**JAVA Exercises**

Contents

[Instructions 2](#_Toc416004289)

[BASICS 3](#_Toc416004290)

[ADVANCED CONCEPTS 4](#_Toc416004291)

[Exercises on Basic concepts 5](#_Toc416004292)

[Language basics 5](#_Toc416004293)

[Classes, Objects 5](#_Toc416004294)

[Inheritance 6](#_Toc416004295)

[Exception Handling 7](#_Toc416004296)

[Enumerations 8](#_Toc416004297)

[Exercises on Advanced concepts 9](#_Toc416004298)

[Generics 9](#_Toc416004299)

[Multi-Threading 9](#_Toc416004300)

[Collections 10](#_Toc416004301)

[I/O 11](#_Toc416004302)

[Regular expressions 11](#_Toc416004303)

# Instructions

(1) This is training by self-learning program. There will not be any class-room sessions.

(2) Read the material provided in the Useful Links tab.

(3) Set up the development environment.

(4) Solve the exercises. You need to solve any **10 Simple, any 8 Medium and any 2 Complex exercises (spread uniformly across all sections)** to be able to get the training completion certificate.

(5) Do the JIRA updates as explained below:

(a) A Kanban board will be created for this training. A User Story will be created for a group of 10 to 12 trainees. A Sub-Task with your name as the Name will be created for your User Story.

(b) Sanketini will provide the details of the Kanban board and your User Story & Sub-Task. Access the Kanban board to make updates and to check the status of progress of your training.

(c) **Assign** your respective Sub-Task **to yourself** and move your Sub-Task to **In Progress** status when you start working on the exercises.

(d) After you solve each exercise, post a comment to the Sub-Task stating the ID of the exercise (S1, S3, M2, M5, C2 etc) and attach the code snippet(s) to the Sub-Task.

(e) When all the required exercises are complete, review if you have posted all the comments properly and attached the necessary code snippets. If the updates are proper, move the status of the Sub-Task to **Resolved**.

(6) When the task is moved to Resolved, Sanketini will check if the required-number of exercises are solved. If the requirements are met, the Sub-Task will be moved to **Verified** state and the training completion certificate will be initiated.

(7) When the certificate is ready, same will be handed over to your manager and the Sub-Task will be moved to **Closed** status.

# BASICS

1. OOPS concepts
   1. Classes, objects, methods.
   2. Abstraction
   3. Encapsulation
   4. Inheritance
   5. Polymorphism
2. JAVA language basics
   1. Executing a small java program
   2. Variables, Data types, Arrays, Operators
   3. Control Statements [if, switch, while, do-while, for]
   4. String manipulations
3. Classes, Objects
   1. Creating a class, instances…
   2. constructors
   3. Methods, passing primitives values and object references.
   4. Recursion
   5. constructors
   6. this keyword
   7. Garbage collection
   8. Access control [private, protected, public]
   9. Static
   10. packages
4. Inheritance
   1. Basics of inheritance
   2. super keyword
   3. Method overriding [Dynamic method dispatch].
   4. final keyword [with constants, final methods and final classes]
   5. java.lang.Object
   6. Abstract classes and interfaces
5. Exception handling
   1. Exception handling concepts [try/catch/finally blocks, throw, throws]
   2. Exception types [Throwable, Error, Exception (Runtime exception)]
   3. Creating custom exceptions.
   4. Chaining exceptions
   5. Try with resources [JAVA SE 7]
6. Enumerations

# ADVANCED CONCEPTS

1. Generics
2. Multi-Threading
   1. Basic threading concepts
      1. Extending Thread
      2. Implementing Runnable
      3. sleep, wait, notify, join methods
   2. Executors and Thread Pools
      1. Cached thread pool
      2. Fixed thread pool
      3. Scheduled thread pool
   3. Synchronization
3. API
   1. java.lang
   2. java.util [Without collections]
   3. java.io
4. Collections
   1. List
   2. Set
   3. Map
5. Regular Expressions

# Exercises on Basic concepts

Target audience: Entry level

## Language basics

Write java programs to

1. Perform arithmetic operations using double. [S1]
2. Compute SI, CI [S2]
3. Swapping of two numbers. [S3]
4. Demonstrate type casting. [S4]
5. Check entered number is even or not. [S5]
6. Check whether the given character is vowel or not using switch. [S6]
7. Greatest of three numbers. [S7]
8. Check entered number is prime or composite. [S8]
9. Prime number and palindrome generation. [S9]
10. Compute factorial. [S10]
11. Fibonacci sequence printing.[S11]
12. Declare arrays and initialize them. [S12]
13. Print an array in reverse direction.[S13]
14. Merge two arrays alternatively. Arrays lengths may not be same. [S14]
15. Matrix addition, subtraction, transpose.[S15]
16. Matrix multiplication.[M1]
17. Pascal triangle computation and printing the same.[M2]
18. Demonstrate strings. (Use most of java.lang.String methods) [S16]
19. Reverse of a string.[S17]
20. Palindrome check. [S18]
21. Use switch with strings (JAVA SE 7) [S19]
22. Demonstrate StringBuffer and StringBuilder [S20]
23. Conversion from primitives to String and vice versa. [S21]

## Classes, Objects

1. Create a class to represent a Dog, instantiate it and call the methods in it. [S22]  
   Create a class called Dog with an overloaded bark( ) method. This method should be overloaded based on various primitive data types, and print different types of barking, howling, etc., depending on which overloaded version is called. Write a main( ) that calls all the different versions.
2. Demonstrate private and public keywords.
3. Use constructors to initialize instance variables. [S23]
4. Constructor overloading and method overloading demonstration. [S24]
5. Using this keyword for accessing the instance variables. (Instance Variable Hiding). [S25]
6. Using this with a Constructor. (Calling constructors from constructors) [S26]
7. Passing object references to methods [S27]
8. Compute Fibonacci sequence using recursion. [M3]
9. Use System.gc(). (Understanding Garbage collection concept.) [S28]
10. Demonstrate class variables and class level methods. (Understanding static)[S29]
11. Write a program that demonstrates that, no matter how many objects you create of a particular class, there is only one instance of a particular static field in that class.[S30]
12. Prove that class loading takes place only once. Prove that loading may be caused by either the creation of the first instance of that class or by the access of a static member.[S31]
13. Demonstrate execution sequence of static block, default block, constructor. (One should know when these blocks will be executed)[S32]
14. Demonstrate inner classes. Instantiate inner class outside of outer class. [S33]
15. Write a method in the inner class that will return the current outer class instance.[S34]
16. Create a class in a package. Create an instance of your class outside of that package.  
    (Understand packages and default package access (which has no keyword)). [S35]

## Inheritance

1. Demonstrate Inheritance [S36].
2. Prove that the base-class constructors are (a) always called and (b) called before derived-class constructors.[S37]
3. Demonstrate method overriding with covariant return types.(Java SE 5) [S38]
4. The super keyword usage. [S39]
   1. To access super class instance variables/methods
   2. To call super class constructor
5. Prove that “If super class has only parameterized constructors, then it is mandatory for the sub class to have a constructor (with or without arguments) and that subclass constructor should have an explicit call to super class’s constructor” [S40]
6. Demonstrate protected keyword. Create a class inside a package. Your class should contain a protected method. Outside of the package, try to call the protected method and explain the results. Now inherit from your class and call the protected method from inside a method of your derived class.[S41]
7. The final keyword usage with CONSTANTS, CLASSes and METHODs [S42]
8. Override toString() from Object class. [S43]
9. Program to override equals and hashcode methods properly. [C1]  
   The equals method implements an equivalence relation. It is:

* Reflexive: For any non-null reference value x, x.equals(x) must return true.
* Symmetric: For any non-null reference values x and y, x.equals(y) must return  
  true if and only if y.equals(x) returns true.
* Transitive: For any non-null reference values x, y, z, if x.equals(y) returns  
  true and y.equals(z) returns true, then x.equals(z) must return true.
* Consistent: For any non-null reference values x and y, multiple invocations  
  of x.equals(y) consistently return true or consistently return false, provided  
  no information used in equals comparisons on the objects is modified.
* For any non-null reference value x, x.equals(null) must return false.

The general contract of hashCode is:

* Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
* If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
* It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

1. Create a class as abstract without including any abstract methods and verify that you cannot create any instances of that class.[S44]
2. Create a base class with an abstract print( ) method that is overridden in a derived class. The overridden version of the method prints the value of an int variable defined in the derived class. At the point of definition of this variable, give it a nonzero value. In the base-class constructor, call this method.[S45]
3. Create an interface containing three methods, in its own package. Implement the interface in a different package.[S46]
4. Prove that all the methods in an interface are automatically public.[S47]
5. Create an interface, and inherit two new interfaces from that interface. Multiple inherit a third interface from the second two.[S48]
6. Create three interfaces, each with two methods. Inherit a new interface that combines the three, adding a new method. Create a class by implementing the new interface and also inheriting from a concrete class. Now write additional four methods, each of which takes one of the four interfaces as an argument. In main( ), create an object of your class and pass it to each of the methods. [M4]
7. Create two interfaces with same method inside those. Create a concrete class by implementing these two interfaces.[S49]

## Exception Handling

1. Demonstrate try/catch with division of two integers and Array index out of bounds exception.[S50]
2. Program to handle multiple built-in exceptions.[S51]
3. Demonstrate checked and unchecked exceptions.[S52]
4. Create a class with a main( ) that throws an object of class Exception inside a try block. Give the constructor for Exception a String argument. Catch the exception inside a catch clause and print the String argument. Add a finally clause and print a message to prove you were there.[S53]
5. Try to produce StackOverflowError, by calling a method recursively and unconditionally.[M5]
6. Create custom exceptions, and throw them using “throw” keyword.[S54]
7. Demonstrate throws keyword.[S55]
8. Create a class with two methods, f( ) and g( ). In g( ), throw an exception of a new type that you define. In f( ), call g( ), catch its exception and, in the catch clause, throw a different exception (of a second type that you define). Test your code in main( ).(Use chaining of exceptions) [M6]
9. Write a program with try with resources. (JAVA SE 5) [M7]

## Enumerations

1. Demonstrate enum in java[S56]
2. Add methods to enum. [S57]
3. Use oridinal(), valueOf() and values() methods on it.[S58]
4. Use enumerations in switch.[S59]
5. Prove “it is possible to create an enum that implements one or more interfaces”.[M8]
6. Prove that “You can declare the method abstract in the enum type and override it with a concrete method in each constant”. (Such methods are known as constant-specific methods)[C2]

# Exercises on Advanced concepts

## Generics

1. Simple java program to demonstrate Generic classes. [M9]  
   Create a holder class that holds three objects of the same type, along with the methods to store and fetch those objects and a constructor to initialize all three.
2. Implement a Generic class with two type parameters.[M10]  
   Check, are you able to overload methods with these parameters as arguments in it.
3. Implement a STACK class using generics.[C3]
4. Demonstrate bounded types.[C4]  
   Create a generic class that will perform arithmetic operations only on “Number”s.
5. Write a generic interface and implement the same.[M11]
6. Implement a generic method in a non-generic class.[M12]
7. Demonstrate bounded wildcard arguments.[C5]  
   Create a generic class [say Stats] that stores an array of “Number”s. It has a method to compute average of these numbers. Now write a method with wildcard arguments to compare two different averages. Stats<Integer>’s average and Stats<Double>’s average should be comparable.
8. Type Inference and Instantiation of Generic Classes. Demonstrate the Diamond operator (**<>**). Java SE 7 [C6]

## Multi-Threading

1. Implement a Thread class to print numbers from 1 to 10. [S60]  
   Do this with both extending Thread and implementing Runnable.
2. Implement a Runnable. Inside run( ), print a message, and then call yield( ). Repeat this three times, and then return from run( ). Put a startup message in the constructor and a shutdown message when the task terminates. Create a number of these tasks and drive them using threads. [M13]
3. Implement the solution for Producer-Consumer problem. (synchronization concepts) [C7]
   1. Using wait(), notify() methods
   2. Using BlockingQueues.
4. Implement a Blocking Queue using arrays. The following are the properties of Blocking Queue [C8]
   1. Puts a record in the empty location.
   2. Gets the record and removes it from array. (Make it null)
   3. If the queue is empty, all reading threads should wait for a value.
   4. If the queue is full, all writing threads should wait for availability of memory.
5. Read a text files in a folder find the exact occurrences of the given word in all txt files in that folder. (Exclude sub directories) [C9]
   1. Optimize the code to use Thread Pool to parallel search multiple files.   
      Hint: use fixed thread pool, blocking queue, AtomicLong variables to maintain count.
6. Write a simple program to demonstrate Callable and Future.[M14]

## Collections



Double click the above image to open it as a PDF document.

Write sample programs to demonstrate each of these. (Total 27 types collections).

## I/O

1. Print the properties of a File using the methods in java.io.File. [M15]
2. Reading & Writing files using various byte streams and character streams.
   1. FileInputStream, FileOutputStream
   2. FileReader, File Writer
   3. BufferedInputStream, BufferedOutputStream, PrintStream
   4. BufferedReader, BufferedWriter, PrintWriter
3. Read file using RandomAccessFile.[S61]
4. Copy the contents of a file to another using FileChannel and MappedByteBuffer.[M16]
5. Practice the utility methods in java.nio.file.Files.[C10]  
   This class consists exclusively of static methods that operate on files, directories, or other types of files. (Java SE 7)

## Regular expressions

1. Explore the Regular expression syntax.  
   Practice these with Pattern and Matcher.  
   <http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html>

#### Characters

x The character x

\\ The backslash character

\0n The character with octal value 0n (0 <= n <= 7)

\0nn The character with octal value 0nn (0 <= n <= 7)

\0mnn The character with octal value 0mnn (0 <= m <= 3, 0 <= n <= 7)

\xhh The character with hexadecimal value 0xhh

\uhhhh The character with hexadecimal value 0xhhhh

\t The tab character ('\u0009')

\n The newline (line feed) character ('\u000A')

\r The carriage-return character ('\u000D')

\f The form-feed character ('\u000C')

\a The alert (bell) character ('\u0007')

\e The escape character ('\u001B')

\cx The control character corresponding to x

#### Character classes

[abc] a, b, or c (simple class)

[^abc] Any character except a, b, or c (negation)

[a-zA-Z] a through z or A through Z, inclusive (range)

[a-d[m-p]] a through d, or m through p: [a-dm-p] (union)

[a-z&&[def]] d, e, or f (intersection)

[a-z&&[^bc]] a through z, except for b and c: [ad-z] (subtraction)

[a-z&&[^m-p]] a through z, and not m through p: [a-lq-z](subtraction)

#### Predefined character classes

. Any character (may or may not match line terminators)

\d A digit: [0-9]

\D A non-digit: [^0-9]

\s A whitespace character: [ \t\n\x0B\f\r]

\S A non-whitespace character: [^\s]

\w A word character: [a-zA-Z\_0-9]

\W A non-word character: [^\w]

#### POSIX character classes (US-ASCII only)

\p{Lower} A lower-case alphabetic character: [a-z]

\p{Upper} An upper-case alphabetic character:[A-Z]

\p{ASCII} All ASCII:[\x00-\x7F]

\p{Alpha} An alphabetic character:[\p{Lower}\p{Upper}]

\p{Digit} A decimal digit: [0-9]

\p{Alnum} An alphanumeric character:[\p{Alpha}\p{Digit}]

\p{Punct} Punctuation: One of !"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~

\p{Graph} A visible character: [\p{Alnum}\p{Punct}]

\p{Print} A printable character: [\p{Graph}\x20]

\p{Blank} A space or a tab: [ \t]

\p{Cntrl} A control character: [\x00-\x1F\x7F]

\p{XDigit} A hexadecimal digit: [0-9a-fA-F]

\p{Space} A whitespace character: [ \t\n\x0B\f\r]

#### java.lang.Character classes (simple java character type)

\p{javaLowerCase} Equivalent to java.lang.Character.isLowerCase()

\p{javaUpperCase} Equivalent to java.lang.Character.isUpperCase()

\p{javaWhitespace} Equivalent to java.lang.Character.isWhitespace()

\p{javaMirrored} Equivalent to java.lang.Character.isMirrored()

#### Classes for Unicode blocks and categories

\p{InGreek} A character in the Greek block (simple block)

\p{Lu} An uppercase letter (simple category)

\p{Sc} A currency symbol

\P{InGreek} Any character except one in the Greek block (negation)

[\p{L}&&[^\p{Lu}]] Any letter except an uppercase letter (subtraction)

#### Boundary matchers

^ The beginning of a line

$ The end of a line

\b A word boundary

\B A non-word boundary

\A The beginning of the input

\G The end of the previous match

\Z The end of the input but for the final terminator, if any

\z The end of the input

#### Greedy quantifiers

X? X, once or not at all

X\* X, zero or more times

X+ X, one or more times

X{n} X, exactly n times

X{n,} X, at least n times

X{n,m} X, at least n but not more than m times

#### Reluctant quantifiers

X?? X, once or not at all

X\*? X, zero or more times

X+? X, one or more times

X{n}? X, exactly n times

X{n,}? X, at least n times

X{n,m}? X, at least n but not more than m times

#### Possessive quantifiers

X?+ X, once or not at all

X\*+ X, zero or more times

X++ X, one or more times

X{n}+ X, exactly n times

X{n,}+ X, at least n times

X{n,m}+ X, at least n but not more than m times

#### Logical operators

XY X followed by Y

X|Y Either X or Y

(X) X, as a capturing group

#### Back references

\n Whatever the nth capturing group matched

#### Quotation

\ Nothing, but quotes the following character

\Q Nothing, but quotes all characters until \E

\E Nothing, but ends quoting started by \Q

#### Special constructs (non-capturing)

(?:X) X, as a non-capturing group

(?idmsux-idmsux) Nothing, but turns match flags i d m s u x on - off

(?idmsux-idmsux:X) X, as a non-capturing group with the given flags i d m s u x on - off

(?=X) X, via zero-width positive lookahead

(?!X) X, via zero-width negative lookahead

(?<=X) X, via zero-width positive lookbehind

(?<!X) X, via zero-width negative lookbehind

(?>X) X, as an independent, non-capturing group