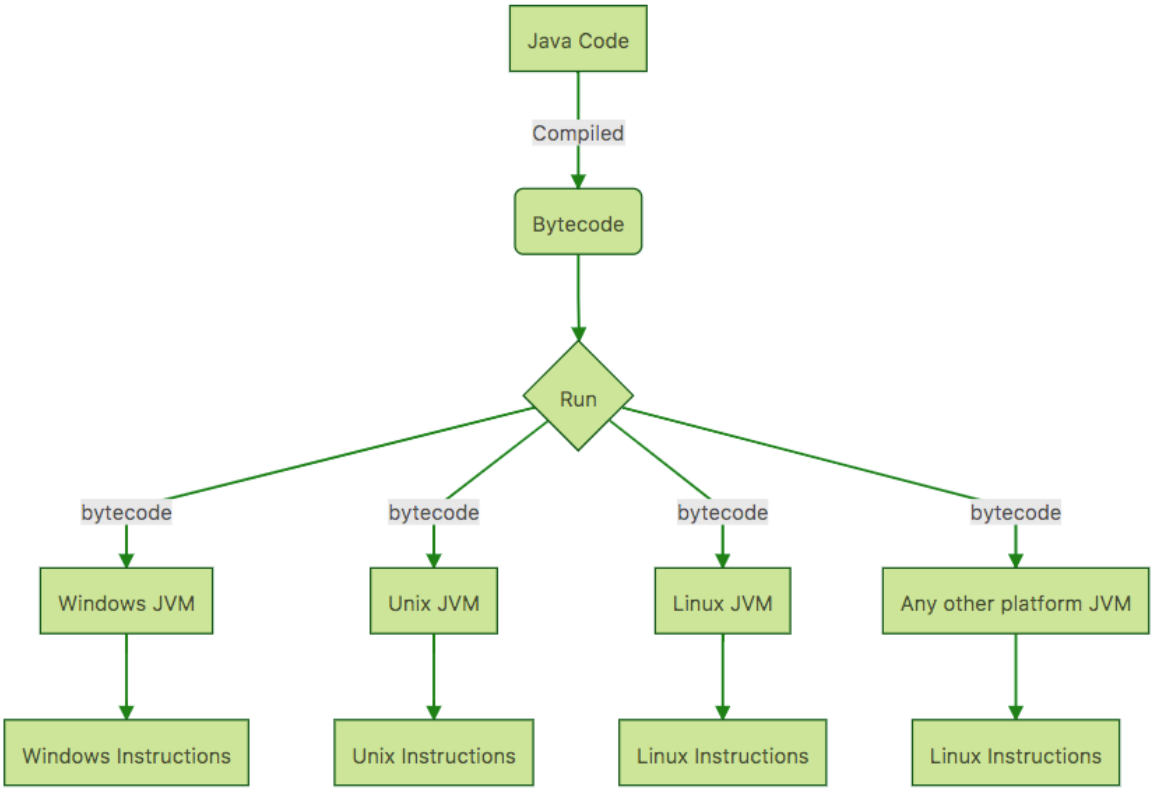
Popularity of Java

- Platform Independent or Portable
- Object Oriented Language
- Security
- Rich API
- Great IDE's
- Omnipresent
    - Web Applications (Java EE (JSP, Servlets), Spring, Struts..)
    - Mobile Apps(Android)
    - Microservices (Spring Boot)

Platform Independence

- Build once, run anywhere



- Java bytecode is the instruction set of the Java virtual machine

```
graph TD
A[Java Code] -->|Compiled| B(Bytecode)
B --> C{Run}
```

```
C -->|bytecode| D[Windows JVM]
D --> K[Windows Instructions]
C -->|bytecode| E[Unix  JVM]
E --> L[Unix Instructions]
C -->|bytecode| F[Linux  JVM]
F --> M[Linux Instructions]
C -->|bytecode| G[Any other platform  JVM]
G --> N[Linux Instructions]
```

JDK vs JVM VS JRE

- JVM (Java Virtual Machine)
    - runs the Java bytecode.
- JRE
    - JVM + Libraries + Other Components (to run applets and other java applications)
- JDK
    - JRE + Compilers + Debuggers

ClassLoader

- Find and Loads Java Classes!

Three Types

- System Class Loader - Loads all application classes from CLASSPATH
- Extension Class Loader - Loads all classes from extension directory
- Bootstrap Class Loader - Loads all the Java core files

Order of execution of ClassLoaders

- JVM needs to find a class, it starts with System Class Loader.
- If it is not found, it checks with Extension Class Loader.
- If it not found, it goes to the Bootstrap Class Loader.
- If a class is still not found, a ClassNotFoundException is thrown.

Passing Variables to Methods

- All variables , primitives and references , in Java, are passed to functions using copy-of-variable-value.

Important Rules

- Member/Static variables are alway initialized with default values.
- Default values for numeric types is 0, floating point types is 0.0, boolean is false, char is '\u0000' and for a object reference variable is null.
- Local variables are not initialized by default by compiler.
- Using a local variable before initialization results in a compilation error.
- Assigning a null value is a valid initialization for reference variables.

Wrapper Classes

- Example 1
- A wrapper class wraps (encloses) around a data type and gives it an object appearance
- Wrapper: Boolean,Byte,Character,Double,Float,Integer,Long,Short
- Primitive: boolean,byte,char ,double, float, int , long,short
- Examples of creating wrapper classes are listed below.
    - Integer number = new Integer(55);//int;
    - Integer number2 = new Integer("55");//String
    - Float number3 = new Float(55.0);//double argument
    - Float number4 = new Float(55.0f);//float argument
    - Float number5 = new Float("55.0f");//String
    - Character c1 = new Character('C');//Only char constructor
    - Boolean b = new Boolean(true);
- Reasons
    - null is a possible value
    - use it in a Collection
    - Object like creation from other types.. like String

Wrapper Class Utility Methods

xxxValue methods help in creating primitives

```
Integer integer = Integer.valueOf(57);
int primitive = integer.intValue();//57
float primitiveFloat = integer.floatValue();//57.0f
```

parseXxx methods are similar to valueOf but they return primitive values

```
int hundredPrimitive =
    Integer.parseInt("100");//100 is stored in variable
```

Boxing and new instances

- Auto Boxing helps in saving memory by reusing already created Wrapper objects. However wrapper classes created using new are not reused.
- Two wrapper objects created using new are not same object.

```
Integer nineA = new Integer(9);
Integer nineB = new Integer(9);
System.out.println(nineA == nineB);//false
System.out.println(nineA.equals(nineB));//true
```

- Two wrapper objects created using boxing are same object.

```
Integer nineC = 9;
Integer nineD = 9;
System.out.println(nineC == nineD);//true
System.out.println(nineC.equals(nineD));//true
```

Strings are immutable

- Value of a String Object once created cannot be modified. Any modification on a String object creates a new String object.

```
String str3 = "value1";
str3.concat("value2");
System.out.println(str3); //value1
```

Note that the value of str3 is not modified in the above example. The result should be assigned to a new reference variable (or same variable can be reused).

```
String concat = str3.concat("value2");
System.out.println(concat); //value1value2
```

Where are string literals stored in memory?

All strings literals are stored in "String constant pool". If compiler finds a String literal,it checks if it exists in the pool. If it exists, it is reused. Following statement creates 1 string object (created on the pool) and 1 reference variable.

```
String str1 = "value";
```

However, if new operator is used to create string object, the new object is created on the heap. Following piece of code create 2 objects.

```
//1. String Literal "value" - created in the "String constant pool"
//2. String Object - created on the heap
String str2 = new String("value");
```

String vs StringBuffer vs StringBuilder

- Immutability : String
- Thread Safety : String(immutable), StringBuffer
- Performance : StringBuilder (especially when a number of modifications are made.)
- Example 1

String Constant Pool

- All strings literals are stored in "String constant pool". If compiler finds a String literal,it checks if it exists in the pool. If it exists, it is reused.
- Following statement creates 1 string object (created on the pool) and 1 reference variable.

```
String str1 = "value";
```

- However, if new operator is used to create string object, the new object is created on the heap. Following piece of code create 2 objects.

```
//1. String Literal "value" - created in the "String constant pool"
//2. String Object - created on the heap
String str2 = new String("value");
```

String Method Examples

String class defines a number of methods to get information about the string content.

```
String str = "abcdefghijk";
```

Get information from String

Following methods help to get information from a String.

```
//char charAt(int paramInt)
System.out.println(str.charAt(2)); //prints a char - c
System.out.println("ABCDEFGH".length());//8
System.out.println("abcdefghij".toString()); //abcdefghij
System.out.println("ABC".equalsIgnoreCase("abc"));//true

//Get All characters from index paramInt
//String substring(int paramInt)
System.out.println("abcdefghij".substring(3)); //defghij

//All characters from index 3 to 6
System.out.println("abcdefghij".substring(3,7)); //defg

String s1 = new String("HELLO");
String s2 = new String("HELLO");
System.out.println(s1 == s2); // false
System.out.println(s1.equals(s2)); // true
```

String Manipulation methods

Most important thing to remember is a String object cannot be modified. When any of these methods are called, they return a new String with the modified value. The original String remains unchanged.

```
//String concat(String paramString)
System.out.println(str.concat("lmn"));//abcdefghijklmn

//String replace(char paramChar1, char paramChar2)
System.out.println("012301230123".replace('0', '4'));//412341234123

//String replace(CharSequence paramCharSequence1, CharSequence paramCharSequence2)
System.out.println("012301230123".replace("01", "45"));//452345234523

System.out.println("ABCDEFGHIJ".toLowerCase()); //abcdefghij

System.out.println("abcdefghij".toUpperCase()); //ABCDEFGHIJ

//trim removes leading and trailings spaces
System.out.println(" abcd ".trim()); //abcd
```

String Concatenation Operator

Three Rules of String Concatenation

- RULE1: Expressions are evaluated from left to right.Except if there are parenthesis.
- RULE2: number + number = number
- RULE3: number + String = String

Reference Variables

```
Integer aReference = new Integer(5);
Integer bReference = new Integer(5);
```

For reference variables, == compares if they are referring to the same object.

```
System.out.println(aReference == bReference);//false
```

Operator & and |

- Logical Operators &, | are similar to &&, || except that they don't short ciruit.
- They execute the second expression even if the result is decided.

Certification Tip : While & and | are very rarely used, it is important to understand them from a certification perspective.

```
int j = 10;
System.out.println(true | ++j==11);//true
System.out.println(false & ++j==12);//false
System.out.println(j);//j becomes 12, as both ++j expressions are executed
```

Arrays

```
//Declaring an Array
int[] marks;

// Creating an array
marks = new int[5]; // 5 is size of array

int marks2[] = new int[5];//Declaring and creating an array in same line.

System.out.println(marks2[0]);//New Arrays are always initialized with default values - 0

//Index of elements in an array runs from 0 to length - 1
marks[0] = 25;

System.out.println(marks[2]);//Printing a value from array

//Printing a 1D Array
int marks5[] = { 25, 30, 50, 10, 5 };
System.out.println(marks5); //[I@6db3f829
System.out.println(
      Arrays.toString(marks5));//[25, 30, 50, 10, 5]

int length = marks.length;//Length of an array: Property length

//Enhanced For Loop
for (int mark: marks) {
      System.out.println(mark);
}

//Fill array with a value
Arrays.fill(marks, 100); //All array values will be 100
```

```
//String Arrays
String[] daysOfWeek = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday" };

//Comparing Arrays
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 4, 5, 6 };

System.out.println(Arrays
.equals(numbers1, numbers2)); //false

int[] numbers3 = { 1, 2, 3 };

System.out.println(Arrays
.equals(numbers1, numbers3)); //true

// Sorting an Array

int rollNos[] = { 12, 5, 7, 9 };
Arrays.sort(rollNos);
System.out.println(Arrays.toString(rollNos));//[5, 7, 9, 12]

//Declaring 2D Array Examples:
int[][] matrix1; //Recommended
int[] matrix2[]; //Legal but not readable. Avoid.
```

Switch Statement

- Choose between a set of options.
- From Java 6, String can be used as the switch argument.

```
//Example 1

int number = 2;
switch (number) {
case 1:
     System.out.println(1);
     break;
case 2:
     System.out.println(2);//PRINTED
     break;
case 3:
     System.out.println(3);
     break;
default:
     System.out.println("Default");
     break;
}
```

Important Tips

- There is a break statement in every case. If there is no break statement, switch continues to execute other cases.

- There is a case named default. If none of the cases match default case is executed.

Enum

- Enum allows specifying a list of valid values (or allowed values) for a Type.

Enum Declaration

Consider the example below. It declares an enum Season with 4 possible values.

```
enum Season {
   WINTER, SPRING, SUMMER, FALL
};
```

Enum Example 1

```
//Enum can be declared outside a class
enum SeasonOutsideClass {
        WINTER, SPRING, SUMMER, FALL
};

public class Enum {
        // Enum can be declared inside a class
        enum Season {
                WINTER, SPRING, SUMMER, FALL
        };

        public static void main(String[] args) {
                /*
                 * //Uncommenting gives compilation error //enum cannot be created in a
                 * <main></main>ethod enum InsideMethodNotAllowed { WINTER, SPRING, SUMMER,
FALL };
                 */

                // Converting String to Enum
                Season season = Season.valueOf("FALL");

                // Converting Enum to String
                System.out.println(season.name());// FALL

                // Default ordinals of enum
                // By default java assigns ordinals in order
                System.out.println(Season.WINTER.ordinal());// 0
                System.out.println(Season.SPRING.ordinal());// 1
                System.out.println(Season.SUMMER.ordinal());// 2
                System.out.println(Season.FALL.ordinal());// 3

                // Looping an enum => We use method values
                for (Season season1 : Season.values()) {
                        System.out.println(season1.name());
                        // WINTER SPRING SUMMER FALL (separate lines)
                }
```

```
                // Comparing two Enums
                Season season1 = Season.FALL;
                Season season2 = Season.FALL;
                System.out.println(season1 == season2);// true
                System.out.println(season1.equals(season2));// true
        }
}
```

Enum Rules

- Enums can be declared in a separate class(SeasonOutsideClass) or as member of a class(Season). Enums cannot be declared in a method.

Conversion of Enum : Function valueOf(String) is used to convert a string to enum.

```
//Converting String to Enum
Season season = Season.valueOf("FALL");
```

Function name() is used to find String value of an enum.

```
//Converting Enum to String
System.out.println(season.name());//FALL
```

Java assigns default ordinals to an enum in order. However, it is not recommended to use ordinals to perform logic.

```
//Default ordinals of enum
// By default java assigns ordinals in order
System.out.println(Season.WINTER.ordinal());//0
System.out.println(Season.SPRING.ordinal());//1
System.out.println(Season.SUMMER.ordinal());//2
System.out.println(Season.FALL.ordinal());//3
```

Looping around an Enum - List of values allowed for an Enum can be obtained by invoking the function values().

```
//Looping an enum => We use method values
for (Season season1: Season.values()) {
    System.out.println(season1.name());
    //WINTER SPRING SUMMER FALL (separate lines)
}
```

Comparing two Enums

```
//Comparing two Enums
Season season1 = Season.FALL;
Season season2 = Season.FALL;
System.out.println(season1 == season2);//true
System.out.println(season1.equals(season2));//true
```

Enum Example 2

```
package com.in28minutes.java.beginners.concept.examples.enums;

public class EnumAdvanced {

        // Enum with a variable,method and constructor
        enum SeasonCustomized {
                WINTER(1), SPRING(2), SUMMER(3), FALL(4);
```

```
		// variable
		private int code;

		// method
		public int getCode() {
			return code;
		}

		// Constructor-only private or (default)
		// modifiers are allowed
		SeasonCustomized(int code) {
			this.code = code;
		}

		// Getting value of enum from code
		public static SeasonCustomized valueOf(int code) {
			for (SeasonCustomized season : SeasonCustomized.values()) {
				if (season.getCode() == code)
					return season;
			}
			throw new RuntimeException("value not found");// Just for kicks
		}

		// Using switch statement on an enum
		public int getExpectedMaxTemperature() {
			switch (this) {
			case WINTER:
				return 5;
			case SPRING:
			case FALL:
				return 10;
			case SUMMER:
				return 20;
			}
			return -1;// Dummy since Java does not recognize this is possible :)
		}

	};

	public static void main(String[] args) {
		SeasonCustomized season = SeasonCustomized.WINTER;

		/*
		 * //Enum constructor cannot be invoked directly //Below line would
		 * cause COMPILER ERROR SeasonCustomized season2 = new
		 * SeasonCustomized(1);
		 */

		System.out.println(season.getCode());// 1

		System.out.println(season.getExpectedMaxTemperature());// 5

		System.out.println(SeasonCustomized.valueOf(4));// FALL

	}

}
```

More Enum Basics

- Enums can contain variables, methods, constructors. In example 2, we created a local variable called code with a getter.
- We also created a constructor with code as a parameter.

```
//variable
private int code;

//method
public int getCode() {
     return code;
}

//Constructor-only private or (default)
//modifiers are allowed
SeasonCustomized(int code) {
     this.code = code;
}
```

Each of the Season Type's is created by assigning a value for code.

```
WINTER(1), SPRING(2), SUMMER(3), FALL(4);
```

Enum constructors can only be (default) or (private) access. Enum constructors cannot be directly invoked.

```
/*//Enum constructor cannot be invoked directly
  //Below line would cause COMPILER ERROR
SeasonCustomized season2 = new SeasonCustomized(1);
*/
```

Example below shows how we can use a switch around an enum.

```
// Using switch statement on an enum
public int getExpectedMaxTemperature() {
          switch (this) {
          case WINTER:
                    return 5;
          case SPRING:
          case FALL:
                    return 10;
          case SUMMER:
                    return 20;
          }
          return -1;
}
```

Inheritance

Inheritance allows extending a functionality of a class and also promotes reuse of existing code.

Every Class extends Object class

- Every class in Java is a sub class of the class Object.
- When we create a class in Java, we inherit all the methods and properties of Object class.

```
String str = "Testing";
System.out.println(str.toString());
System.out.println(str.hashCode());
System.out.println(str.clone());

if(str instanceof Object){
	System.out.println("I extend Object");//Will be printed
}
```

Multiple Inheritance results in a number of complexities. Java does not support Multiple Inheritance.

Inheritance and Polymorphism

Polymorphism is defined as "Same Code" having "Different Behavior".

Example

```
public class Animal {
	public String shout() {
		return "Don't Know!";
	}
}

class Cat extends Animal {
	public String shout() {
		return "Meow Meow";
	}
}

class Dog extends Animal {
	public String shout() {
		return "BOW BOW";
	}

	public void run(){

	}
}
```

Execution

```
Animal animal1 = new Animal();
System.out.println(animal1.shout()); //Don't Know!

Animal animal2 = new Dog();//Animal reference used to store Dog object

//Reference variable type => Animal
//Object referred to => Dog
//Dog's bark method is called.
System.out.println(animal2.shout()); //BOW BOW

//Cannot invoke sub class method with super class reference variable.
//animal2.run();//COMPILE ERROR
```

equals method

equals method is used to compare if two objects are having the same content.

- Default implementation of equals method is defined in Object class. The implementation is similar to == operator.
- By default, two object references are equal only if they are pointing to the same object.
- However, we can override equals method and provide a custom implementation to compare the contents for an object.

Signature of the equals method is "public boolean equals(Object obj) ".

- Note that "public boolean equals(Client client)" will not override the equals method defined in Object. Parameter should be of type Object.
- The implementation of equals method checks if the id's of both objects are equal. If so, it returns true.
- Note that this is a basic implementation of equals.

Abstract Class

An abstract class cannot be instantiated.

```
public abstract class AbstractClassExample {
      public static void main(String[] args) {
         //An abstract class cannot be instantiated
         //Below line gives compilation error if uncommented
         //AbstractClassExample ex = new AbstractClassExample();
      }
}

//Abstract class can contain instance and static variables

public abstract class AbstractClassExample {

      //Abstract class can contain instance and static variables
      public int publicVariable;
```

```
    private int privateVariable;
    static int staticVariable;

}

//An Abstract method does not contain body.
//Abstract Class can contain 0 or more abstract methods
//Abstract method does not have a body
abstract void abstractMethod1();
abstract void abstractMethod2();

//Abstract method can be declared only in Abstract Class.
class NormalClass{
    abstract void abstractMethod();//COMPILER ERROR
}

// Abstract class can contain fully defined non-abstract methods.
public abstract class AbstractClassExample {

    //Abstract class can contain instance and static variables
    public int publicVariable;
    private int privateVariable;
    static int staticVariable;

    //Abstract Class can contain 0 or more abstract methods
    //Abstract method does not have a body
    abstract void abstractMethod1();
    abstract void abstractMethod2();

    //Abstract Class can contain 0 or more non-abstract methods
    public void nonAbstractMethod(){
      System.out.println("Non Abstract Method");
    }

    public static void main(String[] args) {
      //An abstract class cannot be instantiated
      //Below line gives compilation error if uncommented
      //AbstractClassExample ex = new AbstractClassExample();
    }
}

//Extending an abstract class
class SubClass2 extends AbstractClassExample {
    void abstractMethod1() {
      System.out.println("Abstract Method1");
    }

    void abstractMethod2() {
      System.out.println("Abstract Method2");
    }
}

// A concrete sub class should implement all abstract methods.
// Below class gives compilation error if uncommented
/*
class SubClass extends AbstractClassExample {

}
```

```
*/

//An abstract sub class need not implement all abstract methods.
abstract class AbstractSubClass extends AbstractClassExample {
      void abstractMethod1() {
         System.out.println("Abstract Method1");
      }
      //abstractMethod2 is not defined.
}
```

Tips

- Abstract Methods cannot be paired with final or private access modifiers.
- A variable cannot be abstract.

Constructors

- Constructor is invoked whenever we create an instance(object) of a Class. We cannot create an object without a constructor. If we do not provide a constructor, compiler provides a default no-argument constructor.

Creating a Constructor

If we provide a constructor in the class, compiler will NOT provide a default constructor. In the example below we provided a constructor "public Animal(String name)". So, compiler will not provide the default constructor.

Constructor has the same name as the class and no return type. It can accept any number of parameters.

Provide No Argument Constructor

If we want to allow creation of an object with no constructor arguments, we can provide a no argument constructor as well.

Constructor Example 4: Calling a Super Class Constructor

A constructor can invoke another constructor, or a super class constructor, but only as first statement in the constructor. Another constructor in the same class can be invoked from a constructor, using this({parameters}) method call. To call a super class constructor, super({parameters}) can be used.

```
class Animal {
     String name;

     public Animal() {
this.name = "Default Name";
     }
```

```
    // This is called a one argument constructor.
```

Both example constructors below can replace the no argument "public Animal() " constructor in Example 3.

```
public Animal() {
    super();
    this.name = "Default Name";
}

public Animal() {
    this("Default Name");
}
```

super() or this() should be first statements in a Constructor.

Member variables/methods should not be used in constructor calls (super or this). Static variables or methods can be used.

```
public Animal() {
    //member variable cannot be used in a constructor call
    this(name);//COMPILER ERROR since name is member variable
}
```

A constructor cannot be explicitly called from any method except another constructor.

If a super class constructor is not explicitly called from a sub class constructor, super class (no argument) constructor is automatically invoked (as first line) from a sub class constructor.

Since a subclass constructor explicitly calls a super class constructor with no arguments, this can cause a few compiler errors.

```
class Animal {
    String name;

    public Animal(String name) {
this.name = name;
System.out.println("Animal Constructor");
    }
}

class Dog extends Animal {
    public Dog() { // COMPILER ERROR! No constructor for Animal()
System.out.println("Dog Constructor");
    }
}
```

Constructors are NOT inherited.

new Dog("Terry") is not allowed even though there is a constructor in the super class Animal with signature public Animal(String name).

Interface

Methods in an interface

Interface methods are by default public and abstract. A concrete default method (fully defined method) can be created in an interface. Consider the example below:

```
interface ExampleInterface1 {
	//By default - public abstract. No other modifier allowed
	void method1();//method1 is public and abstract
	//private void method6();//COMPILER ERROR!

}
```

Extending an Interface

An interface can extend another interface. Consider the example below:

```
interface SubInterface1 extends ExampleInterface1{
	void method3();
}
```

Interface , Things to Remember

A class should implement all the methods in an interface, unless it is declared abstract. A Class can implement multiple interfaces.

Method Overloading

- A method having the same name as another method (in same class or a sub class) but having different parameters is called an Overloaded Method.

Method Overriding

- Creating a Sub Class Method with same signature as that of a method in SuperClass is called Method Overriding.

Overriding Method Cannot have lesser visibility

Overriding method cannot have lesser visibility than the Super Class method.

Overriding method cannot throw new Checked Exceptions

Consider the example below:

```
class SuperClass{
	public void publicMethod() throws FileNotFoundException{
```

```
    }
}

class  SubClass{
    //Cannot throw bigger exceptions than Super Class
    public  void  publicMethod( )  /*throws IOException*/  {

    }
}
```

Other Overriding Rules

A Sub Class can override only those methods that are visible to it. Methods marked as static or final cannot be overridden. You can call the super class method from the overriding method using keyword super.

Class Modifiers

Non-access modifiers

strictfp, final, abstract modifiers are valid on a class.

Class Access Modifiers

private

a. Private variables and methods can be accessed only in the class they are declared.
b. Private variables and methods from SuperClass are NOT available in SubClass.

default or package

a. Default variables and methods can be accessed in the same package Classes. b. Default variables and methods from SuperClass are available only to SubClasses in same package.

protected

a. Protected variables and methods can be accessed in the same package Classes. b. Protected variables and methods from SuperClass are available to SubClass in any package

public

a. Public variables and methods can be accessed from every other Java classes. b. Public variables and methods from SuperClass are all available directly in the SubClass

Final modifier

- Let's discuss about Final modifier in Java.

Final class cannot be extended

Consider the class below which is declared as final.

Final methods cannot be overriden.

Consider the class FinalMemberModifiersExample with method finalMethod which is declared as final.

Final variable values cannot be changed.

Nested Class

- Nested Classes are classes which are declared inside other classes.

Inner Class

Generally the term inner class is used to refer to a non-static class declared directly inside another class. Consider the example of class named InnerClass.

Static Inner Class

A class declared directly inside another class and declared as static. In the example above, class name StaticNestedClass is a static inner class.

Method Inner Class

A class declared directly inside a method. In the example above, class name MethodLocalInnerClass is a method inner class.

Inner class cannot be directly instantiated.

```
// InnerClass innerClass = new InnerClass(); //Compiler Error
```

To create an Inner Class you need an instance of Outer Class.

```
OuterClass example = new OuterClass();
OuterClass.InnerClass innerClass = example.new InnerClass();
```

Instance variables of Outer Class are available in inner class

Static Inner Nested Class

Creating Static Nested Class

Static Nested Class can be created without needing to create its parent. Without creating NestedClassesExample, we createdStaticNestedClass.

```
OuterClass.StaticNestedClass staticNestedClass1 = new OuterClass.StaticNestedClass();
```

Member variables are not static

Static Nested Class member variables are not static. They can have different values.

```
System.out.println(staticNestedClass1
.getStaticNestedClassVariable()); //5
System.out.println(staticNestedClass2
.getStaticNestedClassVariable()); //10
```

Outer class instance variables are not accessible.

Instance variables of outer class are not available in the Static Class.

Method Inner Class Example

Consider the example below: MethodLocalInnerClass is declared in exampleMethod();

```
class OuterClass {
   private int outerClassInstanceVariable;

   public void exampleMethod() {
int localVariable;
final int finalVariable = 5;
class MethodLocalInnerClass {
   public void method() {
//Can access class instance variables
System.out
.println(outerClassInstanceVariable);

//Cannot access method's non-final local variables
//localVariable = 5;//Compiler Error
System.out.println(finalVariable);//Final variable is fine..
   }
}
```

```
//MethodLocalInnerClass can be instantiated only in this method
MethodLocalInnerClass m1 = new MethodLocalInnerClass();
m1.method();
   }

   //MethodLocalInnerClass can be instantiated only in the method where it is declared
   //MethodLocalInnerClass m1 = new MethodLocalInnerClass();//COMPILER ERROR

}
```

Method inner class is not accessible outside the method

Method inner class can access class instance variables

Method inner class cannot access method's non-final local variables

Variable Arguments

- Variable Arguments allow calling a method with different number of parameters.

Variable Arguments Syntax

Data Type followed ... (three dot's) is syntax of a variable argument.

```
public int sum(int... numbers) {
```

Inside the method a variable argument is similar to an array. For Example: number can be treated in below method as if it is declared as int[] numbers;

```
   public int sum(int... numbers) {
int sum = 0;
for (int number: numbers) {
   sum += number;
}
return sum;
   }
```

Variable Argument: only Last Parameter

Variable Argument should be always the last parameter (or only parameter) of a method.

Exception Handling

Exception Handling Example 4: Try catch block

```
Let's add a try catch block in method2
   public static void main(String[] args) {
method1();
System.out.println("Line after Exception - Main");
   }

   private static void method1() {
method2();
System.out.println("Line after Exception - Method 1");
   }

   private static void method2() {
try {
   String str = null;
   str.toString();
   System.out.println("Line after Exception - Method 2");
} catch (Exception e) {
   // NOT PRINTING EXCEPTION TRACE- BAD PRACTICE
   System.out.println("Exception Handled - Method 2");
}
   }
```

Output

```
Exception Handled - Method 2
Line after Exception - Method 1
Line after Exception - Main
```

Since Exception Handling is added in the method method2, the exception did not propogate to method1. You can see the "Line after Exception - **" in the output for main, method1 since they are not affected by the exception thrown. Since the exception was handled in method2, method1 and main are not affected by it. This is the main essence of exception handling. However, note that the line after the line throwing exception in method2 is not executed.

Few important things to remember from this example. 1.If exception is handled, it does not propogate further. 2.In a try block, the lines after the line throwing the exception are not executed.

Exception Handling Example 6 - Finally

Finally block is used when code needs to be executed irrespective of whether an exception is thrown. Let us now move connection.close(); into a finally block. Also connection declaration is moved out of the try block to make it visible in the finally block.

finally is executed even if there is a return statement in catch or try

try without a catch is allowed

```
private static void method2() {
Connection connection = new Connection();
connection.open();
try {
   // LOGIC
   String str = null;
   str.toString();
} finally {
   connection.close();
}
   }
```

Try without both catch and finally is not allowed.

Exception Handling Hierarchy

Throwable is the highest level of Error Handling classes.

Below class definitions show the pre-defined exception hierarchy in Java.

```
//Pre-defined Java Classes
class Error extends Throwable{}
class Exception extends Throwable{}
class RuntimeException extends Exception{}
```

Errors

Error is used in situations when there is nothing a programmer can do about an error. Ex: StackOverflowError, OutOfMemoryError.

Exception

Exception is used when a programmer can handle the exception.

Un-Checked Exception

RuntimeException and classes that extend RuntimeException are called unchecked exceptions.

Checked Exception

Other Exception Classes (which don't fit the earlier definition). These are also called Checked Exceptions.

```
//Programmer defined classes
class CheckedException1 extends Exception{}
class CheckedException2 extends CheckedException1{}

class UnCheckedException extends RuntimeException{}
class UnCheckedException2 extends UnCheckedException{}
```

Throwing RuntimeException in method

Method addAmounts in Class AmountAdder adds amounts. If amounts are of different currencies it throws an exception.

```
class Amount {
   public Amount(String currency, int amount) {
this.currency = currency;
this.amount = amount;
   }

   String currency;// Should be an Enum
   int amount;// Should ideally use BigDecimal
}

// AmountAdder class has method addAmounts which is throwing a RuntimeException
class AmountAdder {
   static Amount addAmounts(Amount amount1, Amount amount2) {
if (!amount1.currency.equals(amount2.currency)) {
   throw new RuntimeException("Currencies don't match");
}
return new Amount(amount1.currency, amount1.amount + amount2.amount);
   }
}

public class ExceptionHandlingExample2 {

   public static void main(String[] args) {
AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR", 5));
   }

}
```

Output

```
Exception in thread "main" java.lang.RuntimeException: Currencies don't match
at
com.in28minutes.exceptionhandling.AmountAdder.addAmounts(ExceptionHandlingExample2.java:1
7)
at
com.in28minutes.exceptionhandling.ExceptionHandlingExample2.main(ExceptionHandlingExample2.
java:28)
Exception message shows the type of exception(java.lang.RuntimeException) and the string message
passed to the RuntimeException constructor("Currencies don't match");
```

Throwing Exception (Checked Exception) in method

Let us now try to change the method addAmounts to throw an Exception instead of RuntimeException. It gives us a compilation error.

```
class AmountAdder {
    static Amount addAmounts(Amount amount1, Amount amount2) {
if (!amount1.currency.equals(amount2.currency)) {
    throw new Exception("Currencies don't match");// COMPILER ERROR!// Unhandled
exception type Exception
}
return new Amount(amount1.currency, amount1.amount + amount2.amount);
    }
}
```

All classes that are not RuntimeException or subclasses of RuntimeException but extend Exception are called CheckedExceptions. The rule for CheckedExceptions is that they should be handled or thrown. Handled means it should be completed handled - i.e. not throw out of the method. Thrown means the method should declare that it throws the exception

Custom Defined Exception Classes

For the scenario above we can create a customized exception, CurrenciesDoNotMatchException. If we want to make it a Checked Exception, we can make it extend Exception class. Otherwise, we can extend RuntimeException class.

Extending Exception Class

```
class CurrenciesDoNotMatchException extends Exception{
}
```

No we can change the method addAmounts to throw CurrenciesDoNotMatchException - even the declaration of the method changed.

```
class AmountAdder {
   static Amount addAmounts(Amount amount1, Amount amount2)
   throws CurrenciesDoNotMatchException {
if (!amount1.currency.equals(amount2.currency)) {
   throw new CurrenciesDoNotMatchException();
}
return new Amount(amount1.currency, amount1.amount + amount2.amount);
   }
}
```

main method needs to be changed to catch: CurrenciesDoNotMatchException

```
public class ExceptionHandlingExample2 {
   public static void main(String[] args) {
try {
   AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR",
   5));
} catch (CurrenciesDoNotMatchException e) {
```

```
   System.out.println("Exception Handled in Main" + e.getClass());
}
   }
}
```

When should **Unchecked Exceptions** be used? Use **Unchecked** when there is no recovery. For example, when the memory of the server is overused.

RuntimeException is used for errors when your application can not recover. For example, **NullPointerException** and **ArrayOutOfBoundsException**. You can avoid a **RuntimeException** with an 'if' command. You should not handle or catch it.

Multiple Catch Blocks

We can have two catch blocks for a try. Order of Handling of exceptions: a. Same Class b. Super Class.

Specific Exceptions before Generic Exceptions

Specific Exception catch blocks should be before the catch block for a Generic Exception.

Exception Handling Best Practices

In all above examples we have not followed an Exception Handling good practice(s). Never Completely Hide Exceptions. At the least log them. printStactTrace method prints the entire stack trace when an exception occurs. If you handle an exception, it is always a good practice to log the trace.

Console

- Console is used to read input from keyboard and write output.

Getting a Console reference

```
//Console console = new Console(); //COMPILER ERROR
Console console = System.console();
```

Console utility methods

```
console.printf("Enter a Line of Text");
```

```
String text = console.readLine();
console.printf("Enter a Password");
```

Password doesn't show what is being entered

```
char[] password = console.readPassword();

console.format("\nEntered Text is %s", text);
```

Format or Printf

- Format or Printf functions help us in printing formatted output to the console.

Format/Printf Examples

Let's look at a few examples to quickly understand printf function.

```
System.out.printf("%d", 5);//5
System.out.printf("My name is %s", "Rithu");//My name is Rithu
System.out.printf("%s is %d Years old", "Rithu", 5);//Rithu is 5 Years old
```

In the simplest form, string to be formatted starts with % followed by conversion indicator => b - boolean c - char d - integer f - floating point s - string.

Other Format/Printf Examples

```
//Prints 12 using minimum 5 character spaces.
System.out.printf("|%5d|", 12); //prints |   12|

//Prints 1234 using minimum 5 character spaces.
System.out.printf("|%5d|", 1234); //prints | 1234|
//In above example 5 is called width specifier.

//Left Justification can be done by using -
System.out.printf("|%-5d|", 12); //prints |12   |

//Using 0 pads the number with 0's
System.out.printf("%05d", 12); //prints 00012

//Using , format the number using comma's
System.out.printf("%,d", 12345); //prints 12,345

//Using ( prints negative numbers between ( and )
System.out.printf("%(d", -12345); //prints (12345)
System.out.printf("%(d", 12345); //prints 12345

//Using + prints + or - before the number
System.out.printf("%+d", -12345); //prints -12345
System.out.printf("%+d", 12345); //prints +12345
```

String Buffer & String Builder

- StringBuffer and StringBuilder are used when you want to modify values of a string frequently. String Buffer class is thread safe where as String Builder is NOT thread safe.

  Le terme thread-safe signifie qu'une fonction peut être implémentée de telle sorte qu'elle puisse être exécutée par plusieurs threads s'exécutant en même temps.

String Buffer Examples

```
StringBuffer stringbuffer = new StringBuffer("12345");
stringbuffer.append("6789");
System.out.println(stringbuffer); //123456789
//All StringBuffer methods modify the value of the object.
```

String Builder Examples

```
StringBuilder sb = new StringBuilder("0123456789");

//StringBuilder delete(int startIndex, int endIndexPlusOne)
System.out.println(sb.delete(3, 7));//012789

StringBuilder sb1 = new StringBuilder("abcdefgh");

//StringBuilder insert(int indext, String whatToInsert)
System.out.println(sb1.insert(3, "ABCD"));//abcABCDdefgh

StringBuilder sb2 = new StringBuilder("abcdefgh");
//StringBuilder reverse()
System.out.println(sb2.reverse());//hgfedcba
```

Similar functions exist in StringBuffer also.

Method Chaining

All functions also return a reference to the object after modifying it.This allows a concept called method chaining.

```
StringBuilder sb3 = new StringBuilder("abcdefgh");
System.out.println(sb3.reverse().delete(5, 6).insert(3, "---"));//hgf---edba
```

Date

- Date is no longer the class Java recommends for storing and manipulating date and time. Most of methods in Date are deprecated. Use Calendar class instead. Date internally represents date-time as number of milliseconds (a long value) since 1st Jan 1970.

Calendar

- Calendar class is used in Java to manipulate Dates. Calendar class provides easy ways to add or reduce days, months or years from a date. It also provide lot of details about a date (which day of the year? Which week of the year? etc.)

Calendar is abstract

Calendar class cannot be created by using new Calendar. The best way to get an instance of Calendar class is by using getInstance() static method in Calendar.

```
//Calendar calendar = new Calendar(); //COMPILER ERROR
Calendar calendar = Calendar.getInstance();
```

Calendar set day, month and year

Setting day, month or year on a calendar object is simple. Call the set method with appropriate Constant for Day, Month or Year. Next parameter is the value.

```
calendar.set(Calendar.DATE, 24);
calendar.set(Calendar.MONTH, 8);//8 - September
calendar.set(Calendar.YEAR, 2010);
```

dar get method

Let's get information about a particular date - 24th September 2010. We use the calendar get method. The parameter passed indicates what value we would want to get from the calendar , day or month or year or .. Few examples of the values you can obtain from a calendar are listed below.

```
System.out.println(calendar.get(Calendar.YEAR));//2010
System.out.println(calendar.get(Calendar.MONTH));//8
System.out.println(calendar.get(Calendar.DATE));//24
System.out.println(calendar.get(Calendar.WEEK_OF_MONTH));//4
System.out.println(calendar.get(Calendar.WEEK_OF_YEAR));//39
System.out.println(calendar.get(Calendar.DAY_OF_YEAR));//267
System.out.println(calendar.getFirstDayOfWeek());//1 -> Calendar.SUNDAY
```

Calendar - Modify a Date

We can use the calendar add and roll methods to modify a date. Calendar add method can be used to find a date 5 days or 5 months before the date by passing a ,5 i.e. a negative 5.

```
calendar.add(Calendar.DATE, 5);
System.out.println(calendar.getTime());//Wed Sep 29 2010
calendar.add(Calendar.MONTH, 1);
System.out.println(calendar.getTime());//Fri Oct 29 2010
```

```
calendar.add(Calendar.YEAR, 2);
System.out.println(calendar.getTime());//Mon Oct 29 2012
```

Roll method

Roll method will only the change the value being modified. YEAR remains unaffected when MONTH is changed, for instance.

```
calendar.roll(Calendar.MONTH, 5);
System.out.println(calendar.getTime());//Mon Mar 29 2012
```

Collection Interfaces

- Arrays are not dynamic. Once an array of a particular size is declared, the size cannot be modified. To add a new element to the array, a new array has to be created with bigger size and all the elements from the old array copied to new array. Collections are used in situations where data is dynamic. Collections allow adding an element, deleting an element and host of other operations. There are a number of Collections in Java allowing to choose the right Collection for the right context. Before looking into Collection classes, let's take a quick look at all the important collection interfaces and the operations they allow.

Collection Interface

```
interface Collection<E> extends Iterable<E>
{
  boolean add(E paramE);
  boolean remove(Object paramObject);

  int size();
  boolean isEmpty();
  void clear();

  boolean contains(Object paramObject);
  boolean containsAll(Collection<?> paramCollection);

  boolean addAll(Collection<? extends E> paramCollection);
  boolean removeAll(Collection<?> paramCollection);
  boolean retainAll(Collection<?> paramCollection);


  Iterator<E> iterator();

  //A NUMBER OF OTHER METHODS AS WELL..
}
```

List Interface

List interface extends Collection interface. So, it contains all methods defined in the Collection interface. In addition, List interface allows operation specifying the position of the element in the Collection. Any implementation of the List interface would maintain the insertion order. When a new element is inserted, it is inserted at the end of the list of elements. We can also use the void add(int paramInt, E paramE); method to insert an element at a specific position. We can also set and get the elements at a particular index in the list using corresponding methods.

Other important methods are listed below:

```
interface List<E> extends Collection<E>
{
  boolean addAll(int paramInt, Collection<? extends E> paramCollection);

  E get(int paramInt);
  E set(int paramInt, E paramE);

  void add(int paramInt, E paramE);
  E remove(int paramInt);

  int indexOf(Object paramObject);
  int lastIndexOf(Object paramObject);

  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int paramInt);
  List<E> subList(int paramInt1, int paramInt2);
}
```

Map Interface

First and foremost, Map interface does not extend Collection interface. So, it does not inherit any of the methods from the Collection interface. A Map interface supports Collections that use a key value pair. A key-value pair is a set of linked data items: a key, which is a unique identifier for some item of data, and the value, which is either the data or a pointer to the data. Key-value pairs are used in lookup tables, hash tables and configuration files. A key value pair in a Map interface is called an Entry. Put method allows to add a key, value pair to the Map.

```
  V put(K paramK, V paramV);
```

Get method allows to get a value from the Map based on the key.

```
  V get(Object paramObject);
```

Other important methods are shown below:

```
interface Map<K, V>
{
  int size();
```

```
 boolean isEmpty();

 boolean containsKey(Object paramObject);
 boolean containsValue(Object paramObject);

 V get(Object paramObject);
 V put(K paramK, V paramV);
 V remove(Object paramObject);

 void putAll(Map<? extends K, ? extends V> paramMap);
 void clear();

 Set<K> keySet();
 Collection<V> values();
 Set<Entry<K, V>> entrySet();

 boolean equals(Object paramObject);
 int hashCode();

 public static abstract interface Entry<K, V>
 {
   K getKey();
   V getValue();
   V setValue(V paramV);
   boolean equals(Object paramObject);
   int hashCode();
 }
}
```

Set Interface

Set Interface extends Collection Interface. Set interface only contains the methods from the Collection interface with added restriction that it cannot contain duplicates.

```
// Unique things only - Does not allow duplication.
// If obj1.equals(obj2) then only one of them can be in the Set.
interface Set<E> extends Collection<E>
{
}
```

SortedSet Interface

SortedSet Interface extends the Set Interface. So, it does not allow duplicates. In addition, an implementation of SortedSet interface maintains its elements in a sorted order. It adds operations that allow getting a range of values (subSet, headSet, tailSet).
Important Operations listed below:

```
public interface SortedSet<E> extends Set<E> {
```

```
   SortedSet<E> subSet(E fromElement, E toElement);
   SortedSet<E> headSet(E toElement);
   SortedSet<E> tailSet(E fromElement);

   E first();
   E last();

   Comparator<? super E> comparator();
}
```

SortedMap Interface

SortedMap interface extends the Map interface. In addition, an implementation of SortedMap interface maintains keys in a sorted order. Methods are available in the interface to get a ranges of values based on their keys.

```
public interface SortedMap<K, V> extends Map<K, V> {
   Comparator<? super K> comparator();

   SortedMap<K, V> subMap(K fromKey, K toKey);

   SortedMap<K, V> headMap(K toKey);

   SortedMap<K, V> tailMap(K fromKey);

   K firstKey();

   K lastKey();
}
```

Queue Interface

Queue Interface extends Collection interface. Queue Interface is typically used for implementation holding elements in order for some processing.
Queue interface offers methods peek() and poll() which get the element at head of the queue. The difference is that poll() method removes the head from queue also. peek() would keep head of the queue unchanged.

```
interface Queue<E> extends Collection<E>
{
  boolean offer(E paramE);
  E remove();
  E poll();
  E element();
  E peek();
}
```

Iterator interface

Iterator interface enables us to iterate (loop around) a collection. All collections define a method iterator() that gets an iterator of the collection. hasNext() checks if there is another element in the collection being iterated. next() gets the next element.

```
public interface Iterator<E> {
   boolean hasNext();

   E next();
}
```

Collections

- Collections can only hold Objects - not primitives.

ArrayList

ArrayList implements the list interface. So, ArrayList stores the elements in insertion order (by default). Element's can be inserted into and removed from ArrayList based on their position. Let's look at how to instantiate an ArrayList of integers.

```
List<String> arraylist = new ArrayList<String>();

//adds at the end of list
arraylist.add("Sachin");//[Sachin]

//adds at the end of list
arraylist.add("Dravid");//[Sachin, Dravid]

//adds at the index 0
arraylist.add(0, "Ganguly");//[Ganguly, Sachin, Dravid]

//List allows duplicates - Sachin is present in the list twice
arraylist.add("Sachin");//[ Ganguly, Sachin, Dravid, Sachin]

System.out.println(arraylist.size());//4
System.out.println(arraylist.contains("Dravid"));//true
```

Iterating around a list

```
Iterator<String> arraylistIterator = arraylist
.iterator();
while (arraylistIterator.hasNext()) {
   String str = arraylistIterator.next();
   System.out.println(str);//Prints the 4 names in the list on separate lines.
}
```

Other ArrayList (List) methods

indexOf() function - returns index of element if element is found. Negative number otherwise.

get() function - get value at specified index.
remove() function has two variations.

Sorting Collections

```
//Strings - By Default - are sorted alphabetically
Collections.sort(numbers);
```

We get a compiler error if we compare objects from a class that does not implement Comparable interface. We were able to sort numbers in earlier example because String class implements Comparable.

```
class Cricketer implements Comparable<Cricketer> {
    //OTHER CODE/PROGRAM same as previous

    //compareTo takes an argument of the same type of the class
    //compareTo returns -1 if this < that
    //   1 if this > that
    //   0 if this = that
    @Override
    public int compareTo(Cricketer that) {
```

Convert List to Array

There are two ways. First is to use toArray(String) function. Example below. This creates an array of String's

```
String[] numbers1Array = new String[numbers1.size()];
numbers1Array = numbers1.toArray(numbers1Array);
System.out.println(Arrays.toString(numbers1Array));
//prints [one, two, three, four]
```

Other is to use toArray() function. Example below. This creates an array of Objects.

```
Object[] numbers1ObjArray = numbers1.toArray();
System.out.println(Arrays
.toString(numbers1ObjArray));
//[one, two, three, four]
```

Convert Array to List

```
String values[] = { "value1", "value2", "value3" };
List<String> valuesList = Arrays.asList(values);
System.out.println(valuesList);//[value1, value2, value3]
```

List vs ArrayList

List	ArrayList
List is an interface	ArrayList is a class
List interface extends the Collection framework	ArrayList extends AbstractList class and implements List interface
List cannot be instantiated.	ArrayList can be instantiated.

Other List interface implementations

Other classes that implement List interface are Vector and LinkedList.

Vector

Vector has the same operations as an ArrayList. However, all methods in Vector are synchronized. So, we can use Vector if we share a list between two threads and we would want to them synchronized.

Thread-safe vs synchronized ? Synchronized: only one thread can operate at same time. Threadsafe: a method or class instance can be used by multiple threads at the same time without any problems occurring. synchronized is one basic method of achieving ThreadSafe code.

LinkedList

Linked List extends List and Queue.Other than operations exposed by the Queue interface, LinkedList has the same operations as an ArrayList. However, the underlying implementation of Linked List is different from that of an ArrayList. ArrayList uses an Array kind of structure to store elements. So, inserting and deleting from an ArrayList are expensive operations. However, search of an ArrayList is faster than LinkedList. LinkedList uses a linked representation. Each object holds a link to the next element. Hence, insertion and deletion are faster than ArrayList. But searching is slower.

Set Interface

- HashSet, LinkedHashSet and TreeSet implement the Set interface.

HashSet

HashSet implements set interface. Sets do not allow duplicates. HashSet does not support ordering.

LinkedHashSet
LinkedHashSet implements set interface and exposes similar operations to a HashSet. Difference is that LinkedHashSet maintains insertion order. When we iterate a LinkedHashSet, we would get the elements back in the order in which they were inserted.

TreeSet
TreeSet implements Set, SortedSet and NavigableSet interfaces.TreeSet is similar to HashSet except that it stores element's in Sorted Order.

Map Interface

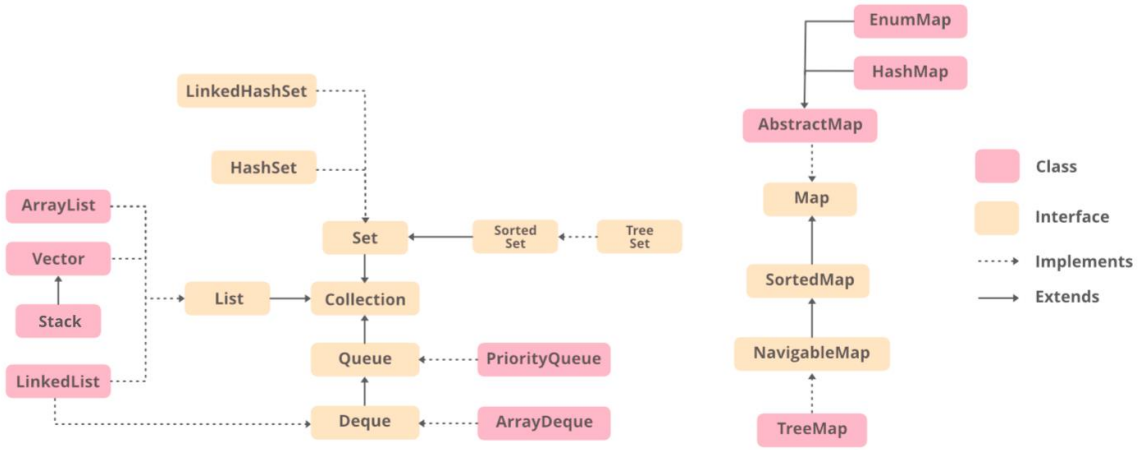
- Let's take a look at different implementations of the Map interface.

HashMap
HashMap implements Map interface , there by supporting key value pairs.

TreeMap
TreeMap is similar to HashMap except that it stores keys in sorted order. It implements NavigableMap interface and SortedMap interfaces along with the Map interface.

PriorityQueue

- PriorityQueue implements the Queue interface.



Generics

- Generics are used to create Generic Classes and Generic methods which can work with different Types(Classes).

```
class MyListGeneric<T> {
    private List<T> values;

    void add(T value) {
values.add(value);
    }

    void remove(T value) {
values.remove(value);
    }

    T get(int index) {
return values.get(index);
    }
}
```

Instead of T, We can use any valid identifier If a generic is declared as part of class declaration, it can be used any where a type can be used in a class - method (return type or argument), member variable etc.

```
MyListGeneric<String> myListString = new MyListGeneric<String>();
```

Generics Restrictions

In MyListGeneric, Type T is defined as part of class declaration. Any Java Type can be used a type for this class. If we would want to restrict the types allowed for a Generic Type, we can use a Generic Restrictions. Consider the example class below:

```
class MyListRestricted<T extends Number> {
    private List<T> values;
```

Generic Method Example

A generic type can be declared as part of method declaration as well. Then the generic type can be used anywhere in the method (return type, parameter type, local or block variable type). Consider the method below:

```
    static <X extends Number> X doSomething(X number){
X result = number;
//do something with result
return result;
    }
```

The method can now be called with any Class type extend Number.

```
Integer i = 5;
Integer k = doSomething(i);
```

Files

File Class

Create a File Object

```
File file = new File("FileName.txt");
```

File basic Methods

Check if the file exists.

```
System.out.println(file.exists());
```
if file does not exist creates it and returns true. If file exists, returns false.

```
System.out.println(file.createNewFile());
```
Getting full path of file.
```
System.out.println(file.getAbsolutePath());
```
A File class in Java represents a file and directory.
```
System.out.println(file.isFile());//true
System.out.println(file.isDirectory());//false
```
Renaming a file

```
File fileWithNewName = new File("NewFileName.txt");
file.renameTo(fileWithNewName);
//There is no method file.renameTo("NewFileName.txt");
```

Creating a directory

```
File newDirectory = new File("newfolder");
System.out.println(newDirectory.mkdir());//true - First Time
```

Read and write from a File

Implementations of Writer and Reader abstract classes help us to write and read (content of) files. Writer methods - flush, close, append (text) Reader methods - read, close (NO FLUSH) Writer implementations - FileWriter,BufferedWriter,PrintWriter Reader implementations - FileReader,BufferedReader

FileWriter and FileReader

- FileWriter and FileReader provide basic file writing and reading operations.

Write a string to a file using FileWriter

```
//FileWriter helps to write stuff into the file
FileWriter fileWriter = new FileWriter(file);
fileWriter.write("How are you doing?");
//Always flush before close. Writing to file uses Buffering.
fileWriter.flush();
fileWriter.close();
```

FileWriter Constructors can accept file(File) or the path to file (String) as argument. When a writer object is created, it creates the file - if it does not exist.

Read from file using FileReader

```
FileReader fileReader = new FileReader(file);
char[] temp = new char[25];

//fileReader reads entire file and stores it into temp
System.out.println(fileReader.read(temp));//18 - No of characters Read from
file

System.out.println(Arrays.toString(temp));//output below
//[H, o, w,  , a, r, e,  , y, o, u,  , d, o, i, n, g, ?, , , , , ,]

fileReader.close();//Always close anything you opened:)
```

FileReader constructors can accept file(File) or the path to file (String) as argument.

BufferedWriter and BufferedReader

- BufferedWriter and BufferedReader provide better buffering in addition to basic file writing and reading operations. For example, instead of reading the entire file, we can read a file line by line.

BufferedWriter Constructors only accept another Writer as argument

```
FileWriter fileWriter3 = new FileWriter("BufferedFileName.txt");
BufferedWriter bufferedWriter = new BufferedWriter(fileWriter3);
```

Using BufferedWriter class

```
bufferedWriter.write("How are you doing Buddy?");
bufferedWriter.newLine();
bufferedWriter.write("I'm Doing Fine");
//Always flush before close. Writing to file uses Buffering.
bufferedWriter.flush();
bufferedWriter.close();
fileWriter3.close();
```

BufferedReader Class

BufferedReader helps to read the file line by line

BufferedReader Constructors only accept another Reader as argument.

```
FileReader fileReader3 = new FileReader("BufferedFileName.txt");
BufferedReader bufferedReader = new BufferedReader(fileReader3);
```

BufferedReader , Reading a file

```
String line;
//readLine returns null when reading the file is completed.
while((line=bufferedReader.readLine()) != null){
    System.out.println(line);
}
```

PrintWriter

- PrintWriter provides advanced methods to write formatted text to the file. It supports printf function.

PrintWriter constructors supports varied kinds of arguments , File, String (File Path) and Writer.

```
PrintWriter printWriter = new PrintWriter("PrintWriterFileName.txt");
```

PrintWriter , Write to a file

Other than write function you can use format, printf, print, println functions to write to PrintWriter file.

```
//writes "My Name" to the file
printWriter.format("%15s", "My Name");

printWriter.println(); //New Line
printWriter.println("Some Text");

//writes "Formatted Number: 4.50000" to the file
printWriter.printf("Formatted Number: %5.5f", 4.5);
printWriter.flush();//Always flush a writer
printWriter.close();
```

Serialization

- Serialization helps us to save and retrieve the state of an object.

Serialization and De-Serialization - Important methods

Serialization => Convert object state to some internal object representation. De-Serialization => The reverse. Convert internal representation to object.

Two important methods 1.ObjectOutputStream.writeObject() // serialize and write to file 2.ObjectInputStream.readObject() // read from file and deserialize

Implementing Serializable Interface

To serialize an object it should implement Serializable interface. In the example below, Rectangle class implements Serializable interface. Note that Serializable interface does not declare any methods to be implemented.

Serializing an object - Example

```
FileOutputStream fileStream = new FileOutputStream("Rectangle.ser");
ObjectOutputStream objectStream = new ObjectOutputStream(fileStream);
objectStream.writeObject(new Rectangle(5, 6));
objectStream.close();
```

De-serializing an object , Example

```
FileInputStream fileInputStream = new FileInputStream("Rectangle.ser");
ObjectInputStream objectInputStream = new ObjectInputStream(
fileInputStream);
Rectangle rectangle = (Rectangle) objectInputStream.readObject();
objectInputStream.close();
```

Serialization , Transient variables

Area in the previous example is a calculated value. It is unnecessary to serialize and deserialize. We can calculate it when needed. In this situation, we can make the variable transient. Transient variables are not serialized. (transient int area;)

```
transient int area;
```

Serialization , readObject method

We need to recalculate the area when the rectangle object is deserialized. This can be achieved by adding readObject method to Rectangle class. In addition to whatever java does usually while deserializing, we can add custom code for the object

Serialization , writeObject method

We can also write custom code when serializing the object by adding the writeObject method to Rectange class. writeObject method accepts an ObjectOutputStream as input parameter. To the writeObject method we can add the custom code that we want to run during Serialization.

Serializing an Object chain

Objects of one class might contain objects of other classes. When serializing and de-serializing, we might need to serialize and de-serialize entire object chain.

Serialization and Initialization

When a class is Serialized, initialization (constructor's, initializer's) does not take place. The state of the object is retained as it is.

Serialization and inheritance

However in the case of inheritance (a sub class and super class relationship), interesting things happen. When subclass is serializable and superclass is not, the state of subclass variables is retained. However, for the super class, initialization (constructors and initializers) happens again.

Serialization and Static Variables

Static Variables are not part of the object. They are not serialized.

Threads

- Threads allow Java code to run in parallel.

Creating a Thread Class

Creating a Thread class in Java can be done in two ways. Extending Thread class and implementing Runnable interface.

Creating a Thread By Extending Thread class

an be created by extending Thread class and implementing the public void run() method. Look at the example below: A dummy implementation for BattingStatistics is provided which counts from 1 to 1000.

```
class BattingStatisticsThread extends Thread {
    //run method without parameters
    public void run() {
for (int i = 0; i < 1000; i++)
    System.out
    .println("Running Batting Statistics Thread "
    + i);
    }
}
```

Creating a Thread by Implementing Runnable interface

Thread class can also be created by implementing Runnable interface and implementing the method declared in Runnable interface Òpublic void run()Ó. Example below shows the Batting Statistics Thread implemented by implementing Runnable interface.

```
class BowlingStatisticsThread implements Runnable {
    //run method without parameters
    public void run() {
for (int i = 0; i < 1000; i++)
    System.out
    .println("Running Bowling Statistics Thread "
    + i);
    }
}
```

Running a Thread

Running a Thread in Java is slightly different based on the approach used to create the thread.

Thread created Extending Thread class

When using inheritance, An object of the thread needs be created and start() method on the thread needs to be called. Remember that the method that needs to be called is not run() but it is start().

```
BattingStatisticsThread battingThread1 = new BattingStatisticsThread();
battingThread1.start();
```

Three steps involved.

1. Create an object of the BowlingStatisticsThread(class implementing Runnable).
2. Create a Thread object with the earlier object as constructor argument.
3. Call the start method on the thread.

```
BowlingStatisticsThread battingInterfaceImpl = new BowlingStatisticsThread();
Thread battingThread2 = new Thread(
battingInterfaceImpl);
battingThread2.start();
```

Thread Example , Complete Program

Let's consider the complete example using all the snippets of code created above.

```
public class ThreadExamples {
    public static void main(String[] args) {
class BattingStatisticsThread extends Thread {
    // run method without parameters
    public void run() {
```

```
for (int i = 0; i < 1000; i++)
    System.out.println("Running Batting Statistics Thread " + i);
    }
}

class BowlingStatisticsThread implements Runnable {
    // run method without parameters
    public void run() {
for (int i = 0; i < 1000; i++)
    System.out.println("Running Bowling Statistics Thread " + i);
    }
}

BattingStatisticsThread battingThread1 = new BattingStatisticsThread();
battingThread1.start();

BowlingStatisticsThread battingInterfaceImpl = new
BowlingStatisticsThread();
Thread battingThread2 = new Thread(battingInterfaceImpl);
battingThread2.start();

    }

}
Output:
Running Batting Statistics Thread 0
Running Batting Statistics Thread 1
..
..
Running Batting Statistics Thread 10
Running Bowling Statistics Thread 0
..
..
Running Bowling Statistics Thread 948
Running Bowling Statistics Thread 949
Running Batting Statistics Thread 11
Running Batting Statistics Thread 12
..
..
Running Batting Statistics Thread 383
Running Batting Statistics Thread 384
Running Bowling Statistics Thread 950
Running Bowling Statistics Thread 951
..
Running Bowling Statistics Thread 998
Running Bowling Statistics Thread 999
Running Batting Statistics Thread 385
..
Running Batting Statistics Thread 998
Running Batting Statistics Thread 999
```

Discussion about Thread Example

Above output shows sample execution of the thread. The output will not be the same with every run. We can notice that Batting Statistics Thread and the Bowling Statistics Threads are alternating in execution. Batting Statistics Thread runs upto 10, then Bowling Statistics Thread runs upto 949, Batting Statistics Thread picks up next and

runs up to 384 and so on. There is no usual set pattern when Threads run (especially when they have same priority , more about this later..). JVM decides which Thread to run at which time. If a Thread is waiting for user input or a network connection, JVM runs the other waiting Threads.

Thread Synchronization

Since Threads run in parallel, a new problem arises. i.e. What if thread1 modifies data which is being accessed by thread2? How do we ensure that different threads don't leave the system in an inconsistent state? This problem is usually called Thread Synchronization Problem. Example: two threads executing the same method on the same object and changing the values of the object attributes at the same time → the values set by the two threads will interfere with each other leading to unexpected results.

The way you can prevent multiple threads from executing the same method is by using the synchronized keyword on the method. If a method is marked synchronized, a different thread gets access to the method only when there is no other thread currently executing the method. Let's mark the method as synchronized:

```
synchronized void synchronizedExample1() {
//All code goes here..
    }
```

Threads & Synchronized Keyword

A method or part of the method can be marked as synchronized. JVM will ensure that there is only thread running the synchronized part of code at any time. However, thread synchronization is not without consequences. There would be a performance impact as the rest of threads wait for the current thread executing a synchronized block. So, as little code as possible should be marked as synchronized.

Synchronized block Example

All code which goes into the block is synchronized on the current object.

```
    void synchronizedExample2() {
synchronized (this){
//All code goes here..
}
    }
```

Synchronized static method Example

```
    synchronized static int getCount(){
return count;
    }
```

States of a Thread

Different states that a thread can be in are defined the class State.

- NEW;
- RUNNABLE;
- RUNNING;
- BLOCKED/WAITING;
- TERMINATED/DEAD;

A thread is in BLOCKED/WAITING/SLEEPING state when it is not eligible to be run by the Scheduler. Thread is alive but is waiting for something. An example can be a Synchronized block. If Thread1 enters synchronized block, it blocks all the other threads from entering synchronized code on the same instance or class. All other threads are said to be in Blocked state. A thread is in DEAD/TERMINATED state when it has completed its execution. Once a thread enters dead state, it cannot be made active again.

Thread Priority

Scheduler can be requested to allot more CPU to a thread by increasing the threads priority. Each thread in Java is assigned a default Priority 5. This priority can be increased or decreased (Range 1 to 10). If two threads are waiting, the scheduler picks the thread with highest priority to be run. If all threads have equal priority, the scheduler then picks one of them randomly. Design programs so that they don't depend on priority.

Priority of thread can be changed by invoking setPriority method on the thread.

```
ThreadExample thread1 = new ThreadExample();
thread1.setPriority(8);
```

Thread Join method

Join method is an instance method on the Thread class. Let's see a small example to understand what join method does. Let's consider the thread's declared below: thread2, thread3, thread4

```
ThreadExample thread2 = new ThreadExample();
ThreadExample thread3 = new ThreadExample();
ThreadExample thread4 = new ThreadExample();
```

Let's say we would want to run thread2 and thread3 in parallel but thread4 can only run when thread3 is finished. This can be achieved using join method.

Join method example

```
Thread3.start();
```

```
thread2.start();
thread3.join();//wait for thread 3 to complete
System.out.println("Thread3 is completed.");
thread4.start();
```

Overloaded Join method

Join method also has an overloaded method accepting time in milliseconds as a parameter.

```
Thread4.join(2000);
```

Thread , Static methods

Thread yield method

Yield is a static method in the Thread class. It is like a thread saying " I have enough time in the limelight. Can some other thread run next?". A call to yield method changes the state of thread from RUNNING to RUNNABLE. However, the scheduler might pick up the same thread to run again, especially if it is the thread with highest priority. Summary is yield method is a request from a thread to go to Runnable state. However, the scheduler can immediately put the thread back to RUNNING state.

Thread sleep method

sleep is a static method in Thread class. sleep method can throw a InterruptedException. sleep method causes the thread in execution to go to sleep for specified number of milliseconds.

Thread and Deadlocks

Let's consider a situation where thread1 is waiting for thread2 (thread1 needs an object whose synchronized code is being executed by thread1) and thread2 is waiting for thread1. This situation is called a Deadlock. In a Deadlock situation, both these threads would wait for one another for ever.

Thread - wait, notify and notifyAll methods

If we want to wait only for a part of run method to be done instead of waiting for the entire run method to be finished (with join) we can use wait and notify methods (wait and notify methods can only be used in a synchronized context).

Wait method example

Below snippet shows how wait is used in earlier program. wait method is defined in the Object class. This causes the thread to wait until it is notified.

```
synchronized(thread){ → inside the main method
    thread.start();
    thread.wait();
}
```

Notify method example

Below snippet shows how notify is used in earlier program. notify method is defined in the Object class. This causes the object to notify other waiting threads.

```
synchronized (this) { → inside the run method
calculateSumUptoMillion();
notify();
    }
```

A combination of wait and notify methods make the main method to wait until the sum of million is calculated. However, not the main method does not wait for Sum of Ten Million to be calculated.

notifyAll method

If more than one thread is waiting for an object, we can notify all the threads by using notifyAll method.

```
Thread.notifyAll();
```

Assert

- Assertions are introduced in Java 1.4. They enable you to validate assumptions. If an assert fails (i.e. returns false), AssertionError is thrown (if assertions are enabled). assert is a keyword in java since 1.4. Before 1.4, assert can be used as identifier.

Basic assert is shown in the example below

```
private int computerSimpleInterest(int principal,float interest,int years){
    assert(principal>0);
    return 100;
}
```

Garbage Collection

- Garbage Collection is a name given to automatic memory management in Java. Aim of Garbage Collection is to Keep as much of heap available (free) for the program as possible. JVM removes objects on the heap which no longer have references from the heap.

When is Garbage Collection run?

Garbage Collection runs at the whims and fancies of the JVM (it isn't as bad as that). Possible situations when Garbage Collection might run are 1.when available memory on the heap is low 2.when cpu is free

Garbage Collection , Important Points

Programmatically, we can request (remember it's just a request - Not an order) JVM to run Garbage Collection by calling System.gc() method.

JVM might throw an OutOfMemoryException when memory is full and no objects on the heap are eligible for garbage collection.

finalize() method on the objected is run before the object is removed from the heap from the garbage collector. We recommend not to write any code in finalize();

Garbage Collection , Important Points

Programmatically, we can request (remember it's just a request - Not an order) JVM to run Garbage Collection by calling System.gc() method.

JVM might throw an OutOfMemoryException when memory is full and no objects on the heap are eligible for garbage collection.

finalize() method on the objected is run before the object is removed from the heap from the garbage collector. We recommend not to write any code in finalize();

Static Initializer

```
public class InitializerExamples {
    static int count;
    int i;

    static{
//This is a static initializers. Run only when Class is first loaded.
//Only static variables can be accessed
System.out.println("Static Initializer");
//i = 6;//COMPILER ERROR
System.out.println("Count when Static Initializer is run is " + count);
    }

    public static void main(String[] args) {
InitializerExamples example = new InitializerExamples();
InitializerExamples example2 = new InitializerExamples();
InitializerExamples example3 = new InitializerExamples();
    }
}
```

Code within static{ and } is called a static initializer. This is run only when class is first loaded. Only static variables can be accessed in a static initializer.

Instance Initializer Block

Let's look at an example

```
public class InitializerExamples {
    static int count;
    int i;
    {
//This is an instance initializers. Run every time an object is created.
//static and instance variables can be accessed
System.out.println("Instance Initializer");
i = 6;
count = count + 1;
System.out.println("Count when Instance Initializer is run is " + count);
    }

    public static void main(String[] args) {
InitializerExamples example = new InitializerExamples();
InitializerExamples example1 = new InitializerExamples();
InitializerExamples example2 = new InitializerExamples();
    }

}
```

Regular Expressions

- Regular Expressions make parsing, scanning and splitting a string very easy. We will first look at how you can evaluate a regular expressions in Java , using Patter, Matcher and Scanner classes. We will then look into how to write a regular expression.

Regular Expression in Java , Matcher and Pattern Example

Code below shows how to execute a regular expression in java.

```
private static void regex(String regex, String string) {
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(string);
    List<String> matches = new ArrayList<String>();
    while (m.find()) {
matches.add(m.start() + "<" + m.group() + ">");
    }
    System.out.println(matches);
}
```

Matcher class

Matcher class has the following utility methods: find(), start(), group(). find() method returns true until there is a match for the regular expression in the string. start() method gives the starting index of the match. group() method returns the matching part of the string.

Simple Regular Expressions

Search for 12 in the string

```
regex("12", "122345612");//[0<12>, 7<12>]
```

Certain characters escaped by \ have special meaning in regular expressions. For example, /s matches a whitespace character. Remember that to represent \ in a string, we should prepend \ to it. Let us see a few examples below.

```
System.out.println("\\");//prints \ (only one slash)
```

Space character - \s

```
regex("\\s", "12 1234 123 ");//[2< >, 7< >, 11< >]
```

Examples

Let's run this method with a regular expression to search for Ò12Ó in the string Ò122345612Ó.

```
regex("12", "122345612");
```

Output

```
[0<12>, 7<12>]
```

Digit - \d

```
regex("\\d", "12 12");//[0<1>, 1<2>, 3<1>, 4<2>]
```

Word character (letter, digits or underscore) - \w

```
regex("\\w", "ab 12 _");//[0<a>, 1<b>, 3<1>, 4<2>, 6<_>]
```

Square brackets are used in regular expressions to search for a range of characters. Few examples below. look for a,b,c,d,1,2,3,4 =>Note that this does not look for capital A,B,C,D

```
regex("[a-d1-4]", "azbkdm 15AB");//[0<a>, 2<b>, 4<d>, 7<1>]
regex("[a-dA-D]", "abyzCD");//[0<a>, 1<b>, 4<C>, 5<D>]
```

Regular Expressions , Multiple Characters

- is used in regular expression to look for 1 or more characters. For example a+ looks for 1 or more character a's.

```
regex("a+", "aaabaayza");//[0<aaa>, 4<aa>, 8<a>]
```

* - 0 or more repetitions.

Below expression looks for 1 or more a's followed by 0 or more b's followed by 1 or more c's. abc => match. ac=> match (since we used * for b). ab => does not match.

```
regex("a+b*c+", "abcdacdabdbc");//[0<abc>, 4<ac>]
```

? - 0 or 1 repetitions.

^a looks for anything other than a

[^abcd]+a looks for anything which is not a or b or c or d, repeated 0 or more times, followed by a

. matches any character

Greedy Regular Expressions

a+ matches a, aa,aaa,aaaa, aaaaa. If you look at the output of the below expression, it matches the biggest only aaaaa. This is called greedy behaviour. similar behavior is shown by *.

```
regex("a+", "aaaaab");//[0<aaaaa>]
```

You can make + reluctant (look for smallest match) by appending ?

```
regex("a+?", "aaaaab");//[0<a>, 1<a>, 2<a>, 3<a>, 4<a>]
```

Similarly *? is reluctant match for the greedy * If you want to look for characters . or * in a regular expression, then you should escape them. Example: If I want to look for ...(3 dots), we should use ... To represent ... as string we should put two 's instead of 1.

```
regex("\\.\\.\\.", "...a....b...c");//[0<...>, 4<...>, 9<...>]
```

Regular Expression using Scanner class

Below code shows how Scanner class can be used to execute regular expression. findInLine method in Scanner returns the match , if a match is found. Otherwise, it returns null.

```
private static void regexUsingScanner(String regex,
String string) {
    Scanner s = new Scanner(string);
    List<String> matches = new ArrayList<String>();

    String token;
    while ((token = s.findInLine(regex)) != null) {
matches.add(token);
    }
    ;
    System.out.println(matches);
}
```

Tokenizing

- Tokenizing means splitting a string into several sub strings based on delimiters. For example, delimiter ; splits the string ac;bd;def;e into four sub strings ac, bd, def and e. Delimiter can in itself be any of the regular expression(s) we looked at earlier. String.split(regex) function takes regex as an argument.