**Java Learning Path**

**(**Java SE 8 Programmer I**)**

Table of Contents

[**1**. **Java Basics** 3](#_Toc536702586)

[**1.1 Define a scope of variables** 3](#_Toc536702587)

[**Instance variables** 3](#_Toc536702588)

[**Local Variable** 3](#_Toc536702589)

[**Static variables** 3](#_Toc536702590)

[**Block Variables** 3](#_Toc536702591)

[**1.2** **Define the structure of a Java class** 4](#_Toc536702592)

[**1.3 Create executable Java applications with a main method; run a Java program from the command line; produce console output** 4](#_Toc536702593)

[**1.3 Import other Java packages to make them accessible in your code** 5](#_Toc536702594)

[**2. Using Operators and Decision Constructs** 5](#_Toc536702595)

[**2.1 Use Java operators; use parentheses to override operator precedence** 5](#_Toc536702596)

[**2.2 Test equality between Strings and other objects using == and equals ()** 6](#_Toc536702597)

[Comparing Strings: 6](#_Toc536702598)

[Comparing Objects 6](#_Toc536702599)

[**2.3 Create if and if/else and ternary constructs** 7](#_Toc536702600)

[Ternary Operator: 7](#_Toc536702601)

[**2.4 Use a switch statement** 8](#_Toc536702602)

[**3. Using Loop Constructs** 9](#_Toc536702603)

[**3.1 Create and use while loops** 9](#_Toc536702604)

[**3.2 Create and use for loops including the enhanced for loop** 9](#_Toc536702605)

[**3.3 Create and use do/while loops** 9](#_Toc536702606)

[**3.4 Compare loop constructs** 10](#_Toc536702607)

[**3.5 Use break and continue** 10](#_Toc536702608)

[**4. Creating and Using Arrays** 11](#_Toc536702609)

[**4.1 Declare, instantiate, initialize and use a one-dimensional array** 11](#_Toc536702610)

[**4.2 Declare, instantiate, initialize and use multi-dimensional arrays** 11](#_Toc536702611)

[**5. Working with Inheritance** 12](#_Toc536702612)

[**5.1 Describe inheritance and its benefits** 12](#_Toc536702613)

[**5.2 Develop code that makes use of polymorphism; develop code that overrides methods; differentiate between the type of a reference and the type of an object** 12](#_Toc536702614)

[**5.3 Determine when casting is necessary** 13](#_Toc536702615)

[**5.4 Use super and this to access objects and constructors** 13](#_Toc536702616)

[**5.5 Use abstract classes and interfaces** 14](#_Toc536702617)

[**6. Working with Java Data Types** 16](#_Toc536702618)

[**6.1 Declare and initialize variables (including casting of primitive data types)** 16](#_Toc536702619)

[**6.2 Differentiate between object reference variables and primitive variables** 16](#_Toc536702620)

[**6.3 Know how to read or write to object fields** 17](#_Toc536702621)

[6.4 Explain an Object's Lifecycle (creation, "dereference by reassignment" and garbage collection) 18](#_Toc536702622)

[**6.5 Develop code that uses wrapper classes such as Boolean, Double, and Integer** 18](#_Toc536702623)

[**7. Working with Methods and Encapsulation** 19](#_Toc536702624)

[**7.1 Create methods with arguments and return values; including overloaded methods** 19](#_Toc536702625)

[**7.2 Apply the static keyword to methods and fields** 19](#_Toc536702626)

[**7.3 Create and overload constructors; differentiate between default and user defined constructors** 19](#_Toc536702627)

[**7.5 Apply encapsulation principles to a class** 20](#_Toc536702628)

[**7.6 Determine the effect upon object references and primitive values when they are passed into methods that change the values** 21](#_Toc536702629)

[**8. Working with Selected classes from the Java API** 22](#_Toc536702630)

[**8.1 Manipulate data using the StringBuilder class and its methods** 22](#_Toc536702631)

[**8.2 Create and manipulate Strings** 22](#_Toc536702632)

[**8.3 Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period** 23](#_Toc536702633)

[**8.4 Declare and use an ArrayList of a given type** 23](#_Toc536702634)

[**9. Handling Exceptions** 24](#_Toc536702635)

[**9.1 Differentiate among checked exceptions, unchecked exceptions, and Errors** 24](#_Toc536702636)

[**9.2 Create a try-catch block and determine how exceptions alter normal program flow** 24](#_Toc536702637)

[**9.3 Describe the advantages of Exception handling** 24](#_Toc536702638)

[**9.4 Create and invoke a method that throws an exception** 25](#_Toc536702639)

[**9.5 Recognize common exception classes (such as NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException)** 25](#_Toc536702640)

# **1**. **Java Basics**

## **1.1 Define a scope of variables**

The Scope of variable defines the part of the program where the variable accessible. There different types of variables.

### **Instance variables**

Instance variables are those defined within a class itself and outside of the method/constructor. They created when instance created .The lifetime of these variables is the same as the lifetime of the object to which it belongs.

### **Local Variable**

Local Variables are the one that we declared with in the methods/constructor. The lifetime of these variables are limited to the method itself.

### **Static variables**

Static variables will declared using static keyword. It's is common to all instances. We can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory. They survive as long as the class stays loaded in the JVM memory.

### **Block Variables**

Block variables lives only as long as code block is executing.

Example:

|  |
| --- |
| *Class ScopeOfVariables{* |
| *int = 10; // instance variable* |
| *static int y = 20; //static variable* |
| *void methodA(){* |
| *int z = 30; // local variable* |
| *}* |
| *}* |

## **1.2** **Define the structure of a Java class**

The structure of java class include below

* Package declaration
* import statements,
* class section

1. Modifiers (public, private, default, protected)
2. Class name
3. Variables
4. Methods
5. Super class/interface if any preceded by the keyword extend/implement.

Example:

|  |
| --- |
| *public class <className> extends/implement <superclass>/<interface>{* |
| *//variable declaration* |
| *//Member methods declaration* |
| *//main () method-where execution of program starts* |
| *}* |

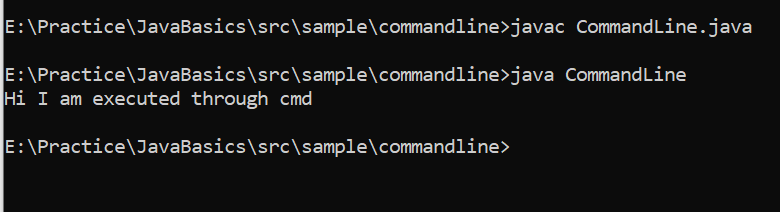
## **1.3 Create executable Java applications with a main method; run a Java program from the command line; produce console output**

To execute through command line will use below commands

* Javac : To compile
* Java : To execute

Example:

|  |
| --- |
| *public class CommandLine {* |
| *public static void main(String[] args) {* |
| *System.out.println("Hi I am executed through cmd");* |
| *}* |
| *}* |



## **1.3 Import other Java packages to make them accessible in your code**

|  |
| --- |
| *import java.text.SimpleDateFormat;* |
| *import java.util.Date;* |
| *public class PackageImport {* |
| *public static void main(String[] args) {* |
| *SimpleDateFormat sampelDate = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");* |
| *System.out.println("Time :"+sampelDate.format(new Date().getTime()));* |
| *}* |
| *}* |

# **2. Using Operators and Decision Constructs**

## **2.1 Use Java operators; use parentheses to override operator precedence**

It will follow the order when the expression has several operators. That is BODMAS.

* Brackets
* Order Of ( 2 power of 3 etc)
* Division
* Multiplication
* Addition
* Subtraction

Example:

|  |
| --- |
| *public class OperatorsPriority{* |
| *public static void main(String[] args) {* |
| *int total1= 2 + 3 \* 5;* |
| *int total= (2 + 3) \* 5;* |
| *System.out.println("Without precedence :"+total1);* |
| *System.out.println("With precedence :"+total);* |
| *}* |
| *}* |

O/P: Without precedence : 17

With precedence :25

## **2.2 Test equality between Strings and other objects using == and equals ()**

"==" compare two objects based on memory reference. so "==" operator will return true only if two object reference it is comparing represent exactly same object otherwise "==" will return false.

The equals() method tests whether two variables refer to objects that have the same state (values) or not.

### Comparing Strings:

String companyName= new String("Pramati Tech");

String officeName= new String("Pramati Tech");

result= companyName==officeName; //return false

System.out.println("Comparing two strings with == operator: " + result);

Note: even though string contain same content, two strings are different object so it will result false

result= companyName.equals(officeName); //returns true

### Comparing Objects

Object obj1 = new Object();

Object obj2 = new Object();

result = (obj1==obj2); //return false

result = obj1.equals(obj2); //return false, obj1 and obj2 are different

## **2.3 Create if and if/else and ternary constructs**

if statement will evaluate whether a statement is true or false. It will run if the statement returns true

if(boolean expression){

//piece of code to run if expression returns true

}

else block is optional, it will run only on when "if" expression runs false.

if(2>3){

//piece of code to run if expression returns true

}

else {

//piece of code to run if expression returns false

}

**else if :** on multiple possible conditions and outputs we can use "else if"

if(condition 1){

//code to execute if condition 1 is true

}

else if (condition 2){

//code to execute if condition 2 is true

}

else if (condition 3){

//code to execute if condition 3 is true

}

else {

//code to execute if above all conditions are false

}

### Ternary Operator:

Ternary Operator use to shorthand for if-else statement.

Here the syntax

Condition? If true expression: expression on false

int age =21;

String eligibility = (age>=21) ? "eligible for vote" : "Your are not eligible";

## **2.4 Use a switch statement**

Switch statements provides easy way to execute different parts of code on the values of the expression.

Example:

|  |
| --- |
| *public class SwitchStatement {* |
| *public void apiRequest(String req){* |
| *switch (req) {* |
| *case "Add":* |
| *System.out.println("Add");* |
| *break;* |
| *case "Delete":* |
| *System.out.println("delete");* |
| *break;* |
| *case "Edit":* |
| *System.out.println("edit");* |
| *break;* |
| *default:* |
| *System.out.println("not a vallid one");* |
| *break;* |
| *}* |
| *}* |
| *public static void main(String[] args) {* |
| *new SwitchStatement().apiRequest("Delete");* |
| *}* |
| *}* |

# **3. Using Loop Constructs**

## **3.1 Create and use while loops**

The while loop use to execute block of statements until the given conditions is true. The block will execute after checking the condition.

## **3.2 Create and use for loops including the enhanced for loop**

The for loop used to repeat a specific block until certain condition is met

## **3.3 Create and use do/while loops**

The do while loop use to execute block of statements until the given conditions is true. In do while guarantees at least one loop execution.

Example:

|  |
| --- |
| *public class LoopConstructsExample {* |
| *public static void main(String[] args) {* |
| *int[] value = {4,5,6};* |
| *int count =4;* |
| *//from for loop* |
| *for (int i = 0; i < value.length; i++) {* |
| *System.out.println("From for loop :"+value[i]);* |
| *}* |
| *//enahnced for loop* |
| *for (int val : value) {* |
| *System.out.println("From enhanced for loop :"+val);* |
| *}* |
| *//While loop* |
| *while (count<10) {* |
| *System.out.println("From While loop :"+ count);* |
| *count++;* |
| *}* |
| *count =1;* |
| *//do while* |
| *do {* |
| *System.out.println("from do-while : "+count);* |
| *count++;* |
| *}* |
| *while (count<=6);* |
| *}* |
| *}* |

## **3.4 Compare loop constructs**

Difference between for loop, while loop, and do while loop is

1. While loop : checks for the condition first before enter into the loop
2. do while loop: execute the statements first before checks for
3. The condition. So in this scenario at least one iteration takes places, even if the condition not met.
4. for loop: for loop used to repeat a specific block until certain condition is met

## **3.5 Use break and continue**

|  |
| --- |
| *public class BreakAndContinueExample {* |
| *public static void main(String[] args) {* |
| *int[] count ={10, 20, 30, 40, 50};* |
| *//enhanced for loop* |
| *for (int number : count) {* |
| *if(number==40){* |
| *break;* |
| *}* |
| *System.out.println(" break :"+number);* |
| *}* |
| *for (int number : count) {* |
| *if(number==40){* |
| *continue;* |
| *}* |
| *System.out.println("continue :"+number);* |
| *}* |
| *}* |
| *}* |

# **4.** **Creating and Using Arrays**

## **4.1 Declare, instantiate, initialize and use a one-dimensional array**

int[] days = new int[7];

int [] days2 = {1,2,3,4,5,6,7};

## **4.2 Declare, instantiate, initialize and use multi-dimensional arrays**

Multidimensional arrays are arrays of arrays.

int [][] multi = new int [2][3];

int [][] multi2 = { { 1, 2 }, { 3, 4, 5 } }; //two dimensional array

Example:

|  |
| --- |
| *public class ArraysExample {* |
| *public static void main(String[] args) {* |
| *int[] days = new int[7];* |
| *int [] days2 = {1,2,3,4,5,6,7}; //one dimensional array* |
| *int [][] multi = new int [2][3];* |
| *int [][] multi2 = { { 1, 2 }, { 3, 4, 5 } }; //two dimensional array* |
| *double[] num = new double[3];* |
| *num[0]=10.2;* |
| *num[1]=11.2;* |
| *num[2]=12.2;* |
| *for (double d : num) {* |
| *System.out.println(d);* |
| *}* |
| *for (int i = 0; i < multi2.length; i++) {* |
| *for (int j = 0; j < multi2[i].length; j++) {* |
| *System.out.print(multi2[i][j]);* |
| *}* |
| *System.out.println("");* |
| *}* |
| *}* |
| *}* |

# **5. Working with Inheritance**

## **5.1 Describe inheritance and its benefits**

The key benefits of the inheritance is to minimize the duplicate code by sharing common code among several subclasses.

Reusability: Facility to use public methods without rewriting the same.

Extensibility: Extending the base class logic as per business logic of the derived class.

public class <classname> extends <interface name>{ }

## **5.2 Develop code that makes use of polymorphism; develop code that overrides methods; differentiate between the type of a reference and the type of an object**

Example:

|  |
| --- |
| *abstract class Pet {* |
| *public abstract void makeSound();* |
| *}* |
|  |
| *public class Dog extends Pet {* |
| *@Override* |
| *public void makeSound() {* |
| *System.out.println("Woof");* |
| *}* |
| *}* |
|  |
| *public class Cat extends Pet{* |
| *@Override* |
| *public void makeSound() {* |
| *System.out.println("Meow");* |
| *}* |
| *}* |
|  |
|  |
| *import java.util.ArrayList;* |
| *import java.util.List;* |
|  |
| *public class PolymorphismExample {* |
| *public static void main(String[] args) {* |
| *List<Pet> pets = new ArrayList<Pet>();* |
| *pets.add(new Dog());* |
| *pets.add(new Cat());* |
| *//enhanced for loop* |
| *for (Pet pet : pets) {* |
| *pet.makeSound();* |
| *}* |
| *}* |
| *}* |

Overloading and overriding are two forms of Polymorphism available in Java.

You cannot compare Polymorphism with method Overloading and overriding. Polymorphism is ability of variable to behave differently based which kind of object it is referring.

## **5.3 Determine when casting is necessary**

Casting is the process of one variable types behaves as a variable of another type. Will do casting for primitive types and objects. There are two type of referencing variable casting

Up casting: Casting subtype to a super type.

Down casting: Casting to subtype.

## **5.4 Use super and this to access objects and constructors**

Will super keyword to access the members of parent class when child & parent have member with same name. Will call parent class constructors.

Example:

|  |
| --- |
| *public class SuperKeyWordParentClass{* |
| *String name = "Pramati Tech";* |
| *public SuperKeyWordParentClass() {* |
| *System.out.println("form super class constructor");* |
| *}* |
| *}* |
|  |
| *public class SuperKeyWordChildClass extends SuperKeyWordParentClass{* |
| *String name = "Pramati";* |
| *public SuperKeyWordChildClass(String name, int count) {* |
| *System.out.println("parameterized constructor");* |
| *System.out.println("Company Name: "+name+" Count :"+count);* |
| *}* |
| *public SuperKeyWordChildClass() {* |
| *this("Pramati Technologies",1400);* |
| *System.out.println("child class constructor");* |
| *}* |
| *public void getMessage(){* |
| *System.out.println("from get message");* |
| *System.out.println("name :"+name);* |
| *System.out.println("super.name :"+super.name);* |
| *this.getMessage("Pramati-Hyd");* |
| *}* |
| *public void getMessage(String name){* |
| *System.out.println("from parameterized get message : "+name);* |
| *}* |
| *public static void main(String[] args) {* |
| *SuperKeyWordChildClass subClass = new SuperKeyWordChildClass();* |
| *subClass.getMessage();* |
| *}* |
| *}* |

## **5.5 Use abstract classes and interfaces**

|  |
| --- |
| *public interface Interfacexample {* |
| *public static void getStaticMessage(){* |
| *System.out.println("from interface static");* |
| *}* |
| *public void getMessage();* |
| *}* |
|  |
|  |
| *public abstract class AbstractCalssExample{* |
| *abstract void getName();* |
| *abstract void getCount();* |
| *public void getBusinessUnit(){* |
| *System.out.println("Imaginea");* |
| *}* |
|  |
| *}* |
|  |
|  |
| *public class AbstarctImpl extends AbstractCalssExample implements Interfacexample{* |
|  |
| *@Override* |
| *void getName() {* |
| *System.out.println("Pramati");* |
| *}* |
| *@Override* |
| *void getCount() {* |
| *System.out.println("count");* |
| *}* |
| *@Override* |
| *public void getMessage() {* |
| *System.out.println("get message");* |
| *}* |
| *}* |
|  |
|  |
| *public class AbstractAndInterfaceExample {* |
| *public static void main(String[] args) {* |
| *AbstractCalssExample impl = new AbstarctImpl();* |
| *impl.getBusinessUnit();* |
| *impl.getCount();* |
| *((AbstarctImpl) impl).getMessage();* |
| *impl.getName();* |
| *}* |
| *}* |

# **6. Working with Java Data Types**

## **6.1 Declare and initialize variables (including casting of primitive data types)**

|  |
| --- |
| *public class DeclareAndInitalizeVariables {* |
| *public static void main(String[] args) {* |
| *int val = 20;* |
| *char a1 ='c';* |
| *float f = 12.2f;* |
| *int d = (int) f;* |
| *System.out.println("val :"+val); System.out.println("char :"+a1);* |
| *System.out.println("float :"+f); System.out.println("int :"+ d);* |
| *}* |
| *}* |

## **6.2 Differentiate between object reference variables and primitive variables**

The difference between primitive and reference type is that former can never be null. If no value is assigned that will take their default value.

Example:

For Boolean: false

For bye, short, char, int & long: 0

For float & double: 0.0

## **6.3 Know how to read or write to object fields**

|  |
| --- |
| *public class PersionInfo {* |
| *private String name;* |
| *private String location;* |
| *private int number;* |
| *public String getName() {* |
| *return name;* |
| *}* |
| *public void setName(String name) {* |
| *this.name = name;* |
| *}* |
| *public String getLocation() {* |
| *return location;* |
| *}* |
| *public void setLocation(String location) {* |
| *this.location = location;* |
| *}* |
| *public int getNumber() {* |
| *return number;* |
| *}* |
| *public void setNumber(int number) {* |
| *this.number = number;* |
| *}* |
|  |
| *}* |
| *public class ReadingObjFields {* |
| *public static void main(String[] args) {* |
| *PersionInfo info = new PersionInfo();* |
| *info.setLocation("hyderabad");* |
| *info.setName("Venkatesh Kakani");* |
| *info.setNumber(456);* |
| *System.out.println("Name:"+info.getName());* |
| *System.out.println("Location:"+info.getLocation());* |
| *System.out.println("Number:"+info.getNumber());* |
| *}* |
|  |
| *}* |

## 6.4 Explain an Object's Lifecycle (creation, "dereference by reassignment" and garbage collection)

Here is the states in Object lifecycle.

* Create: New memory allocation on object creation and it will directly move to In Use state.
* In use: If the Object will held at least by one reference, then we consider to be "In Use".
* Invisible: If there are no longer reference variable that are accessible to the code (there might be references).
* Unreachable: If no more strong references exist.
* Collected: If the garbage collector has recognized as unreachable object.
* Finalized: An object is in the “finalized” state if it is still unreachable after it’s finalize method.
* De-Allocated : Final step, still unreachable after all the above work has done, then this is the state for deallocation

## **6.5 Develop code that uses wrapper classes such as Boolean, Double, and Integer**

|  |
| --- |
| public class WrapperClassExample { |
| public static void main(String[] args) { |
| int i = 23; double d = 22.3; |
| long l = 12345678; float f = 1.2f; |
| char c = 'A'; boolean bool = true; |
|  |
| System.out.println("Primitives :"); |
| System.out.println("i :"+i+" d:"+d+" l:"+l+" f:"+f+" c:"+c+" bool:"+bool ); |
| //wrapper |
|  |
| Integer ival = new Integer(i); |
| Boolean bval = new Boolean(bool); |
| Double dval = new Double(d); |
| Float fval = new Float(f); |
| Character cval = new Character(c); |
| System.out.println("Wrapper:"); |
| System.out.println("ival :"+ival+" dval:"+dval+" fval:"+fval+" cval:"+cval+" bool:"+bval ); |
| } |
| } |

# 

# **7. Working with Methods and Encapsulation**

## **7.1 Create methods with arguments and return values; including overloaded methods**

|  |
| --- |
| *public int getEmployeeCode(int recordId){* |
| *return 12268;* |
| *}* |
| *public int getEmployeeCode(int recordId, String name){* |
| *return 12268;* |
| *}* |

## **7.2 Apply the static keyword to methods and fields**

|  |
| --- |
| *static int employee\_id = 12268;* |
| *public static String getEmployeeCode(){* |
| *return "Pramati";* |
| *}* |

## **7.3 Create and overload constructors; differentiate between default and user defined constructors**

|  |
| --- |
| *Encapsulation () {* |
| *System.out.println ("From default constructor");* |
| *}* |
|  |
| *Encapsulation (String name) {* |
| *System.out.println ("From parameterized constructor");* |
| *}* |

## **7.5 Apply encapsulation principles to a class**

|  |
| --- |
| *public class Encapsulation {* |
| *static int employee\_id = 12268;* |
| *private int id;* |
| *//constructor overloading* |
| *Encapsulation (){* |
| *System.out.println("From default constructor");* |
| *}* |
| *Encapsulation (String name){* |
| *System.out.println("From parameterized constructor");* |
| *}* |
| *public int getId() {* |
| *return id;* |
| *}* |
| *public void setId(int id) {* |
| *this.id = id;* |
| *}* |
|  |
| *//Overloaded Method* |
| *public String getEmployeeCode(int recordId){* |
| *return "from employee code with one orgs";* |
| *}* |
| *public String getEmployeeCode(int recordId, String name){* |
| *return "from employee code with two orgs";* |
| *}* |
| *//static method* |
| *public static String getEmployeeCode(){* |
| *return "From static method";* |
| *}* |
| *}* |

|  |
| --- |
| public class EncapsulationImpl { |
| public static void main(String[] args) { |
| Encapsulation encap = new Encapsulation(); |
| encap.setId(12172); |
| System.out.println("Encap :"+ encap.getId()); |
| System.out.println(encap.getEmployeeCode(12181)); |
| System.out.println(encap.getEmployeeCode(13014, "Uday")); |
| System.out.println(Encapsulation.getEmployeeCode()); |
| Encapsulation encap\_p = new Encapsulation("Pramati"); |
| } |
| } |

## **7.6 Determine the effect upon object references and primitive values when they are passed into methods that change the values**

When primitives are passed in method a copy pasted inside method into local variable can change only inside the method body and original will be remain same.

When we pass object reference to a method, the change in the method will change the object via object reference.

Example:

|  |
| --- |
| *public class PrimitiveAndObjectReference {* |
| *public static void main(String[] args) {* |
| *PrimitiveAndObjectReference ref = new PrimitiveAndObjectReference();* |
| *int count = 12;* |
| *ref.getCount(count);* |
| *System.out.println("get count :"+ count +" //original value not changed");* |
| *StringBuilder sb = new StringBuilder("Msg from ouside");* |
| *System.out.println("from sb :"+sb);* |
| *ref.getMesg(sb);* |
| *System.out.println("from sb :"+sb +" //original value got effected");* |
| *}* |
| *void getCount(int val){* |
| *val = 10;* |
| *System.out.println("from primitive :"+ val);* |
| *}* |
| *void getMesg(StringBuilder sb){* |
| *sb.append(" got changed");* |
| *}* |
| *}* |

# **8. Working with Selected classes from the Java API**

## **8.1 Manipulate data using the StringBuilder class and its methods**

|  |
| --- |
| *public class StringBuilderExample {* |
| *public static void main(String[] args) {* |
| *StringBuilder builder = new StringBuilder("Pramati Tech : ");* |
| *System.out.println(builder);* |
| *builder.append("It's Product based");* |
| *System.out.println(builder);* |
| *builder.insert(13, "Pvt Ltd");* |
| *System.out.println(builder);* |
| *}* |
| *}* |

## **8.2 Create and manipulate Strings**

|  |
| --- |
| *public class StringExample {* |
| *public static void main(String[] args) {* |
| *String str = "Pramati Technologies pvt ltd.”;* |
| *System.out.println(str.charAt(3));* |
| *System.out.println(str.length());* |
| *System.out.println(str.concat("from hyd"));* |
| *System.out.println(str.equalsIgnoreCase("pramati technologies pvt ltd."));* |
| *System.out.println(str.substring(3));* |
| *System.out.println(str.toLowerCase());* |
| *System.out.println(str.replace("Technologies", "Tech"));* |
| *}* |
| *}* |

## **8.3 Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period**

|  |
| --- |
| *import java.time.LocalDate;* |
| *import java.time.LocalDateTime;* |
| *import java.time.LocalTime;* |
| *import java.time.Period;* |
| *public class LocalDateExample {* |
| *public static void main(String[] args) {* |
| *LocalDateTime dateTime = LocalDateTime.now();* |
| *LocalDate date = LocalDate.now();* |
| *LocalTime time = LocalTime.now();* |
| *Period period = Period.of(2019, 01, 05);* |
| *System.out.println(dateTime);* |
| *System.out.println(LocalDateTime.of(date, time));* |
| *System.out.println(date);* |
| *System.out.println(LocalDate.of(2019, 01, 03));* |
| *System.out.println(time);* |
| *System.out.println(LocalTime.of(14,30)+" PM");* |
| *System.out.println(period);* |
|  |
| *}* |
|  |
| *}* |

## **8.4 Declare and use an ArrayList of a given type**

|  |
| --- |
| *import java.util.ArrayList;* |
| *public class ArrayLiatExample {* |
| *public static void main(String[] args) {* |
| *ArrayList<String> list = new ArrayList<String>();* |
| *list.add("Pramati");* |
| *list.add(1, "Service");* |
| *list.add(2, "Manipulate");* |
| *for (String string : list) {* |
| *System.out.println(string);* |
| *}* |
| *}* |
| *}* |

# **9. Handling Exceptions**

## **9.1 Differentiate among checked exceptions, unchecked exceptions, and Errors**

Throwable is the top of all exceptions, under Throwable we have Error and Exception. Under Exception we have RuntimeException. We have two type of exceptions checked and unchecked exception.

Checked: Exceptions are that are checked at compile time.

Unchecked: Exceptions that are not checked at compile time.

In Java exceptions under Error and RuntimeException classes are unchecked exceptions. Everything else under throwable is checked.

## **9.2 Create a try-catch block and determine how exceptions alter normal program flow**

Refer 9.4

## **9.3 Describe the advantages of Exception handling**

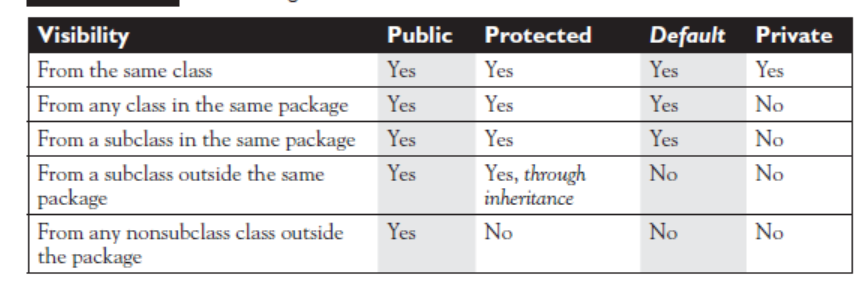
* Easy identification of program code and error-handling code.
* Propagation of errors.
* Grouping exceptions based on their type.

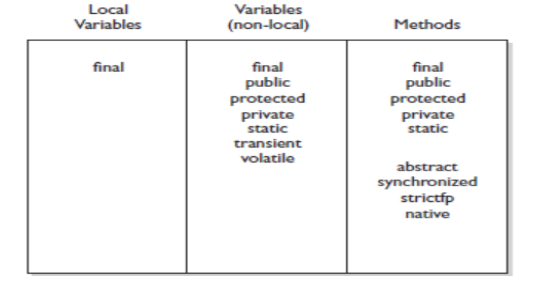
## **9.4 Create and invoke a method that throws an exception**

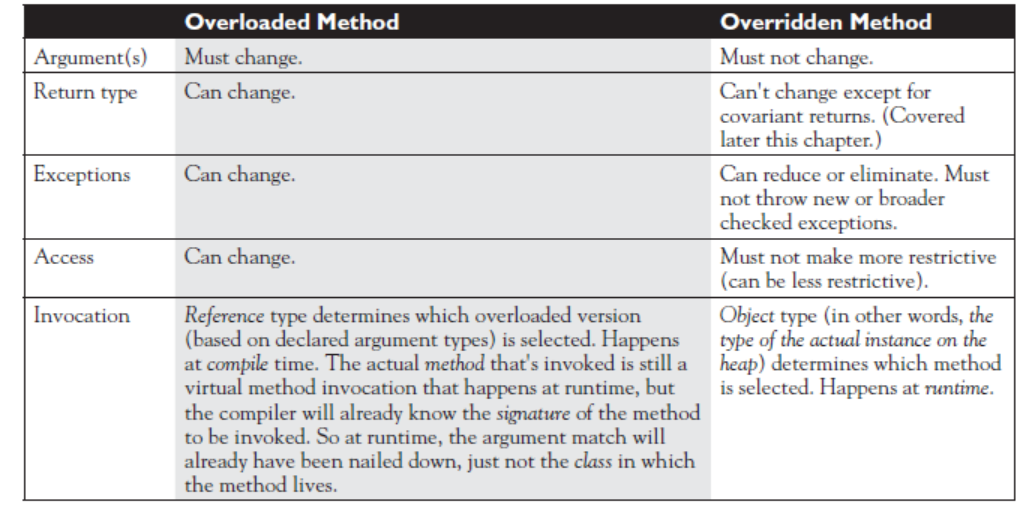
|  |
| --- |
| *public class ExceptionExample {* |
| *public static void main(String[] args) {* |
| *String name = null;* |
| *try {* |
| *if(name.equals("Pramati")){* |
| *System.out.println("both are equal");* |
| *}* |
| *else* |
| *System.out.println("both are not equal");* |
|  |
| *} catch (Exception e) {* |
| *System.out.println("from catch block");* |
| *}* |
| *}* |

## **9.5 Recognize common exception classes (such as NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException)**

* NullPointerException is thrown when an application trying to use an object reference which has value "null".
* Thrown when an exceptional arithmetic condition occurred. An integer divide by zero.
* ArrayIndexOutOfBoundsException is thrown to indicate that we are trying to access element with illegal index.
* ClassCastException in java is a run time error it occurs when an object cannot be casted to another type.

****

**Comparison of modifiers on variables vs. methods**

****