StringBuffer object to perform string operations
- differentiate between the StringBuffer and the StringBuilder object, and determine the right choice for your specific use-case
- create wrapper objects such as Integer, Float, etc. to encapsulate primitive (non-object) data types
- perform type conversions using methods available in various wrapper objects
- extract primitive data types back from wrapper objects of type Integer, Float, etc.


Java Programming: Conditional Evaluation

Objectives

- modify control flow in a program using the if construct
- use the else clause to provide an alternate code path to control program flow
- demonstrate additional applications of using the else clause to provide an alternate code path to control program flow
- use nested if-else conditions to deal with mutually exclusive conditions
- use nested if-else clauses to deal with more complex combinations of user input
- use strings in if-else constructs, relying on the .equals method and avoiding the == operator
- use the switch construct rather than nested if-else statements to control program flow
- use the default keyword to deal with unexpected values in switch evaluation
- use enums (enumerated types) along with switch statements
- use string variables in switch statements


Java Programming: Iterative Evaluation

Objectives

- use for loops to iteratively execute a block of code
- prematurely terminate loop execution using the break keyword
- short-circuit execution of the current loop iteration using the continue keyword
- effectively use nested loops
- use enhanced for loops, also known as for-each loops, to iterate over arrays and iterable objects in Java
- use while loops to control the number of times a block of code executes based on the value of a specific expression
- use multiple loop variables to control the number of iterations in a while loop
- identify common causes of infinite looping, such as incorrect use of the continue keyword
- use the do-while loop control structure to ensure that the body of a loop is always executed at least once, regardless of the value of the loop variable


Java SE 13: Understanding Classes & Objects in Java

Objectives

- recognize what classes and objects are
- recognize what member variables, static member variables, and member functions are
- define a class and instantiate an object of that class in Java
- print out the details of an object to the system console
- define and invoke constructors in classes to correctly initialize member variables
- define and invoke getter and setter methods to access private member variables of a class from outside that class


Java SE 13: Constructors & Other Methods

Objectives

- describe the initialization of member variables of class objects
- define and invoke a parameterized constructor to instantiate objects of a class
- define and invoke multiple constructors to instantiate objects of a class
- use the this keyword appropriately inside the constructor and other methods of a class
- define getter and setter methods to work with access protected member variables
- use the this keyword appropriately inside getter and setter methods of a class
- make use of method overloading in your programs


Java SE 13: More Operations on Member Variables & Methods

Objectives

- appropriately override the .toString method provided by the Object base class
- correctly chain constructors to achieve code reuse
- identify use cases for static member variables
- define and use static member variables
- recognize how static methods can be accessed by instances of a class
- invoke static and non-static methods of a class


Java SE 13: Semantics of Method Invocation & Nested Classes

Objectives

- use initialization blocks to appropriately initialize static and non-static member variables
- recognize how variables of primitive types are governed by pass-by-value semantics in Java
- recognize how variables of reference types are governed by pass-by-reference semantics in Java
- avoid pitfalls related to re-initialization and modification of objects (reference types) while invoking functions in Java
- correctly define static nested classes
- correctly initialize objects of static nested classes
- correctly define inner classes
- correctly initialize objects of inner classes


Java SE 13: Understanding Exception Handling in java

Objectives

- define an exception and recognize how it disrupts program execution
- describe and implement the catch-or-specify restriction
- recognize how the JRE searches for an exception handler for an exception
- enumerate the important built-in exception classes in Java
- recognize how exceptions such as NullPointerException, NumberFormatException, and IndexOutOfBoundsException occur in code
- identify differences between IO-related exceptions and other commonly encountered exceptions
- enumerate differences between checked exceptions, runtime exceptions, and errors


Java SE 13: Correctly Using Try-Catch-Finally Blocks

Objectives

- identify the situations when divide-by-zero exceptions arise
- apply the methods of the Exception base class to obtain exception information
- use multiple independent catch blocks with a single try block
- combine multiple independent catch blocks using the | operator
- combine multiple related catch blocks using the | operator
- employ correct exception-handling techniques for IO-intensive operations such as working with files
- use the finally keyword to ensure correct release of file handles and other system resources
- recognize the exact semantics and limitations of the finally block
- correctly place the finally block relative to the catch blocks


Java SE 13: Try-with-resources & Custom Exceptions

Objectives

- correctly use the throws clause while defining method signatures
- implement exception handling in a chain of methods that sequentially call each other
- ensure that resources are automatically closed, relying on language support, using the try-with-resources construct
- recognize the precise semantics of the try-with-resources construct in Java
- throw an object of a built-in exception type in order to respond to an unexpected situation in a program
- differentiate between the semantics of throwing runtime exceptions vs. throwing checked exceptions
- correctly invoke a method that throws an exception of a checked exception type
- correctly declare or handle exceptions in a chain of functions, each of which throws different types of exceptions
- create a custom exception object by extending the base class Exception, and throw an instance of this exception from your code
- instantiate and throw custom exception objects of both checked and unchecked exception types


Java SE 13: Byte & Character Streams in Java

Objectives

- recognize Java's class hierarchy for byte and character streams
- read a byte stream using the InputStream and FileInputStream classes
- apply the skip and available methods of FileInputStream objects and identify when the end of a byte stream has been reached
- write a byte stream using the OutputStream and FileOutputStream classes
- copy any file as a sequence of bytes using the FileInputStream and FileOutputStream classes
- read a character stream using the FileReader and FileWriter classes
- copy a character file as a sequence of characters using the FileReader and FileWriter classes


Java SE 13: Streams for Primitive Types, Objects, & Buffered IO

Objectives

- use the BufferedInputStream and BufferedOutputStream classes to efficiently interact with byte streams
- use the BufferedReader and BufferedWriter classes to efficiently interact with character streams
- use the DataOutputStream class to write primitive types and strings to byte streams
- use the DataInputStream class to read primitive types and strings from byte streams
- use the ObjectOutputStream class to write objects of any type to byte streams
- use the ObjectInputStream class to read objects of any type from byte streams
- implement the Serializable interface in user-defined classes so that they can be written to and read from using ObjectOutputStream and ObjectInputStream objects


Java SE 13: Working with Files & Directories

Objectives

- use the Path interface and the Paths.get methods to work with files and directories using the modern java.nio namespace
- use the methods of the Path interface to normalize, parse, and resolve paths
- create files using both the legacy java.io as well as the modern java.nio namespaces
- perform operations such as copying files, checking for their existence, and deleting files using the Files class in the java.nio namespace
- perform operations such as copying and creating directories using the Files class in the java.nio namespace
- get and set file attributes (including both platform-agnostic as well as POSIX attributes), and query file stores for total and free space
- create directories and iterate over their contents, including via the use of glob specifiers to filter directory contents


Apprentice Developer

Objectives

- In this lab, you will perform Apprentice Developer tasks such as working with arrays and static and non static methods, managing exceptions in Java, and working with FileWriter and FileReader. Then, test your skills by answering assessment questions after using Do-While Loop, working with inner classes and using try-with-resources.

Final Exam: Apprentice Developer

Objectives

- appropriately override the .toString method provided by the Object base class
- combine multiple independent catch blocks using the | operator
- copy a character file as a sequence of characters using the FileReader and FileReader classes
- correctly chain constructors to achieve code reuse
- correctly initialize objects of inner classes
- correctly initialize objects of static nested classes
- correctly invoke a method that throws an exception of a checked exception type
- correctly use the throws clause while defining method signatures
- create a custom exception object by extending the base class Exception, and throw an instance of this exception from your code
- create and use enums (enumerated data types) in Java
- create files using both the legacy java.io as well as the modern java.nio namespaces
- create String objects using different input arguments and forms of memory allocation
- create wrapper objects such as Integer, Float, etc. to encapsulate primitive (non-object) data types
- customize IntelliJ to incorporate your individual preferences
- define and invoke a parameterized constructor to instantiate objects of a class
- define and invoke constructors in classes to correctly initialize member variables
- define and invoke getter and setter methods to access private member variables of a class from outside that class
- define and invoke multiple constructors to instantiate objects of a class
- describe and use the -=, *=, %= and %= self-assignment operators
- differentiate between equality tests based on the == operator and .equals
- enumerate differences between checked exceptions, runtime exceptions, and errors
- enumerate the important built-in exception classes in Java
- identify differences between primitive and object types in Java
- identify the situations when divide-by-zero exceptions arise
- identify use-cases for the unary increment and decrement operators
- implement the Serializable interface in user-defined classes so that they can be written to and read from using ObjectOutputStream and ObjectInputStream object
- install Java on a Unix or Mac system
- install the Eclipse IDE on a Mac-based system
- invoke static and non-static methods of a class
- list the reasons for Java’s popularity
- make use of method overloading in your programs
- modify control flow in a program using the if construct
- perform conversions between variables of primitive types
- perform operations such as copying and creating directories using the Files class in the java.nio namespace
- perform operations such as copying files, checking for their existence, and deleting files using the Files class in the java.nio namespace
- prematurely terminate loop execution using the break keyword
- read a byte stream using the InputStream and FileInputStream classes
- recognize how exceptions such as NullPointerException, NumberFormatException, and IndexOutOfBoundsException occur in code
- recognize how variables of primitive types are governed by pass-by-value semantics in Java
- recognize how variables of reference types are governed by pass-by-reference semantics in Java
- recognize Java's class hierarchy for byte and character streams
- recognize the exact semantics and limitations of the finally block
- recognize the precise semantics of the try-with-resources construct in Java

- recognize what member variables, static member variables, and member functions are
- short-circuit execution of the current loop iteration using the continue keyword
- throw an object of a built-in exception type in order to respond to an unexpected situation in a program
- use enums (enumerated types) along with switch statements
- use for loops to iteratively execute a block of code
- use multiple independent catch blocks with a single try block
- use nested if-else conditions to deal with mutually exclusive conditions
- use strings in if-else constructs, relying on the .equals method and avoiding the == operator
- use string variables in switch statements
- use the DataOutputStream class to write primitive types and strings to byte streams
- use the do-while loop control structure to ensure that the body of a loop is always executed at least once, regardless of the value of the loop variable
- use the logical AND, logical OR, and logical negation operators
- use the ObjectOutputStream class to write objects of any type to byte streams
- use the StringBuffer object to perform string operations
- use while loops to control the number of times a block of code executes based on the value of a specific expression
- utilize both single-line as well as multi-line comments in Java
- work with variables of double, char, and String types in Java



Java OOP: Understanding Inheritance & Polymorphism in Java

Objectives

- identify advantages and applications of inheritance
- apply inheritance to model real-world entities
- identify advantages and applications of polymorphism
- recognize how the methods derived from java.lang.Object work
- recognize how all built-in Java classes possess the methods derived from java.lang.Object
- recognize how objects of derived classes will have independent copies of member variables even from the derived class
- recognize how in single inheritance, every object of the derived class is an object of the base class, but not every object of the base class is an object of the derived class
- recognize how to upcast and downcast and how downcasting can sometimes be dangerous
- recognize how you can create one base class with multiple derived classes


Java OOP: The Role of Constructors in Inheritance

Objectives

- identify the role of access modifiers and recognize the basics of constructors
- recognize when you add your own constructor, Java takes away the default no-argument constructor it had provided
- apply the two ways of initializing derived classes - implement an explicit no argument constructor in the base class or use the super keyword and have only parameterized constructors in both the base and derived classes
- recognize that a base class's constructor is invoked when a derived class object is instantiated
- identify what happens when there is a base and derived class that have many different constructors
- add a member variable into a derived class and ensure it is correctly initialized, even in addition to any invocations of the base class constructor


Java OOP: Multi-level Inheritance

Objectives

- recognize how you can access multiple derived classes by using a base class in multilevel inheritance
- use the instanceof operator to check that an object is an instance of every class in its inheritance hierarchy
- recognize how to apply multilevel inheritance when every object of the derived class is an object of the base class, but not every object of the base class is an object of the derived class
- recognize how you can upcast multiple levels up and how downcasting is very dangerous
- perform various casting operations
- construct two parallel base classes and derived classes for each of those base classes
- recognize situations where multiple inheritance is not allowed


Java OOP: Run-time & Compile-time Polymorphism

Objectives

- describe run-time and compile-time polymorphism
- recognize how run-time polymorphism works
- use the @Override annotation
- recognize how polymorphism works in the presence of a multilevel inheritance hierarchy
- recognize how easy it is to add a class into an inheritance hierarchy
- use compile-time polymorphism and method overriding
- use compile-time polymorphism, including type promotions
- recognize that compile-time polymorphism does not include type demotions
- use a combination of run-time and compile-time polymorphism


Java OOP: Understanding Overriding & Hiding in Inheritance

Objectives

- apply method overriding and recognize the concept of hiding
- use the super keyword inside any method invocation in the derived class
- use advanced forms of overriding
- combine method overriding with fairly complex inheritance hierarchies
- recognize that static methods don't support run-time polymorphism
- recognize how static methods are bound using compile-time rather than run-time binding
- override the method .toString, which is inherited from java.lang.Object
- describe how the .equals method and the == operator are related
- correctly override the .hashCode method and write the code to correctly override the .equals method, which is inherited from java.lang.Object
- completely and correct implement the .hashCode contract


Java OOP: The Semantics of the final & abstract Keywords

Objectives

- recognize how you can assign a member variable to be final and the implications of doing so
- define and use static final member variables
- instantiate final arrays
- create and use final classes and final methods
- recognize the implications of defining a class as abstract
- recognize how to extend abstract classes and the implications of doing so
- recognize that a class can only be instantiated if it has implemented all abstract methods


Java OOP: Access Modifiers for Regulating Access

Objectives

- describe properties of public, private, and protected access modifiers
- recognize how access works from one package into another
- use the protected access modifier
- use the protected keyword across packages
- recognize the semantics of using default access modifiers inside the same package
- recognize the semantics of using default access modifiers outside the current package


Java OOP: Interfaces & Anonymous Classes

Objectives

- use interfaces and recognized that all of their methods are abstract by default
- recognize how the instanceof operator works the same way with interfaces as with classes
- implement a derived class that can both extend a base class and implement an interface
- set up two inheritance hierarchies and combine these into a set of objects that implement both set of interfaces
- use interfaces that support default method implementations
- set up a class that implements some of the methods of an interface and then declares itself abstract
- create and define anonymous inner class objects
- access outer class variables from your anonymous inner class


Git & GitHub: Introduction

Objectives

- recall the purpose and fundamental features of version control systems
- contrast Git with other version control system and identify its main features
- describe the use cases and the process of branching and merging in Git
- identify the role of GitHub when working with Git repositories


Git & GitHub: Working with Git Repositories

Objectives

- configure your Git client with your own information
- create a local Git repository and commit code to it
- set up a remote Git repository on GitHub
- connect your local Git repository to a remote one on GitHub using SSH
- push commits from your local Git repository to a remote repo
- recognize the steps required to add new files to your Git repository
- make changes to existing files in your Git repo and push those modifications to the remote repo
- view the commit history of your repository
- recognize the effect of creating Git branches and recording commits on them
- remove files from your repository as well as your development workspace using Git
- identify the steps required to merge a feature branch to the master of a Git repository
- merge a feature branch to the master of your local Git repo and push the changes to the remote repo


Git & GitHub: Using GitHub for Source Code Management

Objectives

- recall the purpose of a pull request and the steps involved in creating and approving them for a Git merge operation
- add a teammate to collaborate with you on GitHub
- recognize the steps involved for a merge when a collaborator raises a pull request
- identify the situations which can lead to a merge conflict and how these could be resolved
- recognize the purpose of a Git pull operation to keep your local repository in sync with the remote repo
- view statistics related to your GitHub repo using the insights feature
- label specific commits in your repository's history using tags
- manage the tags associated with your repo by removing them from your local as well as remote repositories
- document and track issues related to your application using GitHub issues
- group similar GitHub issues together using milestones
- track the progress of issues created for your repository using project boards
- document information about your GitHub repo using the Wiki feature
- enable organized collaboration with teammates using GitHub Organizations
- create a local copy of a remote GitHub repo by generating a clone

Final Exam: Senior Developer

Objectives

- access outer class variables from your anonymous inner class
- add a teammate to collaborate with you on GitHub
- apply inheritance to model real-world entities
- apply method overriding and recognize the concept of hiding
- apply the two ways of initializing derived classes - implement an explicit no argument constructor in the base class or use the super keyword and have only parameterized constructors in both the base and derived classes
- configure your Git client with your own information
- connect your local Git repository to a remote one on GitHub using SSH
- construct two parallel base classes and derived classes for each of those base classes
- contrast Git with other version control system and identify its main features
- correctly override the .hashCode method and write the code to correctly override the .equals method, which is inherited from java.lang.Object
- create a local copy of a remote GitHub repo by generating a clone
- create a local Git repository and commit code to it
- create and define anonymous inner class objects
- create and use final classes and final methods
- define and use static final member variables
- describe how the .equals method and the == operator are related
- describe properties of public, private, and protected access modifiers
- describe run-time and compile-time polymorphism
- describe the use cases and the process of branching and merging in Git
- document and track issues related to your application using GitHub Issues
- document information about your GitHub repo using the Wiki feature
- identify advantages and applications of inheritance
- identify the role of access modifiers and recognize the basics of constructors
- identify the situations which can lead to a merge conflict and how these could be resolved
- identify what happens when there is a base and derived class that have many different constructors
- implement a derived class that can both extend a base class and implement an interface

- label specific commits in your repository's history using tags
- manage the tags associated with your repo by removing them from your local as well as remote repositories
- merge a feature branch to the master of your local Git repo and push the changes to the remote repo
- override the method .toString, which is inherited from java.lang.Object
- recall the purpose of a pull request and the steps involved in creating and approving them for a Git merge operation
- recognize how access works from one package into another
- recognize how easy it is to add a class into an inheritance hierarchy
- recognize how in single inheritance, every object of the derived class is an object of the base class, but not every object of the base class is an object of the derived class
- recognize how objects of derived classes will have independent copies of member variables even from the derived class
- recognize how run-time polymorphism works
- recognize how static methods are bound using compile-time rather than run-time binding
- recognize how the methods derived from java.lang.Object work
- recognize how to extend abstract classes and the implications of doing so
- recognize how when you add your own constructor, Java takes away the default no-argument constructor it had provided
- recognize how you can create one base class with multiple derived classes
- recognize how you can upcast multiple levels up and how downcasting is very dangerous
- recognize situations where multiple inheritance is not allowed
- recognize situations where multiple inheritance is not allowed
- recognize that a base class's constructor is invoked when a derived class object is instantiated
- recognize that compile-time polymorphism does not include type demotions
- recognize the semantics of using default access modifiers inside the same package
- recognize the steps required to add new files to your Git repository
- remove files from your repository as well as your development workspace using Git
- set up a remote Git repository on GitHub
- use advanced forms of overriding
- use compile-time polymorphism and method overriding
- use compile-time polymorphism, including type promotions
- use interfaces and recognized that all of their methods are abstract by default
- use interfaces that support default method implementations
- use the instanceof operator to check that an object is an instance of every class in its inheritance hierarchy
- use the protected keyword across packages
- use the super keyword inside any method invocation in the derived class
- view statistics related to your GitHub repo using the Insights feature
- view the commit history of your repository

Senior Developer

Objectives

- In this lab, you will perform Senior Developer tasks such as working with derived methods, using the Super Keyword, implementing polymorphism and working with final classes and methods. Then, test your skills by answering assessment questions after implementing cross-package access of protected members, pushing from a local to remote repo, merging branches and working with issue and milestone in Github.



Advanced Features in Java: Getting Started with Java Collections

Objectives

- construct arrays to contain objects of various types
- describe the two limitations of an array
- describes the is-a relationship of an ArrayList with Collection, Iterable, and List
- describe how an ArrayList is ordered and allows duplicates
- recognize individual the methods from the collection interface
- recognize how the Iterable and Iterator interfaces can be accessed using loops to avoid run-time errors from going over the limits of a list
- recognize how the .get method can be used to access individual elements of a list


Advanced Features in Java: Working with Lists in Java

Objectives

- describe how lists can be instantiated with type parameters to achieve type safety
- describe the ArrayList, LinkedList, and Vector classes and how these are all instances of the list, collection, and iterable interfaces
- describe important methods of the List interface and how the .addall, .removeall, and .retainall methods work
- describe how the .get, .set, .add, .indexof, and .lastindexof methods work
- describe how you can iterate over lists using ListIterator
- describe how the .hasnext, .next, .previous, and .hasprevious methods can be used to iterate over lists in different ways
- describe when you can use the .sublist method to get a part of a list


Advanced Features in Java: List Algorithms & Implementations

Objectives

- differentiate between ArrayLists and LinkedLists and the use cases where they are suitable
- describe the LinkedList and Vector implementations of the List interface
- describe how you can create a custom comparator
- describe how you can use the .copy function to take values from one list and put them in another
- describe how overriding the .equals method will affect duplicates
- describe how the overridden version of .equals can be used to control list equality operations


Advanced Features in Java: Working with Sets In Java

Objectives

- create a HashSet object and invoke multiple methods on it, and also correctly override the .hashCode and .equals method of the contained class
- describe how the HashSet behaves with duplicates and how overridden versions of the .equals and the .hashCode methods influence this behavior
- recognize the different types of sets and how they all extend the Set, Collection, and Iterable interfaces
- differentiate between methods used to compute set union, difference, intersection, and equality operations
- analyze the performance characteristics of LinkedHashSets, TreeSets, and HashSets
- recognize how and why EnumSets should be used to represent categorical data
- describe how different implementations of the Set interface differ in their notions of set order
- create a TreeSet and sort it using various custom comparators
- create a TreeSet that stores custom objects without also creating a custom comparator


Advanced Features in Java: Working with Maps in Java

Objectives

- describe how maps work and how they are not quite collections but are an important part of the Java Collections API
- demonstrate that custom objects can be keys or values in maps, but that in order to ensure that there are no duplicates you have to override the .hashCode and .equals methods of those contained objects correctly
- recognize that maps are not collections with a series of instanceof checks and a series of comparisons
- recognize how maps work and that they are very similar to their set counterparts
- create a TreeMap and use a custom comparator


Advanced Features in Java: Using the Java Stream API with Collections

Objectives

- recognize what streams are and some of the methods you can invoke on them
- describe how to test whether all elements in a stream satisfy a specific condition
- demonstrate that predicates can be used to transform one stream object into another
- associate streams containing integer, long, float, and double objects to specific stream types
- apply terminal operations on predicates
- assess how to test streams with the .collect function in order to return a collection
- define a custom collector and use it with a stream predicate


Advanced Features in Java: Using Built-in Annotations

Objectives

- define a method, override it, and then mark that overridden version with the @Override annotation
- apply the @Override annotation to detect typographical errors in method names at compile-time rather than at run-time
- recognize different common overrides of methods in java.lang.Object such as the .toString, .equals, and .hashCode methods
- identify correctly and incorrectly overridden implementations of the .equals method
- recognize how the @Deprecated annotation can be used to flag the instantiation of objects of deprecated classes at compile-time
- analyze how the @Deprecated annotation works on elements other than classes
- recognize how the forRemoval named element from the @Deprecated annotation works
- enumerate different scenarios in which the @SuppressWarnings annotation can be applied
- recognize the different warnings that can be eliminated using the @SuppressWarnings annotation, and why this annotation ought to be used sparingly, if at all


Advanced Features in Java: Using Custom Annotations

Objectives

- describe how varargs are defined and used
- demonstrate that the @SafeVarargs annotation is purely indicative and does not perform any run-time or compile-time checks
- recognize how functional interfaces must contain exactly one abstract method and what qualifies as an abstract method in this context
- recognize exactly what counts as a functional interface and what does not
- describe the use of custom annotations and recognize that these are usually purely intended to be understood by programmers and that the standard Java compiler does not understand them
- change the target policy to control exactly what code elements an annotation can be applied to
- ensure that all the elements marked with an annotation satisfy certain conditions
- demonstrate the use of named elements in an annotation
- apply default values to an element in annotation and also experiment with unnamed elements
- identify how to use an annotation with a type parameter
- recognize how annotations can be used with target policy Type_Use


Advanced Features in Java: Using Generic Type Parameters

Objectives

- recognize how using type parameters can ensure both type safety and code reuse
- recognize different aspects of specifying and using type parameters
- describe the raw type underlying a generic type, and how if no type parameter is specified, Java makes the object of type java.lang.Object the type parameter
- recognize that the names of type parameters don’t matter so long as certain variable naming rules are not violated
- iterate over objects that have multiple type parameters
- recognize how type parameters can be applied not only to classes, but also to specific methods inside classes
- describe the use of type parameters within functions
- describe the use of more than one type parameter
- incorporate constraints, known as bounds, on type parameters
- differentiate between bounded and unbounded type parameters


Advanced Features in Java: Wildcards and Type Capture

Objectives

- recognize how you can constrain type parameters to extend the Comparable and Serializable interfaces
- demonstrate various relationships between base and derived classes in the presence of type parameters
- recognize how type parameters may exist only in the base class, or only in the derived class, or in both
- recognize use-cases where a base class specifies one type parameter and the derived class specifies multiple type parameters
- describe how wildcards mitigate some limitations of invoking methods with generic input arguments
- use upper bounded wildcards to specify that a type parameter must be a sub-class of a certain type
- recognize how unbounded wildcards can be used where no information about inheritance relationships is available
- use lower bounded wildcards to specify that a type parameter must be a super-class of a certain type
- recognize scenarios where capture errors might occur and some potential fixes for such capture errors
- analyze compiled bytecode to see how Java uses type erasure to ensure type safety and prevent code bloat


Data Structures & Algorithms in Java: Introduction

Objectives

- recall the importance of using data structures and algorithms
- recognize the differences between data structures and abstract data types
- measure performance based on time, space, and network bandwidth usage
- apply complexity to measure performance
- use the big-O notation as a measure of complexity
- set up a new Java project
- analyze algorithms with constant time complexity
- analyze algorithms with linear time complexity
- recognize more algorithms that have linear time complexity
- time operations to see how the running time changes based on input size
- recognize simple examples from the real world that exhibit linear time complexity
- analyze algorithms with quadratic time complexity
- analyze algorithms with cubic time complexity
- analyze algorithms with logarithmic time complexity


Data Structures & Algorithms in Java: Working with Singly Linked Lists

Objectives

- recognize the basic structure of the linked list
- insert a node into a linked list
- search for a node with specific data in a linked list
- delete a node from a linked list
- count the number of nodes in a linked list
- set up the basic structure of a singly linked list node
- insert a new node at the head and count the number of nodes in a singly linked list
- insert a new node at the tail of the linked list
- insert a new node at a specified index
- implement the pop and contains operations on a linked list
- delete a node from a linked list and rewire the list
- find all nodes less than a certain value in a linked list
- recall the differences between arrays and linked lists


Data Structures & Algorithms in Java: Doubly & Circular Linked Lists

Objectives

- insert a new node at the head and tail of a doubly linked list
- insert a new node at any index position in a doubly linked list
- delete a node from a doubly linked list
- traverse a doubly linked list from the last element to the first
- insert a new node in a circular linked list
- count nodes and delete nodes in a circular linked list


Data Structures & Algorithms in Java: Working with Stacks

Objectives

- recall the basic characteristics of the stack data structure
- perform simple operations on a stack implemented using arrays
- push new elements on to a stack implemented using arrays
- pop elements from a stack implemented using arrays
- peek into a stack implemented using arrays
- push new elements on to a stack implemented using linked lists
- pop elements from and peek into a stack implemented using linked lists
- set up the basic interface for the Command object
- implement undo using stacks and the Command object
- check for well formed arithmetic expressions using stacks
- find the minimum element in constant time in a stack


Data Structures & Algorithms in Java: Working with Queues

Objectives

- recall the basic characteristics of a queue
- perform simple operations using Is Full, Is Empty, and Size on a queue implemented using arrays
- enqueue elements in a queue implemented using arrays
- dequeue elements in a queue implemented using arrays
- recall why enqueue is an O(N) operation in a queue implemented using arrays
- enqueue elements in a queue implemented as a circular queue
- dequeue elements in a queue implemented as a circular queue
- enqueue elements in a queue implemented using linked lists
- dequeue elements and peek into a queue implemented using linked lists
- implement a double-ended queue
- build a queue using two stacks
- use the built-in classes in Java for queues and stacks
- use the priority queue and specify priorities using a comparator

Final Exam: Lead Developer

Objectives

- analyze algorithms with constant time complexity
- analyze algorithms with cubic time complexity
- analyze algorithms with linear time complexity
- analyze algorithms with logarithmic time complexity
- analyze algorithms with quadratic time complexity
- analyze compiled bytecode to see how Java uses type erasure to ensure type safety and prevent code bloat
- apply default values to an element in annotation and also experiment with unnamed elements
- apply terminal operations on predicates
- apply the @Override annotation to detect typographical errors in method names at compile-time rather than at run-time
- change the target policy to control exactly what code elements an annotation can be applied to
- count nodes and delete nodes in a circular linked list
- create a HashSet object and invoke multiple methods on it, and also correctly override the .hashCode and .equals method of the contained class
- create a TreeSet and sort it using various custom comparators
- define a method, override it, and then mark that overridden version with the @Override annotation
- delete a node from a linked list and rewire the list
- demonstrate that custom objects can be keys or values in maps, but that in order to ensure that there are no duplicates you have to override the .hashCode and .equals methods of those contained objects correctly
- demonstrate that predicates can be used to transform one stream object into another
- demonstrate that the @SafeVarargs annotation is purely indicative and does not perform any run time or compile checks
- demonstrate various relationships between base and derived classes in the presence of type parameters
- dequeue elements in a queue implemented as a circular queue
- dequeue elements in a queue implemented using arrays
- describe how different implementations of the Set interface differ in their notions of set order
- describe how the .get, .set, .add, .indexof, and .lastindexof methods work
- describe how the overridden version of .equals can be used to control list equality operations
- describe how varargs are defined and used
- describe how you can create a custom comparator
- describe how you can iterate over lists using ListIterator
- describe how you can use the .copy function to take values from one list and put them in another
- describe important methods of the List interface and how the .addall, .removeall, and .retainall methods work
- describes the is-a relationship of an ArrayList with Collection, Iterable, and List
- describe the two limitations of an array
- describe when you can use the .sublist method to get a part of a list
- differentiate between ArrayLists and LinkedLists and the use cases where they are suitable
- differentiate between bounded and unbounded type parameters
- differentiate between methods used to compute set union, difference, intersection, and equality operations
- enqueue elements in a queue implemented as a circular queue
- enqueue elements in a queue implemented using arrays

- enumerate different scenarios in which the @SuppressWarnings annotation can be applied
- incorporate constraints, known as bounds, on type parameters
- insert a new node at the head, count the number of nodes in a linked list
- insert a new node at the tail of the linked list
- pop elements from and peek into a stack implemented using linked lists
- pop elements from a stack implemented using arrays
- push new elements on to a stack implemented using linked list
- push new elements on to the stack implemented using arrays
- recognize different aspects of specifying and using type parameters
- recognize exactly what counts as a functional interface and what does not
- recognize how maps work and that they are very similar to their set counterparts
- recognize how the @Deprecated annotation can be used to flag the instantiation of objects of deprecated classes at compile-time
- recognize how the Iterable and Iterator interfaces can be accessed using loops to avoid run-time errors from going over the limits of a list
- recognize how type parameters can be applied not only to classes, but also to specific methods inside classes
- recognize how type parameters may exist only in the base class, or only in the derived class, or in both
- recognize how using type parameters can ensure both type safety and code reuse
- recognize the different types of sets and how they all extend the Set, Collection, and Iterable interfaces
- recognize use-cases where a base class specifies one type parameter and the derived class specifies multiple type parameters
- recognize what streams are and some of the methods you can invoke on them
- search for a node with specific data in a linked list
- traverse a doubly linked list from the last element to the first
- use the built-in classes in Java for queues and stacks
- use upper bounded wildcards to specify that a type parameter must be a sub-class of a certain type

Lead Developer

Objectives

- In this lab, you will perform Lead Developer tasks such as working with ArrayList, using Comparator with TreeSet, working with Maps and Streams. Then, test your skills by answering assessment questions after implementing custom annotations, using TypeParameters, working with LinkedList and managing queue.



Mobile App Development: An Introduction Android Development

Objectives

- describe the Android operating system, its origins, and history
- list the IDEs and languages that can be used for Android application development
- identify the different files and directories that make up a project in Android Studio
- recognize the purpose of Android activities and the files needed to define them
- recall the different elements that can be used to define an activity and how they can be grouped together
- identify the features and quirks of the Kotlin programming language
- recognize how the Kotlin language handles null values and late variable initializations, and how the language applies to the Android OS
- describe the various steps involved in building an Android app, from creating build scripts to testing on a virtual device emulator


Mobile App Development: Building a Basic Android App

Objectives

- download and install the official IDE for Android app development - Android Studio
- create and setup an Android project in the Android Studio IDE
- recognize the different files and directories in an Android project and the purpose each of them serves
- define the appearance of one screen of an Android app by using the visual UI editor in Android Studio
- install and configure a virtual device on which to deploy and test your Android app
- launch an Android app on a virtual device emulator and interact with it
- modify the appearance of one screen in your Android app by editing the XML layout file
- add and configure a field in your Android app in order to receive text input from the end user
- define the structure and positioning of different elements on one screen of your Android app using a ConstraintLayout
- eliminate hardcoded data in your layout definitions by creating references to strings in a separate resource file
- create an Activity layout that displays a value supplied by a user in a different Activity in your app
- pass a message from one Activity to another in an Android app
- deploy and test the behavior of your Android app on a virtual device emulator


Mobile App Development: Defining the UI for an Android App

Objectives

- recognize requirements for an Android application that performs currency conversion using live foreign exchange data
- use the resources files for an Android application to define strings and colors that will be referenced from other files
- add an image to the resources of an Android app
- define the header elements for a navigation menu in an Android app
- configure a menu with several items in an Android app
- combine a header and menu in order to design a navigation pane for an Android app
- design a screen containing an image, drop-downs, buttons, and numeric input fields
- create a splash screen rendering text for an Android app
- define a screen that displays a vertically-scrollable list of information along with other details
- configure a help page with dynamically populated answers


Mobile App Development: Coding the Behavior of an Android App

Objectives

- code the logic for an app's splash screen that includes directing a user to the main screen of the app
- define a property to hold live foreign exchange data that is accessible to all Activities in your Android app
- configure the Gradle build script of an Android project with the dependencies of your app
- create a base Activity class that others can inherit in order to implement a common navigation pane
- define the actions to be performed for the selection of each individual item in a navigation pane
- initialize the main activity of your Android app and implement inheritance in the Kotlin language
- use Kotlin libraries in order to issue HTTP requests and parse JSON data
- write the code to load a series of values into a ListView widget
- configure the launcher activity and define permissions for your Android application using its manifest file
- launch your app in a virtual device emulator and verify its behavior
- create an APK file that can be used to load your app onto a physical Android device


Mobile App Development: Authentication in an Android App

Objectives

- define the layout for the login screen of an Android app
- design a registration screen for your Android app using text fields for names, dates, e-mails, and passwords
- modify the navigation pane of an app to include a logout item
- enable the Firebase authentication service for your Android app
- integrate your app with the real-time database service of Firebase that is hosted on the cloud
- code the user registration Activity of your app to perform some simple data validation
- define the logic for an Activity that stores data input by a user in a Firebase real-time database
- write the code for an Activity that integrates with the Firebase authentication service to confirm a user's identity
- enable the logging out of a user from an app through the navigation menu
- set up Firebase to authenticate your app's users and to store their data in a database
- deploy your app on a virtual device and verify the connectivity with the Firebase database
- confirm that registered users of your app are able to sign in with the credentials they registered with


Mobile App Development: Testing an Android App

Objectives

- run a unit test using the Android Studio UI and analyze the results
- define a simple unit test that does not involve the use of UI elements of your Android app
- modularize your unit tests by separating the common setup tasks into a separate function
- execute an instrumented test using Android Studio
- code the startup tasks for an instrumented test on one of the Activities in your app
- define a functional test for your Android app involving interactions with various UI elements
- write functional tests for your Android app involving interactions with spinners, text fields, and buttons
- automate the testing of various UI elements in your app by codifying the navigation to the different Activities
- execute a functional test against the main Activity of your app and verify the results
- run multiple tests that involve common setup tasks and verify the behavior of the app in an emulator
- verify the functionality of validators in your app by running tests against an Activity that uses them
- record your actions on an emulator in order to simplify the creation of UI tests for your Android app
- apply final touches to the auto-generated code from an Espresso recording and execute the tests
- run all the tests defined for your Android app from the command line

Final Exam: Journeyman Developer

Objectives

- add an image to the resources of an Android app
- Apply final touches to the auto-generated code from an Espresso recording and execute the tests
- Code the logic for an app's splash screen that includes directing a user to the main screen of the app
- Code the startup tasks for an instrumented test on one of the Activities in your app
- Code the user registration Activity of your app to perform some simple data validation
- combine a header and menu in order to design a navigation pane for an Android app
- Configure a menu with several items in an Android app
- configure the Gradle build script of an Android project with the dependencies of your app
- Configure the launcher activity and define permissions for your Android application using its manifest file
- Confirm that registered users of your app are able to sign in with the credentials they registered with
- create a base Activity class that others can inherit in order to implement a common navigation pane
- create an Activity layout that displays a value supplied by a user in a different Activity in your app
- create an APK file that can be used to load your app onto a physical Android device
- create and setup an Android project in the Android Studio IDE
- create a splash screen rendering text for an Android app
- Define a functional test for your Android app involving interactions with various UI elements

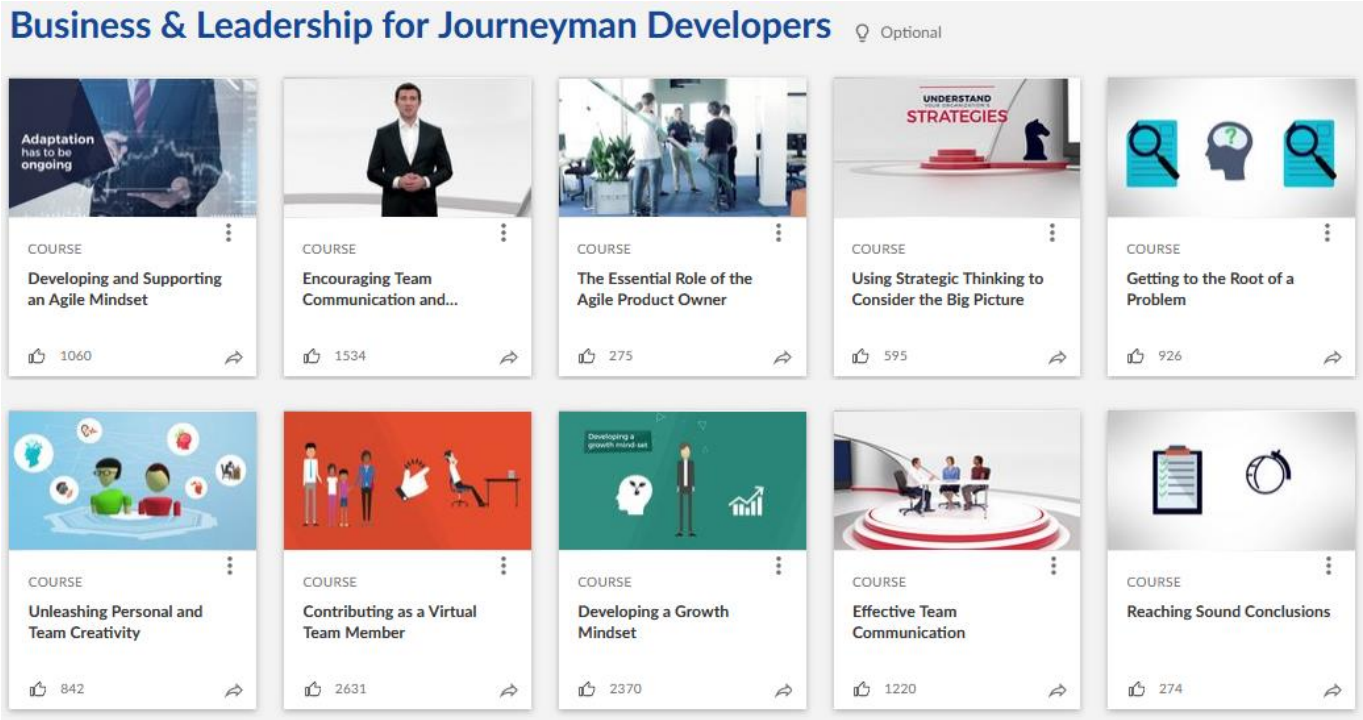
- define a property to hold live foreign exchange data that is accessible to all Activities in your Android app
- Define a screen that displays a vertically-scrollable list of information along with other details
- Define a simple unit test that does not involve the use of UI elements of your Android app
- Define the actions to be performed for the selection of each individual item in a navigation pane
- define the appearance of one screen of an Android app by using the visual UI editor in Android Studio
- define the header elements for a navigation menu in an Android app
- Define the layout for the login screen of an Android app
- Define the logic for an Activity that stores data input by a user in a Firebase real-time database
- define the structure and positioning of different elements on one screen of your Android app using a ConstraintLayout
- Deploy your app on a virtual device and verify the connectivity with the Firebase database
- describe the Android operating system, its origins, and history
- Design a registration screen for your Android app using text fields for names, dates, e-mails, and passwords
- Design a screen containing an image, drop-downs, buttons, and numeric input fields
- download and install the official IDE for Android app development - Android Studio
- eliminate hardcoded data in your layout definitions by creating references to strings in a separate resource file
- Enable the Firebase authentication service for your Android app
- Enable the logging out of a user from an app through the navigation menu
- Execute a functional test against the main Activity of your app and verify the results
- Execute an instrumented test using Android Studio
- identify the different files and directories that make up a project in Android Studio
- identify the features and quirks of the Kotlin programming language
- initialize the main activity of your Android app and implement inheritance in the Kotlin language
- install and configure a virtual device on which to deploy and test your Android app
- Integrate your app with the real-time database service of Firebase that is hosted on the cloud
- Launch your app in a virtual device emulator and verify its behavior
- list the IDEs and languages that can be used for Android application development
- modify the appearance of one screen in your Android app by editing the XML layout file
- modify the layout of your Android app
- Modularize your unit tests by separating the common setup tasks into a separate function
- pass a message from one Activity to another in an Android app
- recognize how the Kotlin language handles null values and late variable initializations, and how the language applies to the Android OS
- Recognize requirements for an Android application that performs currency conversion using live foreign exchange data
- recognize the different files and directories in an Android project and the purpose each of them serves
- recognize the purpose of Android activities and the files needed to define them
- Record your actions on an emulator in order to simplify the creation of UI tests for your Android app
- Run all the tests defined for your Android app from the command line
- Run a unit test using the Android Studio UI and analyze the results

- Run multiple tests that involve common setup tasks and verify the behavior of the app in an emulator
- Set up Firebase to authenticate your app's users and to store their data in a database
- Use Kotlin libraries in order to issue HTTP requests and parse JSON data
- use the resources files for an Android application to define strings and colors that will be referenced from other files
- Verify the functionality of validators in your app by running tests against an Activity that uses them
- Write functional tests for your Android app involving interactions with spinners, text fields, and buttons
- Write the code for an Activity that integrates with the Firebase authentication service to confirm a user's identity


Journeyman Developer

Objectives

- In this lab, you will perform Journeyman Developer tasks such as configuring Android Project, managing Android Virtual Device, working with Layout, and linking Intent. Then, test your skills by answering assessment questions after working with NavigationUI, managing splash screen and user interface, and using Espresso.



# Bookshelf

Optional


BOOK

**Beginning Git and GitHub: A Comprehensive Guide to...**

14


BOOK

**Java in Easy Steps, 7th Edition**

9


BOOK

**Java 13 Revealed: For Early Adoption and Migration,...**

4


BOOK

**The Java Module System**

3


BOOK

**Functional Interfaces in Java: Fundamentals and Examples**

10


BOOK

**Let us Java, Fourth Edition**


BOOK

**Java: A Complete Practical Solution**


BOOK

**Java Program Design: Principles, Polymorphism,...**


BOOK

**Java Design Patterns: A Hands-On Experience with...**


BOOK

**Java: The Complete Reference, Eleventh Edition**


BOOK

**Java: A Beginner's Guide, Eighth Edition**

22


BOOK

**Modern Java in Action: Lambdas, Streams, Reactive...**

27

BOOK

**Learn Git in a Month of Lunches**

6


BOOK

**Git for Humans**


BOOK

**Professional Git**

17


BOOK

**Pro Git, Second Edition**

58


BOOK

**Java For Artists: The Art, Philosophy, and Science of...**

4


BOOK

**Android Studio IDE Quick Reference: A Pocket Guide ...**

3


BOOK

**Learn Android Studio 3 with Kotlin: Efficient Android Ap...**

5


BOOK

**Learn Android Studio 3: Efficient Android App...**

4


BOOK

**Beginning Android Programming with Android...**

14


BOOK

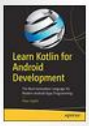
**Expert Android Studio**

2


BOOK

**Learn Android Studio: Build Android Apps Quickly and...**

9




BOOK

**Learn Kotlin for Android Development: The Next...**

12


BOOK

**Java Programming for Android Developers for...**

5

## Optional Resources

Optional


LAB

**Apprentice Developer to Journeyman Developer...**

Java SE 11

3


LAB

**App Development with Java Sandbox**

Java SE 11

12

## Productivity Tools for Journeyman Developers

Optional

COURSE

**Signing in & Setting up a Team**

21

COURSE

**Using the Conversation Tools**

20

COURSE

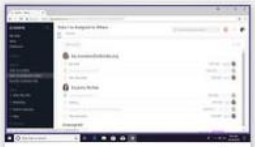
**Creating & Managing Projects**

17

COURSE

**Finding & Sharing Items**

10



COURSE

**Running Reports & Configuring Projects**

11


COURSE

**Signing In & Setting Up**

28

COURSE

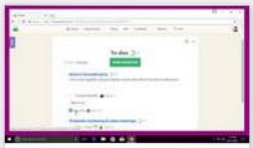
**Using the Team Communication Tools**

82

COURSE

**Setting Up & Tracking Projects**

25



COURSE

**Managing your Project Tasks & Assets**

20

COURSE

**Using the Calendar Tools**

22

COURSE

**Using Basecamp for iOS**

21


COURSE

**Signing in & Navigating within Spaces**

40

COURSE

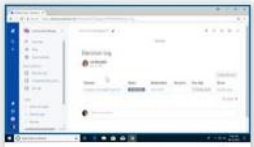
**Setting Up & Managing Spaces**

33


COURSE

**Working with Space**

27



COURSE

**Working with Team Members**

75

COURSE

**Configuring Spaces**

20

COURSE

**Sign-in & Setup**

18

COURSE

**Communication Tools**

23

COURSE

**Working with Groups**

14

COURSE

**Creating, Finding, & Sharing Information**

12

# Productivity Tools for Journeyman Developers (Continued)

COURSE

**Configuring Convo**

6

COURSE

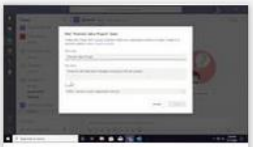
**The Convo iOS App**

13

COURSE

**Getting to know the application**

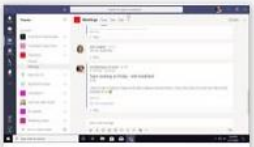
1009



COURSE

**Using Teams & Channels**

746



COURSE

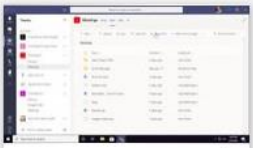
**Communicating via the App**

729

COURSE

**Formatting, Illustrating & Reacting to Messages**

542



COURSE

**Creating, Finding & Organizing Files**

536

COURSE

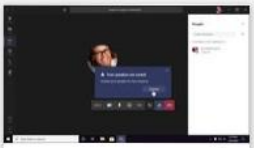
**Working with Apps, Tabs & Wiki**

465

COURSE

**Making calls, Organizing Contacts & Using Voicemail**

453



COURSE

**Creating, Joining & Managing Meetings**

461


COURSE

**Sign-in & Setup**

65

COURSE

**Creating Teams & Boards**

43

COURSE

**Managing Cards**

22

COURSE

**Finding & Sharing Information**

20

COURSE

**Setting Up**

61

COURSE

**Posting & Reacting to Status Updates**

46

COURSE

**Using Groups**

14

COURSE

**Collaborating & Communicating**

81

COURSE

**Configuring Networks**

25

COURSE

**Creating & Setting Up Projects in Jira Cloud**

164

COURSE

**Configuring & Managing Boards in Jira Cloud**

105

COURSE

**Planning & Working on a Software Project in Jira...**

85

COURSE

**Reporting in Jira Software**

82

**FOLLOW US ON:**


www.skilltech.pl

**email:** biuro@skilltech.pl

**tel. +48 22 44 88 827**


SkillTech
Technology hired for excellence