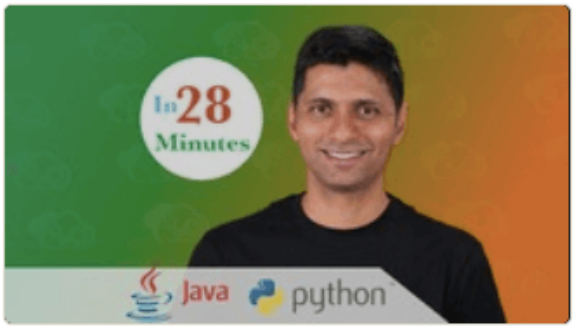
Java Tutorial For Beginners - A Cheat Sheet

Review Java 9 Concepts at Jet Speed.

Complete Java Course



Python for Beginners - Go from Java to Python in 100 Steps

Learn Python Programming using Your Java Skills. For Beginner Python Programmers.

in28Minutes Official

4.5 ★★★★½ (531)

8 total hours • 103 lectures • All Levels

Learn Java Functional Programming with Lambdas & Streams

Learn Java Functional Programming with Lambdas & Streams. Solve Java Functional Programming Puzzles & Exercises.

in28Minutes Official

4.5 ★★★★½ (499)

4.5 total hours • 44 lectures • All Levels

Bestseller

Introduction

Background

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.

First Java Program

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }

}
```

Notes

- Every line of code we write in Java is part of something called Class. We will talk about Class later.
- First line defines a public class called HelloWorld. All the code in a class is between { and }.
- When a program runs, Java should know which line of code has to be run first. public static void main(String[] args) is the first method that is run when a program is executed.

> Java, like any other programming language, is particular about syntax!!

Using Java and JavaC

There are two steps involved in running a Java Program

- Compilation
- Execution

Compilation

We use javac to compile java code.

- Open CommandPrompt/Terminal and cd to the folder where HelloWorld.java file is present
- execute the command below

```
javac HelloWorld.java
```

- You should see two files HelloWorld.java and HelloWorld.class in the folder.
- HelloWorld.class contains the java bytecode

Execution

- Now we can run the program using JVM
- execute the command below

```
java HelloWorld
```

- You should see the output "Hello World" printed in the console.

Class and Object

- What is a class?
- Definining an instance of a class - an object
- Invoking a method on the object

Variables

- Value of a variable changes during the course of a program execution.

```
int number;
number = 5;
System.out.println(number);//5
number = number + 2;
System.out.println(number);//7
number = number + 2;
System.out.println(number);//9
```

Declaring and Initializing Variables

- Declaration is give a variable a name and type

```
TYPE variableName;
```

Tips

- Two or more variables of single type can be declared together.
- Variable can be local or global. The local variables can be referenced (ie, are valid) only within the scope of their method (or function).
- All six numeric types in Java are signed.

Primitive Variables

Variables that store value.

Java defines few types like int (numbers), float(floating point numbers), char (characters). Variables of these types store the value of the variable directly. These are not objects. These are called primitive variables.

An example is shown below: Primitive Variables contains bits representing the value of the variable.

```
int value = 5;
```

Different primitive types in java are char, boolean, byte, short, int, long, double, or float. Because of these primitive types, Java is NOT considered to be a pure objected oriented language.

Numeric Data Types

- Types : byte, short, int, long, float, double
- Number of bits : 8, 16, 32, 64, 32, 64
- Range : -x to x-1 where x = Power(2, number of bits -1)

char Data Type

- Used to store characters. Size of character is 16 bits.

Examples

```
int i = 15;
long longValue = 10000000000001;
byte b = (byte)254;

float f = 26.012f;
double d = 123.567;
boolean isDone = true;
boolean isGood = false;
char ch = 'a';
char ch2 = ';';
```

Reference Variables

```
Animal dog = new Animal();
```

The instance of new Animal - Animal object - is created in memory. The memory address of the object created is stored in the dog reference variable.

Reference Variables contains a reference or a guide to get to the actual object in memory.

Puzzles

```
Animal dog1 = new Animal();
dog1 = new Animal();
```

What will happen? Two objects of type Animal are created. Only one reference variable is created.

```
Animal animal1 = new Animal();
Animal animal2 = new Animal();
animal1 = animal2;
```

What will happen? What would happen if the same was done with primitive variables?

Identifiers

Names given to a class, method, interface, variables are called identifiers.

Legal Identifier Names

- Combination of letters, numbers, $ and under-score(_)
- Cannot start with a number
- Cannot be a keyword
- No limit on length of identifier

Java Keywords

List of Java Keywords

- Primitives DataTypes : byte,short,int,long,float,double,char,boolean
- Flow Control : if, else,for,do, while, switch, case, default, break, continue,return
- Exception Handling : try, catch, finally,throw,throws,assert
- Modifiers : public,private,protected,final,static,native,abstract, synchronized,transient,volatile,strictfp
- Class Related : class,interface,package,extends,implements,import
- Object Related : new, instanceof,super,this
- Literals : true, false, null
- Others : void, enum
- Unused : goto,const

Literals

Any primitive data type value in source code is called Literal.

There are four types of literals:

- Integer & Long
- Floating Point
- Boolean
- Double

Literals

Integer Literals

- There are 3 ways of representing an Integer Literal.
    - Decimal. Examples: 343, 545
    - Octal. Digits 0 to 7. Place 0 before a number. Examples : 070,011
    - Hexadecimal. Digits 0 to 9 and alphabets A to F (10-15). Case insensitive.
- An integer literal by default is int.

Long Literals

- All 3 integer formats: Decimal, Octal and Hexadecimal can be used to represent long by appending with L or l.

Floating point Literals

- Numbers with decimal points. Example: `double d = 123.456;`
- To declare a float, append f. Example: float f = 123.456f;
- Floating point literals are double by default.
- Appending d or D at end of double literal is optional Example: `double d = 123.456D;`

Boolean Literals

- Valid boolean values are true and false.
- TRUE, FALSE or True, False are invalid.

Character Literals

- Represented by single character between single quotes Example: `char a = 'a'`
- Unicode Representation also can be used. Prefix with \u. Example: char letterA = '\u0041';
- A number value can also be assigned to character. Example: char letterB = 66; Numeric value can be from 0 to 65535;
- Escape code can be used to represent a character that cannot be typed as literal. Example: char newLine = '\n';

Puzzles

```
int eight = 010;
int nine=011;
int invalid = 089;//COMPILER ERROR! 8 and 9 are invalid in Octal
int sixteen = 0x10;
int fifteen = 0XF;
int fourteen = 0xe;
int x = 23,000;
long a = 1234567891;
long b = 0x9ABCDEFGHL;
long c = 0123456789L;

float f = 123.456;//COMPILER ERROR! A double value cannot be assigned to a
float.

boolean b = true; boolean b=false;
boolean b = TRUE;//COMPILATION ERROR
boolean b = 0; //COMPILER ERROR. This is not C Language

char ch = a;
char a = 97;
char ch1 = 66000; //COMPILER ERROR!
```

Tip - Assignment Operator

Assignment operator evaluates the expression on the right hand side and copies the value into the variable on the left hand side.

Basic Examples

```
int value = 35;//35 is copied into 35

int squareOfValue = value * value;//value * value = 35 * 35 is stored into
squareOfValue

int twiceOfValue = value * 2;
```

Puzzles

```
int a1 = 5;
int b1 = 6;
b1 = a1; // value of a1 is copied into b1
a1 = 10; // If we change a1 or b1 after this, it would not change the other
variable.. b1 will remain 6

Actor actor1 = new Actor();
actor1.setName("Actor1");
//This creates new reference variable actor1 of type Actor  new Actor() on the
heap assigns the new Actor on the heap to reference variable

Actor actor2 = actor1;
actor2.setName("Actor2");
System.out.println(actor1.getName());//Actor2
```

Casting - Implicit and Explicit

Casting is used when we want to convert one data type to another.

- A literal integer is by default int. Operation involving int-sized or less always result in int.
- Floating point literals are by default double

Implicit Casting

- Implicit Casting is done directly by the compiler.
    - Example : Widening Conversions i.e. storing smaller values in larger variable types.

```
byte b = 10; //byte b = (int) 10; Example below compiles because compiler
introduces an implicit cast.

short n1 = 5;
short n2 = 6;
//short sum = n1 + n2;//COMPILER ERROR
short sum = (short)(n1 + n2);//Needs an explicit cast

byte b = 5;
b += 5; //Compiles because of implicit conversion

int value = 100;
long number = value; //Implicit Casting
float f = 100; //Implicit Casting
```

Explicit Casting

- Explicit Casting needs to be specified by programmer in code.
    - Example: Narrowing Conversions. Storing larger values into smaller variable types;

- Explicit casting would cause truncation of value if the value stored is greater than the size of the variable.

```
long number1 = 25678;
int number2 = (int)number1;//Explicit Casting
//int x = 35.35;//COMPILER ERROR
int x = (int)35.35;//Explicit Casting

int bigValue = 280;
byte small = (byte) bigValue;
System.out.println(small);//output 24. Only 8 bits remain.

//float avg = 36.01;//COMPILER ERROR. Default Double
float avg = (float) 36.01;//Explicit Casting
float avg1 = 36.01f;
float avg2 = 36.01F; //f or F is fine

//byte large = 128; //Literal value bigger than range of variable type causes compilation error
byte large = (byte) 128;//Causes Truncation!
```

Compound Assignment Operators

- Examples : +=, -=, *=

```
int a = 5;
a += 5; //similar to a = a + 5;
a *= 10;//similar to a = a * 10;
a -= 5;//similar to a = a - 5;
a /= 5;//similar to a = a / 5;
```

Other Operators

Remainder(%) Operator

- Remainder when one number is divided by another.

```
System.out.println(10 % 4);//2
System.out.println(15 % 4);//3
System.out.println(-15 % 4);//-3
```

Conditional Operator

- Conditional Operator is a Ternary Operator (3 Operands)
- syntax : `booleanCondition ? ResultIfTrue: ResultIfFalse;`

```
int age = 18;

System.out.println(
age >= 18 ? "Can Vote": "Cannot Vote");//Can Vote

age = 15;

System.out.println(
age >= 18 ? "Can Vote": "Cannot Vote");//Cannot Vote
```

Bitwise Operators

- You can work at bit level with these operators.
- & is bitwise AND, | is bitwise OR, ~ is bitwise complement (negation), ^ is bitwise XOR, << is left shift bitwise operator and >> is right shift bitwise operator.

```
System.out.println(25|12);//output will be 29
/*convert to binary and calculate:
00001100 (12 in decimal)
00011001 (25 in decimal)
________
00011101 (29 in decimal) */
System.out.println(25&12);//output will be 8
System.out.println(25^12);//output will be 21
```

Passing Variables to Methods

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

Passing Variables to Methods : Example

- Passing a primitive variable and modifying the value in a method
- Passing a reference variable and modifying the value in a method

Returning a Value From Method

- null is a valid return value for an object.
- You can return andy type that can be implicitly coverted to return type.
- You cannot return anything from a void method.

Types of Variables

- Different Types of Variables: Static, Member (or instance), Local, Block

Instance Variables

- Declared inside a class outside any method.
- Each instance of the class would have its own values.

- Also called member value, field or property.

Local Variables

- Variables declared in a method
- Local Variables can only be marked with final modifier
- If the name of a Local Variable is same as the name of an instance variable, it results in shadowing.

Member Variables

- Defined at class level and without keyword static.

Static Variable

- Defined at class level and using keyword static.

Member Variable and Static Variable

- Member Variables can be accessed only through object references.
- Static Variables can be accessed through a. Class Name and b. Object Reference. It is NOT recommended to use object reference to refer to static variables.

Example Static and Member Variables

```
public class StaticAndMemberVariables {
    public static void main(String[] args) {
  Actor actor1 = new Actor();
  actor1.name = "ACTOR1";
  //Actor.name //Compiler Error

  //Below statement can be written as actor1.count++
  //But NOT recommended.
  Actor.count++;

  Actor actor2 = new Actor();
  actor2.name = "ACTOR2";

  //Below statement can be written as actor2.count++
  //But NOT recommended.
  Actor.count++;

  System.out.println(actor1.name);//ACTOR1
  System.out.println(actor2.name);//ACTOR2

  //Next 3 statements refer to same variable
  System.out.println(actor1.count);//2
  System.out.println(actor2.count);//2
  System.out.println(Actor.count);//2
    }
}
```

```
class Actor {
    //RULE 1: Member Variables can be accessed
    //only through object references
    String name;

    //RULE 2:Static Variables can be accessed
    //through a.Class Name and b.Object Reference
    //It is NOT recommended to use object reference
    //to refer to static variables.
    static int count;
}
```

Scope of a Variable

- Scope of a variable defines where (which part of code) a variable can be accessed.

Important Rules

- Static Variable can be used anywhere in the class.
- Member Variable can be used in any non-static method.
- Local Variable can be used only in the method where it is declared.
- Block Variable can be used only in the block (code between { and }) where it is declared.

Variable Scope Examples

Below code shows all these Rules in action:

```
public class VariablesExample {
    //RULE 1:Static Variable can be used anywhere in the class.
    static int staticVariable;

    //RULE 2:Member Variable can be used in any non-static method.
    int memberVariable;

    void method1() {
    //RULE 3: method1LocalVariable can be used only in method1.
    int method1LocalVariable;

    memberVariable = 5;//RULE 2
    staticVariable = 5;//RULE 1

    //Some Code
    {
        //RULE 4:blockVariable can be used only in this block.
        int blockVariable;
        //Some Code
    }
```

```
    //blockVariable = 5;//COMPILER ERROR - RULE 4
    }

    void method2() {
    //method1LocalVariable = 5; //COMPILER ERROR - RULE3
    }

    static void staticMethod() {
    staticVariable = 5;//RULE 1
    //memberVariable = 5; //COMPILER ERROR - RULE 2
    }
}
```

Scope Example 1

- staticVariable is declared using keyword static.
- It is available in the instance method method1 and static method named staticMethod.

Scope Example 2

- memberVariable is declared directly in the class and does NOT use keyword static. So, it is an instance variable.
- It is available in the instance method method1 but not accessible in the static method named staticMethod.

Scope Example 3

- method1LocalVariable is declared in the method method1. So, it is a local variable.
- It is available in the instance method method1 but available in any other instance or static methods.

Scope Example 4

- blockVariable is declared in a block in method1. So, it is a block variable.
- It is available only in the block where it is defined.
- It is not accessible any where out side the block , even in the same method.

Variable Initialization

- Initialization defines the default value assigned to a variable if it is not initialized.

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.

Variable Initialization Examples

Lets look at an example program to understand all the rules regarding variable initialization.

```
package com.in28minutes.variables;

//RULE1: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 object reference variable is null.

//RULE2:Local/block variables are NOT initialized by compiler.

//RULE3    :If local variables are used before initialization,
//it would result in Compilation Error

public class VariableInitialization {
    public static void main(String[] args) {
    Player player = new Player();

    //score is an int member variable - default 0
    System.out.println(player.score);//0 - RULE1

    //name is a member reference variable - default null
    System.out.println(player.name);//null - RULE1

    int local; //not initialized
    //System.out.println(local);//COMPILER ERROR! RULE3

    String value1;//not initialized
    //System.out.println(value1);//COMPILER ERROR! RULE3

    String value2 = null;//initialized
    System.out.println(value2);//null - NO PROBLEM.
    }
}


class Player{
    String name;
    int score;
}
```

Initialization Example 1

- player is an instance of the class Player. It contains member variables named name and score.
- All member variables are initialized by default. Since name refers to a String i.e a reference variable it is initialized to null. score is an int variable and hence initialized to 0.

Initialization Example 2

- local is a local variable defined in the main method.
- An attempt to access a local variable without initialization would result in a compilation error.
- Same is the case with value1 which is a String local variable.
- If null is assigned to a reference variable, reference variable is considered to be assigned.

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
- A primitive wrapper class in the Java programming language is one of eight classes provided in the java.lang package to provide object methods for the eight primitive types. All of the primitive wrapper classes in Java are immutable.

Wrapper classes are final and immutable.

Creating Wrapper Classes

```
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
//Character c2 = new Character(124);//COMPILER ERROR

Boolean b = new Boolean(true);

//"true" "True" "tRUe" - all String Values give True
```

```
//Anything else gives false
Boolean b1 = new Boolean("true");//value stored - true
Boolean b2 = new Boolean("True");//value stored - true
Boolean b3 = new Boolean("False");//value stored - false
Boolean b4 = new Boolean("SomeString");//value stored - false

b = false;
```

Wrapper Class Utility Methods

- A number of utility methods are defined in wrapper classes to create and convert them.

valueOf Methods

Provide another way of creating a Wrapper Object

```
Integer seven =
    Integer.valueOf("111", 2);//binary 111 is converted to 7

Integer hundred =
    Integer.valueOf("100");//100 is stored in variable
```

xxxValue methods

xxxValue methods help in creating primitives

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

Float floatWrapper = Float.valueOf(57.0f);
int floatToInt = floatWrapper.intValue();//57
float floatToFloat = floatWrapper.floatValue();//57.0f
```

parseXxx methods

parseXxx methods are similar to valueOf but they return primitive values

```
int sevenPrimitive =
    Integer.parseInt("111", 2);//binary 111 is converted to 7

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

static toString method

Look at the example of the toString static method below.

```
Integer wrapperEight = new Integer(8);
System.out.println(Integer.
toString(wrapperEight));//String Output: 8
```

Overloaded static toString method

2nd parameter: radix

```
System.out.println(Integer
.toString(wrapperEight, 2));//String Output: 1000
```

static toYyyyString methods.

Yyyy can be Hex,Binary,Octal

```
System.out.println(Integer
.toHexString(wrapperEight));//String Output:8
System.out.println(Integer
.toBinaryString(wrapperEight));//String Output:1000
System.out.println(Integer
.toOctalString(wrapperEight));//String Output:10
```

Wrapper Class , Auto Boxing

```
Integer ten = new Integer(10);
ten++;//allowed. Java does the work behind the screen for us
```

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
```

String Class

- A String class can store a sequence of characters. String is not a primitive in Java but a Class in its own right.

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

```
System.out.println(5 + "Test" + 5); //5Test5
System.out.println(5 + 5 + "Test"); //10Test
System.out.println("5" + 5 + "Test"); //55Test
System.out.println("5" + "5" + "25"); //5525
System.out.println(5 + 5 + "25"); //1025
System.out.println("" + 5 + 5 + "25"); //5525
System.out.println(5 + (5 + "25")); //5525
System.out.println(5 + 5 + 25); //35
```

Increment and Decrement Operators

- Lets learn about the increment and decrement operators in Java.

Basics of Increment and Decrement Operators

Except for a minor difference ++i,i++ is similar to i = i+1 and --i,i-- is similar to i = i-1

++i is called pre-increment and i++ post increment

Increment Operators

Pre increment statement returns value after increment. Post increment statement returns value before increment

```
int i = 25;
int j = ++i;//i is incremented to 26, assigned to j
System.out.println(i + " " + j);//26 26

i = 25;
j = i++;//i value(25) is assigned to j, then incremented to 26
System.out.println(i + " " + j);//26 25
```

Decrement Operators

Decrement Operators are similar to increment operators.

```
i = 25;
j = --i;//i is decremented to 24, assigned to j
System.out.println(i + " " + j);//24 24

i = 25;
j = i--;//i value(25) is assigned to j, then decremented to 24
System.out.println(i + " " + j);//24 25
```

Relational Operators

- Relation Operators are used to compare operands. They a always return true or false. List of Relation Operators include <, <=, >, >=, ==, and !=.

Relation Operators Examples

Let's consider a few examples of relational operators. Let's assume a int variable named number with a value 7.

```
int number = 7;
```

greater than operator

```
System.out.println(number > 5);//true
System.out.println(number > 7);//false
```

greater than equal to operator

```
System.out.println(number >= 7);//true
```

less than operator

```
System.out.println(number < 9);//true
System.out.println(number < 7);//false
```

less than equal to operator

```
System.out.println(number <= 7);//true
```

is equal to operator

```
System.out.println(number == 7);//true
System.out.println(number == 9);//false
```

NOT equal to operator

```
System.out.println(number != 9);//true
System.out.println(number != 7);//false
```

> single = is assignment operator and == is comparison. Below statement uses =.

```
System.out.println(number = 7);//7
```

== (equals) operator

Let's look at how == equals operator works with primitives and reference variables.

Primitive Variables

- Equality for Primitives only compares values

```
int a = 5;
int b = 5;
```

Below statement compares if a and b have same value.

```
System.out.println(a == b);//true
```

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

bReference = aReference;

//Now both are referring to same object
System.out.println(aReference == bReference);//true
```

Bitwise operators

- Bitwise Operators are |,&,^ and ~

```
int a = 5;
int b = 7;

// bitwise and
// 0101 & 0111=0101 = 5
System.out.println("a&b = " + (a & b));

// bitwise or
// 0101 | 0111=0111 = 7
System.out.println("a|b = " + (a | b));

// bitwise xor
// 0101 ^ 0111=0010 = 2
System.out.println("a^b = " + (a ^ b));

// bitwise and
// ~0101=1010
// will give 2's complement of 1010 = -6
System.out.println("~a = " + ~a);

// can also be combined with
// assignment operator to provide shorthand
// assignment
// a=a&b
a &= b;
System.out.println("a= " + a); // a = 5
```

Logical Operators

- Logical Operators are &&, ||, |, &, ! and ^.

Short Circuit And Operator - &&

- True when both operands are true.

```
System.out.println(true && true);//true
System.out.println(true && false);//false
System.out.println(false && true);//false
System.out.println(false && false);//false
```

Short Circuit Or Operator - ||

True when atleast one of operands are true.

```
System.out.println(true || true);//true
System.out.println(true || false);//true
System.out.println(false || true);//true
System.out.println(false || false);//false
```

> Certification Tip : Logical Operators work with boolean values but not numbers.

```
//System.out.println(5 || 6);//COMPILER ERROR
```

Short circuit operators are Lazy

- They stop execution the moment result is clear.
  - For &&, if first expression is false,result is false.
  - For ||, if first expression is true, the result is true.
  - In above 2 situations, second expressions are not executed.

```
int i = 10;
System.out.println(true || ++i==11);//true
System.out.println(false && ++i==11);//false
System.out.println(i);//i remains 10, as ++i expressions are not executed.
```

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
```

Operator exclusive-OR (^)

- Result is true only if one of the operands is true.

```
System.out.println(true ^ false);//true
System.out.println(false ^ true);//true
System.out.println(true ^ true);//false
System.out.println(false ^ false);//false
```

Not Operator (!)

Result is the negation of the expression.

```
System.out.println(!false);//true
System.out.println(!true);//false
```

Arrays

- TODO : Why do we need 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;
marks[1] = 30;
marks[2] = 50;
marks[3] = 10;
marks[4] = 5;

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" };
```

2D Arrays

Best way to visualize a 2D array is as an array of arrays.

```
int[][] matrix = { { 1, 2, 3 }, { 4, 5, 6 } };

int[][] matrixA = new int[5][6];

//Accessing elements from 2D array:
System.out.println(matrix[0][0]); //1
System.out.println(matrix[1][2]); //6

//Looping a 2D array
for (int[] array: matrix) {
    for (int number: array) {
         System.out.println(number);
    }
}

// Printing a 2D Array
int[][] matrix3 = { { 1, 2, 3 }, { 4, 5, 6 } };
System.out.println(matrix3); //[[I@1d5a0305
System.out.println(

Arrays.toString(matrix3));
//[[I@6db3f829, [I@42698403]

System.out.println(Arrays.deepToString(matrix3));
//[[1, 2, 3], [4, 5, 6]]

System.out.println(matrix3[0]);//[I@86c347 - matrix3[0] is a 1D Array
System.out.println(Arrays.toString(matrix3[0]));//[1, 2, 3]
```

Other Array Operations

```
//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]
```

Array of Objects

```
Person[] persons = new Person[3];

//By default, an array of 3 reference variables is created.
//The person objects are not created
System.out.println(persons[0]);//null

//Let's create the new objects
persons[0] = new Person();
persons[1] = new Person();
persons[2] = new Person();

//Creating and initializing person array in one statement
Person[] personsAgain = { new Person(),new Person(),new Person()};

//Another example
Person[][] persons2D =
    {
    { new Person(),new Person(),new Person()},
    { new Person(),new Person()}
    };
```

Array Certification Tips and Puzzles

```
//You can Declare, Create and Initialize Array on same line.
int marks3[] = { 25, 30, 50, 10, 5 };

//Leaving additional comma is not a problem. (note that comma after 5)
int marks4[] = { 25, 30, 50, 10, 5, };


int marks[]; //Not Readable
```

```
int[] runs; //Readable


//int values[5];//Compilation Error!Declaration of an Array should not include size.

//marks = new int[];//COMPILER ERROR! Size of an array is mandatory to create an array.


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

//Access 10th element when array has only length 5
//Runtime Exception: ArrayIndexOutOfBoundsException
//System.out.println(marks[10]);

//Array can contain only values of same type.

//COMPILE ERROR!!
//int marks4[] = {10,15.0}; //10 is int 15.0 is float

//Cross assigment of primitive arrays is ILLEGAL
int[] ints = new int[5];
short[] shorts = new short[5];
//ints = shorts;//COMPILER ERROR
//ints = (int[])shorts;//COMPILER ERROR


//The first dimension of a 2D array is mandatory
matrixA = new int[3][];//FINE
//matrixA = new int[][5];//COMPILER ERROR
//matrixA = new int[][];//COMPILER ERROR

//Each row in a 2D Array can have a different size. This is called a Ragged Array.
matrixA = new int[3][];//FINE
matrixA[0] = new int[3];
matrixA[0] = new int[4];
matrixA[0] = new int[5];
```

If Else Condition

- Conditionally execute code!
- Code inside If is executed only if the condition is true.

// Basic Example

```
if(true){
    System.out.println("Will be printed");
}

if(false){
    System.out.println("Will NOT be printed");//Not executed
}

//Example 1
int x = 5;

if(x==5){
    System.out.println("x is 5");//executed since x==5 is true
}

//Example 2
x = 6;
if(x==5){
    System.out.println("x is 5");//Not executed since x==5 is false
}

//Example 3
int y = 10;

if(y==10){
    System.out.println("Y is 10");//executed-condn y==10 is true
} else {
    System.out.println("Y is Not 10");
}

//Example 4
y = 11;

if(y==10){
    System.out.println("Y is 10");//NOT executed
} else {
    System.out.println("Y is Not 10");//executed
}

//Example 5
int z = 15;
//Only one condition is executed. Rest of the conditions are skipped.
if(z==10){
    System.out.println("Z is 10");//NOT executed
} else if(z==12){
    System.out.println("Z is 12");//NOT executed
} else if(z==15){
    System.out.println("Z is 15");//executed.
} else {
```

```
    System.out.println("Z is Something Else.");//NOT executed
}

z = 18;
if(z==10){
    System.out.println("Z is 10");//NOT executed
} else if(z==12){
    System.out.println("Z is 12");//NOT executed
} else if(z==15){
    System.out.println("Z is 15");//NOT executed
} else {
    System.out.println("Z is Something Else.");//executed
}

//If else Example: without Blocks
int number = 5;
if(number < 0)
    number = number + 10; //Not executed
    number++; //This statement is not part of if. Executed.
System.out.println(number);//prints 6
```

If else Puzzles

```
//Puzzle 1
int k = 15;
if (k > 20) {
    System.out.println(1);
} else if (k > 10) {
    System.out.println(2);
} else if (k < 20) {
    System.out.println(3);
} else {
    System.out.println(4);
}

//Output is 2.
//Once a condition in nested-if-else is true the rest of the code is not
executed.


//Puzzle 2
int l = 15;

if(l<20)
    System.out.println("l<20");
if(l>20)
    System.out.println("l>20");
else
    System.out.println("Who am I?");
```

//Output is "I<20" followed by "Who am I?" on next line. //else belong to the last if before it unless brackets ({}) are used.

```
//Puzzle 3

int m = 15;

if(m>20)
    if(m<20)
System.out.println("m>20");
    else
System.out.println("Who am I?");

//Nothing is printed to output.

//Code above is similar to the code snippet shown below

if(m>20) {//Condn is false. So, code in if is not executed
    if(m<20)
System.out.println("m>20");
    else
System.out.println("Who am I?");
}
```

Puzzles Continued

```
//Puzzle 4

int x1 = 0;
//Condition in if should always be boolean
//if(x1) {} //COMPILER ERROR
//if(x1=0) {}//COMPILER ERROR. Using = instead of ==
//If else condition should be boolean

//Puzzle 5

boolean isTrue = false;

if(isTrue==true){
    System.out.println("TRUE TRUE");//Will not be printed
}

if(isTrue=true){
    System.out.println("TRUE");//Will be printed.
}
```

```
//Condition is isTrue=true. This is assignment. Returns true. So, code in if is executed.
```

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;
}
// Output of above example is 2.The case which is matched is executed.
```

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.

```
//Switch Statement Example 2 , No Breaks
number = 2;
switch (number) {
case 1:
    System.out.println(1);
case 2:
    System.out.println(2);
case 3:
    System.out.println(3);
default:
    System.out.println("Default");
}
```

Output of above switch

```
2
3
Default
```

Since there is no break after case 2, execution falls through to case 3. There is no break in case 3 as well. So, execution falls through to default.

> Code in switch is executed from a matching case until a break or end of switch statement is encountered.

Switch Statement Example 3 , Few Break's

```
number = 2;
switch (number) {
case 1:
    System.out.println(1);
    break;
case 2:
case 3:
    System.out.println("Number is 2 or 3");
    break;
default:
    System.out.println("Default");
    break;
}
```

Program Output

```
number is 2 or 3.
```

Case 2 matches. Since there is no code in case 2, execution falls through to case 3, executes the println. Break statement takes execution out of the switch.

Switch Statement Example 4 , Let's Default

- default is executed if none of the case's match.

```
number = 10;
switch (number) {
case 1:
    System.out.println(1);
    break;
case 2:
    System.out.println(2);
    break;
case 3:
    System.out.println(3);
    break;
default:
```

```
    System.out.println("Default");
    break;
}
```

Code Output

```
Default
```

Switch Statement Example 5 - Default need not be Last

```
number = 10;
switch (number) {
default:
    System.out.println("Default");
    break;
case 1:
    System.out.println(1);
    break;
case 2:
    System.out.println(2);
    break;
case 3:
    System.out.println(3);
    break;
}
```

Output

```
Default
```

Switch statement Example 6

Switch can be used only with char, byte, short, int, String or enum

```
long l = 15;
/*switch(l){//COMPILER ERROR. Not allowed.
}*/
```

Case value should be a compile time constant.

```
number = 10;
switch (number) {
//case number>5://COMPILER ERROR. Cannot have a condition
//case number://COMPILER ERROR. Should be constant.
}
```

Loops

A loop is used to run same code again and again.

While Loop

```
int count = 0;

while(count < 5){//while this condn is true, loop is executed.
    System.out.print(count);
    count++;
}
//Output - 01234
```

While loop Example 2

```
count = 5;
while(count < 5){//condn is false. So, code in while is not executed.
    System.out.print(count);
    count++;
}//Nothing is printed to output
```

Do While Loop

- The difference between a while and a do while is that the code in do while is executed at least once.
- In a do while loop, condition check occurs after the code in loop is executed once.

Do While loop Example 1

```
int count = 0;
do{
    System.out.print(count);
    count++;
}while(count < 5);//while this condn is true, loop is executed.
//output is 01234
```

Do While loop Example 2

```
count = 5;
do{
    System.out.print(count);
    count++;
}while(count < 5);
//output is 5
```

For Loop

For loop is used to loop code specified number of times.

For Loop Example 1

```
for (int i = 0; i < 10; i++) {
    System.out.print(i);
}
//Output - 0123456789
```

Syntax - For loop statement has 3 parts

- Initialization => int i=0. Initialization happens the first time a for loop is run.
- Condition => i<10. Condition is checked every time before the loop is executed.
- Operation (Increment or Decrement usually) => i++. Operation is invoked at the start of every loop (except for first time).

For Loop Example 2: There can be multiple statements in Initialization or Operation separated by commas

```
for (int i = 0,j = 0; i < 10; i++,j--) {
    System.out.print(j);
}
//Output - 0123456789
```

Enhanced For Loop

Enhanced for loop can be used to loop around array's or List's.

```
int[] numbers = {1,2,3,4,5};

for(int number:numbers){
    System.out.print(number);
}
//Output - 12345
```

Any of 3 parts in a for loop can be empty.

```
for (;;) {
    System.out.print("I will be looping for ever..");
}
//Infinite loop => Loop executes until the program is terminated.
```

Break Statement

Break statement breaks out of a loop

Example 1

```
for (int i = 0; i < 10; i++) {
    System.out.print(i);
    if (i == 5) {
         break;
    }
}
//Output - 012345
//Even though the for loop runs from 0 to 10, execution stops at i==5 because
of the break statement. ÒBreak statementÓ stops the execution of the loop and
takes execution to the first statement after the loop.
```

Break can be used in a while also.

```
int i = 0;
while (i < 10) {
    System.out.print(i);
    if (i == 5) {
break;
    }
    i++;
}
//Output - 012345
```

Break statement takes execution out of inner most loop.

```
for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 10; k++) {
System.out.print(j + "" + k);
if (k == 5) {
    break;//Takes out of loop using k
}
    }
}
//Output - 000102030405101112131415
//Each time the value of k is 5 the break statement is executed.
//The break statement takes execution out of the k loop and proceeds to the
next value of j.
```

Labels can be used to label and refer to specific for loop in a nested for loop.

```
outer:
    for (int j = 0; j < 2; j++) {
       for (int k = 0; k < 10; k++) {
             System.out.print(j + "" + k);
             if (k == 5) {
               break outer;//Takes out of loop using j
             }
        }
    }
//Output - 000102030405
```

Continue Statement

- Continue statement skips rest of the statements in the loop and starts next iteration

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
      continue;
    }
    System.out.print(i);
}

//Output => 012346789

//Note that the output does not contain 5.
//When i==5 continue is executed. Continue skips rest of the code and goes to
next loop iteration.
//So, the print statement is not executed when i==5.
```

Continue can be used in a while also

```
int i = 0;
while (i < 10) {
    i++;
    if (i == 5) {
        continue;
    }
    System.out.print(i);
}
//Output - 1234678910
```

Continue statement takes execution to next iteration of inner most loop.

```
for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 10; k++) {
       if (k == 5) {
          continue;//skips to next iteration of k loop
        }
        System.out.print(j + "" + k);
    }
}
//Output - 00010203040607080910111213141617181 9
//When k==5 the print statement in the loop is skipped due to continue.
//So 05 and 05 are not printed to the console.
```

Label Example

```
outer:
    for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 10; k++) {
        if (k == 5) {
        continue outer;//skips to next iteration of j loop
        }
        System.out.print(j + "" + k);
    }
    }
//Output - 00010203041011121314
//When k==5 is true, continue outer is called.
//So, when value of k is 5, the loop skips to the next iteration of j.
```

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;
}
```

Enum Example 3

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

public class EnumAdvanced2 {

  // Enum with a variable,method and constructor
  enum SeasonCustomized {
    WINTER(1) {
      public int getExpectedMaxTemperature() {
        return 5;
      }
    },
    SPRING(2), SUMMER(3) {
      public int getExpectedMaxTemperature() {
        return 20;
      }
    },
    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;
    }

    public int getExpectedMaxTemperature() {
      return 10;
    }

  };
```

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

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

    System.out.println(SeasonCustomized.FALL.getExpectedMaxTemperature());//
10

  }

}
```

Enum Constant Class - In the example above, take a look at how the Winter Type is declared: It provides an overriding implementation for the getExpectedMaxTemperature method already declared in the Enum. This feature in an Enum is called a Constant Class.

```
WINTER(1) {
    public int getExpectedMaxTemperature() {
         return 5;
    }
}
```

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
}
```

Create a class Actor

```
public class Actor {
    public void act(){
        System.out.println("Act");
    };
}
```

We can extend this class by using the keyword `extends`. `Hero class extends Actor`

```
//IS-A relationship. Hero is-a Actor
public class Hero extends Actor {
    public void fight(){
       System.out.println("fight");
    };
}
```

Since Hero extends Actor, the methods defined in Actor are also available through an instance of Hero class.

```
Hero hero = new Hero();
//act method inherited from Actor
hero.act();//Act
hero.fight();//fight
```

Let's look at another class extending Actor class - Comedian.

```
//IS-A relationship. Comedian is-a Actor
public class Comedian extends Actor {
    public void performComedy(){
       System.out.println("Comedy");
    };
}
```

Methods in Animal class can be executed from an instance of Comedian class.

```
Comedian comedian = new Comedian();
//act method inherited from Actor
comedian.act();//Act
comedian.performComedy();//Comedy
```

Super class reference variable can hold an object of sub class

```
Actor actor1 = new Comedian();
Actor actor2 = new Hero();
```

Object is super class of all classes. So, an Object reference variable can hold an instance of any class.

```
//Object is super class of all java classes
Object object = new Hero();
```

Inheritance: IS-A Relationship

We should use inheritance only when there is an IS-A relationship between classes. For example, Comedian IS-A Actor, Hero IS-A Actor are both true. So, inheritance is correct relationship between classes.

- Comedian is called a Sub Class. Actor is Super Class.

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

```
class Dog extends Animal, Pet { //COMPILER ERROR
}
```

We can create an inheritance chain.

```
class Pet extends Animal {
}

class Dog extends Pet {
}
```

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
```

Puzzle and Tips - instanceof Operator in depth

instanceof operator checks if an object is of a particular type.

```
class SuperClass {
};

class SubClass extends SuperClass {
};

interface Interface {
};

class SuperClassImplementingInteface implements Interface {
};

class SubClass2 extends SuperClassImplementingInteface {
};

class SomeOtherClass {
};

SubClass subClass = new SubClass();
Object subClassObj = new SubClass();

SubClass2 subClass2 = new SubClass2();
SomeOtherClass someOtherClass = new SomeOtherClass();

//We can run instanceof operator on the different instances created earlier.
System.out.println(subClass instanceof SubClass);//true
System.out.println(subClass instanceof SuperClass);//true
System.out.println(subClassObj instanceof SuperClass);//true
```

```
System.out.println(subClass2 instanceof SuperClassImplementingInteface);//true

//instanceof can be used with interfaces as well.
//Since Super Class implements the interface, below code prints true.
System.out.println(subClass2
instanceof Interface);//true

//If the type compared is unrelated to the object, a compilation error occurs.
//System.out.println(subClass
//    instanceof SomeOtherClass);//Compiler Error

//Object referred by subClassObj(SubClass)- NOT of type SomeOtherClass
System.out.println(subClassObj instanceof SomeOtherClass);//false
```

Class, Object, State and Behavior

- In this tutorial, lets look at a few important object oriented concepts.

Class, Object, State and Behavior Example

```
package com.in28minutes;

public class CricketScorer {
    //Instance Variables - constitute the state of an object
    private int score;

    //Behavior - all the methods that are part of the class
    //An object of this type has behavior based on the
    //methods four, six and getScore
    public void four(){
score = score + 4;
    }

    public void six(){
score = score + 6;
    }

    public int getScore() {
return score;
    }

    public static void main(String[] args) {
CricketScorer scorer = new CricketScorer();
scorer.six();
//State of scorer is (score => 6)
scorer.four();
//State of scorer is (score => 10)
System.out.println(scorer.getScore());
```

```
    }
}
```

Class

A class is a Template.

- In above example, class CricketScorer is the template for creating multiple objects.

A class defines state and behavior that an object can exhibit.

Object

An instance of a class.

- In the above example, we create an object using new CricketScorer().
- The reference of the created object is stored in scorer variable.
- We can create multiple objects of the same class.

State

State represents the values assigned to instance variables of an object at a specific time.

Consider following code snippets from the above example.

- The value in score variable is initially 0.
- It changes to 6 and then 10.

State of an object might change with time.

```
scorer.six();
//State of scorer is (score => 6)

scorer.four();
//State of scorer is (score => 10)
```

Behavior

Behaviour of an object represents the different methods that are supported by it.

- Above example the behavior supported is six(), four() and getScore().

toString method

toString() method in Java is used to print the content of an object.

Example

```
class Animal {

    public Animal(String name, String type) {
      this.name = name;
      this.type = type;
```

```
    }

    String name;
    String type;

}



Animal animal = new Animal("Tommy", "Dog");

//Output does not show the content of animal (what name? and what type?).
System.out.println(animal);//com.in28minutes.Animal@f7e6a96
```

To show the content of the animal object, we can override the default implementation of toString method provided by Object class.

```
//Adding toString to Animal class
class Animal {

    public Animal(String name, String type) {
        this.name = name;
        this.type = type;
    }

    String name;
    String type;

    public String toString() {
        return "Animal [name=" + name + ", type=" + type + "]";
    }

}

Animal animal = new Animal("Tommy","Dog");

//Output now shows the content of the animal object.
System.out.println(animal);//Animal [name=Tommy, type=Dog]
```

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.

Example

```
class Client {
    private int id;

    public Client(int id) {
      this.id = id;
    }

    @Override
    public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + id;
      return result;
    }
}

// == comparison operator checks if the object references are pointing to the
same object.
// It does NOT look at the content of the object.
Client client1 = new Client(25);
Client client2 = new Client(25);
Client client3 = client1;

//client1 and client2 are pointing to different client objects.
System.out.println(client1 == client2);//false

//client3 and client1 refer to the same client objects.
System.out.println(client1 == client3);//true

//similar output to ==
System.out.println(client1.equals(client2));//false
System.out.println(client1.equals(client3));//true

//overriding equals method
class Client {
    private int id;

    public Client(int id) {
      this.id = id;
    }

    @Override
    public boolean equals(Object obj) {
      Client other = (Client) obj;
      if (id != other.id)
          return false;
      return true;
```

```
    }
}
```

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.

Example

```
Client client1 = new Client(25);
Client client2 = new Client(25);
Client client3 = client1;

//both id's are 25
System.out.println(client1.equals(client2));//true

//both id's are 25
System.out.println(client1.equals(client3));//true
```

Any equals implementation should satisfy these properties:

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

Let's now provide an implementation of equals which satisfy these properties:

```
//Client class
@Override
public boolean equals(Object obj) {
    if (this == obj)
return true;
    if (obj == null)
return false;
    if (getClass() != obj.getClass())
return false;
    Client other = (Client) obj;
    if (id != other.id)
return false;
    return true;
```

```
}
```

hashCode method

- HashCode's are used in hashing to decide which group (or bucket) an object should be placed into.
    - A group of object's might share the same hashcode.
    - The implementation of hash code decides effectiveness of Hashing.
    - A good hashing function evenly distributes object's into different groups (or buckets).

hashCode method properties

- If obj1.equals(obj2) is true, then obj1.hashCode() should be equal to obj2.hashCode()
- obj.hashCode() should return the same value when run multiple times, if values of obj used in equals() have not changed.
- If obj1.equals(obj2) is false, it is NOT required that obj1.hashCode() is not equal to obj2.hashCode(). Two unequal objects MIGHT have the same hashCode.

Example

```
//Client class
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + id;
    return result;
}
```

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.

Constructor Example 1: Default Constructor

In the example below, there are no Constructors defined in the Animal class. Compiler provides us with a default constructor, which helps us create an instance of animal class.

```
public class Animal {
    String name;

    public static void main(String[] args) {
// Compiler provides this class with a default no-argument constructor.
// This allows us to create an instance of Animal class.
Animal animal = new Animal();
    }
}
```

Constructor Example 2: 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.

```
class Animal {
    String name;

    // This is called a one argument constructor.
    public Animal(String name) {
this.name = name;
    }

    public static void main(String[] args) {
// Since we provided a constructor, compiler does not
// provide a default constructor.
// Animal animal = new Animal();//COMPILER ERROR!

// The only way we can create Animal1 object is by using
Animal animal = new Animal("Tommy");
    }
}
```

Constructor Example 3: 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.

```
class Animal {
    String name;

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

    // This is called a one argument constructor.
    public Animal(String name) {
this.name = name;
    }

    public static void main(String[] args) {
// Since we provided a constructor, compiler does not
// provide a default constructor.
// Animal animal = new Animal();//COMPILER ERROR!

// The only way we can create Animal1 object is by using
Animal animal = new Animal("Tommy");
    }
}
```

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.

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.

Below examples will throw a compilation error if the super or this calls are uncommented.

```
public Animal() {
    this.name = "Default Name";
    //super(), if called, should always the first statement in a constructor.
    //super(); //COMPILER ERROR
}

public Animal() {
    System.out.println("Creating an Animal");
    //this(string), if called, should always the first statement in a
constructor.
    //this("Default Name");//COMPILER ERROR
}
```

Constructor Example 5

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
}
```

Constructor Example 6: Constructor cannot be directly called

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

```
class Animal {
    String name;

    public Animal() {
    }

    public method() {
Animal();// Compiler error
    }
}
```

Constructor Example 7: Super Class Constructor is invoked automatically

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.

Consider the example below:

```
class Animal {
    public Animal() {
System.out.println("Animal Constructor");
    }
}

class Dog extends Animal {
    public Dog() {
System.out.println("Dog Constructor");
    }
}

class Labrador extends Dog {
    public Labrador() {
System.out.println("Labrador Constructor");
    }
}

public class ConstructorExamples {
    public static void main(String[] args) {
Labrador labrador = new Labrador();
    }
}
//Output - Animal Constructor
Dog Constructor
Labrador Constructor
```

It is almost as if super() method is invoked as the first line of every constructor. The example code below shows how the code above behaves.

```
class Animal {
    public Animal() {
super();// IMPLICIT CALL
System.out.println("Animal Constructor");
    }
}

class Dog extends Animal {
    public Dog() {
super();// IMPLICIT CALL
System.out.println("Dog Constructor");
    }
}

class Labrador extends Dog {
    public Labrador() {
super();// IMPLICIT CALL
System.out.println("Labrador Constructor");
    }
}
```

Constructor Example 8

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");
    }
}
```

public Dog() makes an implicit super() call i.e. a call to Animal() (no argument) constructor. But no such constructor is defined in Animal class.

Constructor Example 9

Similar example below except that the Dog no argument constructor is not provided by programmer. However, the compiler would give the no argument constructor, which would invoke super() method. This would again result in a compilation error.

```
class Animal {
    String name;

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

class Dog extends Animal {// COMPILER ERROR! No constructor for Animal()
}
```

Two ways to fix above errors. 1.Create a no arguments constructor in Animal class. 2.Make a explicit super("Default Dog Name") call in the Dog() constructor.

Creating a super class no argument constructor

```
class Animal {
    String name;

    public Animal() {
    }

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

Making an explicity super call

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

Constructors are NOT inherited.

```
class Animal {
    String name;
```

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

class Dog extends Animal {
}

public class ConstructorExamples {
    public static void main(String[] args) {
// Dog dog = new Dog("Terry");//COMPILER ERROR
    }
}
```

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

Solution is to create an explicit constructor in sub class invoking the super class constructor. Add below constructor to Dog class.

```
class Dog extends Animal {
    public Dog() {
super("Default Dog Name");
    }
}
```

Coupling

- Coupling is a measure of how much a class is dependent on other classes. There should minimal dependencies between classes. So, we should always aim for low coupling between classes.

Coupling Example Problem

Consider the example below:

```
class ShoppingCartEntry {
    public float price;
    public int quantity;
}

class ShoppingCart {
    public ShoppingCartEntry[] items;
}

class Order {
    private ShoppingCart cart;
    private float salesTax;
```

```
    public Order(ShoppingCart cart, float salesTax) {
this.cart = cart;
this.salesTax = salesTax;
    }

    // This method know the internal details of ShoppingCartEntry and
    // ShoppingCart classes. If there is any change in any of those
    // classes, this method also needs to change.
    public float orderTotalPrice() {
float cartTotalPrice = 0;
for (int i = 0; i < cart.items.length; i++) {
    cartTotalPrice += cart.items[i].price
    * cart.items[i].quantity;
}
cartTotalPrice += cartTotalPrice * salesTax;
return cartTotalPrice;
    }
}
```

Method orderTotalPrice in Order class is coupled heavily with ShoppingCartEntry and ShoppingCart classes. It uses different properties (items, price, quantity) from these classes. If any of these properties change, orderTotalPrice will also change. This is not good for Maintenance.

Coupling Example Solution

Consider a better implementation with lesser coupling between classes below: In this implementation, changes in ShoppingCartEntry or CartContents might not affect Order class at all.

```
class ShoppingCartEntry
{
    float price;
    int quantity;

    public float getTotalPrice()
    {
return price * quantity;
    }
}

class CartContents
{
    ShoppingCartEntry[] items;

    public float getTotalPrice()
    {
float totalPrice = 0;
for (ShoppingCartEntry item:items)
```

```
{
    totalPrice += item.getTotalPrice();
}
return totalPrice;
    }
}

class Order
{
    private CartContents cart;
    private float salesTax;

    public Order(CartContents cart, float salesTax)
    {
this.cart = cart;
this.salesTax = salesTax;
    }

    public float totalPrice()
    {
return cart.getTotalPrice() * (1.0f + salesTax);
    }
}
```

Cohesion

- Cohesion is a measure of how related the responsibilities of a class are. A class must be highly cohesive i.e. its responsibilities (methods) should be highly related to one another.

Cohesion Example Problem

Example class below is downloading from internet, parsing data and storing data to database. The responsibilities of this class are not really related. This is not cohesive class.

```
class DownloadAndStore{
    void downloadFromInternet(){
    }

    void parseData(){
    }

    void storeIntoDatabase(){
    }

    void doEverything(){
downloadFromInternet();
parseData();
storeIntoDatabase();
    }
```

```
}
```

Cohesion Example Solution

This is a better way of approaching the problem. Different classes have their own responsibilities.

```
class InternetDownloader {
    public void downloadFromInternet() {
    }
}

class DataParser {
    public void parseData() {
    }
}

class DatabaseStorer {
    public void storeIntoDatabase() {
    }
}

class DownloadAndStore {
    void doEverything() {
new InternetDownloader().downloadFromInternet();
new DataParser().parseData();
new DatabaseStorer().storeIntoDatabase();
    }
}
```

Encapsulation

- Encapsulation is hiding the implementation of a Class behind a well defined interfaceÓ. Encapsulation helps us to change implementation of a class without breaking other code.

Encapsulation Approach 1

In this approach we create a public variable score. The main method directly accesses the score variable, updates it.

Example Class

```
public class CricketScorer {
    public int score;
}
```

Let's use the CricketScorer class.

```
public static void main(String[] args) {
CricketScorer scorer = new CricketScorer();
scorer.score = scorer.score + 4;
}
```

Encapsulation Approach 2

In this approach, we make score as private and access value through get and set methods. However, the logic of adding 4 to the score is performed in the main method.

Example Class

```
public class CricketScorer {
    private int score;

    public int getScore() {
return score;
    }

    public void setScore(int score) {
this.score = score;
    }
}
```

Let's use the CricketScorer class.

```
public static void main(String[] args) {
CricketScorer scorer = new CricketScorer();

int score = scorer.getScore();
scorer.setScore(score + 4);
}
```

Encapsulation Approach 3

In this approach - For better encapsulation, the logic of doing the four operation also is moved to the CricketScorer class.

Example Class

```
public class CricketScorer {
    private int score;

    public void four() {
score += 4;
    }

}
```

Let's use the CricketScorer class.

```
public static void main(String[] args) {
  CricketScorer scorer = new CricketScorer();
  scorer.four();
}
```

Encapsulation Example

In terms of encapsulation Approach 3 > Approach 2 > Approach 1. In Approach 3, the user of scorer class does not even know that there is a variable called score. Implementation of Scorer can change without changing other classes using Scorer.

Interface

- An interface defines a contract for responsibilities (methods) of a class. Let's look at a few examples of interfaces.

Defining an Interface

An interface is declared by using the keyword interface. Look at the example below: Flyable is an interface.

```
//public abstract are not necessary
public abstract interface Flyable {
    //public abstract are not necessary
    public abstract void fly();
}
```

An interface can contain abstract methods -- NOT TRUE ANY MORE

In the above example, fly method is abstract since it is only declared (No definition is provided).

Implementing an Interface

We can define a class implementing the interface by using the implements keyword. Let us look at a couple of examples:

Example 1

Class Aeroplane implements Flyable and implements the abstract method fly().

```
public class Aeroplane implements Flyable{
    @Override
    public void fly() {
System.out.println("Aeroplane is flying");
    }
}
```

Example 2

```
public class Bird implements Flyable{
    @Override
    public void fly() {
System.out.println("Bird is flying");
    }
}
```

Using the Interface and Implementation

The interface classes can directly be instantiated and stored in the class reference variables

```
Bird bird = new Bird();
bird.fly();//Bird is flying

Aeroplane aeroplane = new Aeroplane();
aeroplane.fly();//Aeroplane is flying
```

An interface reference variable can hold objects of any implementation of interface.

```
Flyable flyable1 = new Bird();
Flyable flyable2 = new Aeroplane();
```

Variables in an interface

Variables in an interface are always public, static, final. Variables in an interface cannot be declared private.

```
interface ExampleInterface1 {
    //By default - public static final. No other modifier allowed
    //value1,value2,value3,value4 all are - public static final
    int value1 = 10;
    public int value2 = 15;
    public static int value3 = 20;
    public static final int value4 = 25;
    //private int value5 = 10;//COMPILER ERROR
}
```

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();
}
```

Class implementing SubInterface1 should implement both methods - method3 and method1(from ExampleInterface1) An interface cannot extend a class.

```
/* //COMPILE ERROR IF UnCommented
   //Interface cannot extend a Class
interface SubInterface2 extends Integer{
    void method3();
}
*/
```

A class can implement multiple interfaces. It should implement all the method declared in all Interfaces being implemented.

```
interface ExampleInterface2 {
    void method2();
}

class SampleImpl implements ExampleInterface1,ExampleInterface2{
    /* A class should implement all the methods in an interface.
       If either of method1 or method2 is commented, it would
       result in compilation error.
     */
    public void method2() {
System.out.println("Sample Implementation for Method2");
    }

    public void method1() {
System.out.println("Sample Implementation for Method1");
    }

}
```

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. No new checked exceptions can be thrown by implementations of methods in an interface.

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 Overloading Example 1

doIt method is overloaded in the below example:

```
class Foo{
    public void doIt(int number){

    }
    public void doIt(String string){

    }
}
```

Method Overloading Example 2

Overloading can also be done from a sub class.

```
class Bar extends Foo{
    public void doIt(float number){

    }
}
```

Overloading - Other Rules

An overloaded method should have different arguments than the original method. It can also have a different return type. A method cannot be overloaded just by only changing the return type. Overloaded methods are always treated as if they are different methods altogether. Overloading does not put any restrictions on access modifiers or exceptions thrown from the method. Overloaded method invocation is based on the Type of the Reference variable. It is NOT based on the object it refers to.

- Java Example
    - Constructors
    - public HashMap(int initialCapacity, float loadFactor)
    - public HashMap() {
    - public HashMap(int initialCapacity)
    - Methods

    - public boolean addAll(Collection<? extends E> c)
    - public boolean addAll(int index, Collection<? extends E> c)
- Rules

Method Overriding

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

Method Overriding Example 1:

Let's define an Animal class with a method shout.

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

Let's create a sub class of Animal , Cat - overriding the existing shout method in Animal.

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

bark method in Cat class is overriding the bark method in Animal class.

- Java Example
    - HashMap public int size() overrides AbstractMap public int size()
- Example

Overriding Method Cannot have lesser visibility

Overriding method cannot have lesser visibility than the Super Class method. Consider these two examples

Example 1

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

    }
}

class SubClass{
    //Cannot reduce visibility of SuperClass Method
    //So, only option is public
    public void publicMethod() {

    }
}
```

publicMethod in SubClass can only be declared as public. Keyword protected, private or (default) instead of public would result in Compilation Error.

Example 2

```
class SuperClass{
    void defaultMethod(){

    }
}

class SubClass{
    //Can be overridden with public,(default) or protected
    //private would give COMPILE ERROR!
    public void defaultMethod(){

    }

}
```

defaultMethod in SuperClass is declared with default access. Any method overriding it can have access default or greater. So default, protected and public are fine. Overriding method cannot be private.

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*/ {

    }
}
```

publicMethod() in SuperClass throws FileNotFoundException. So, the SubClass publicMethod() can throw FileNotFoundException or any sub class of FileNotFoundException. It can also not throw an Exception (as in the example). But, it cannot throw any new Exception. For example, Òpublic void publicMethod() throws IOExceptionÓ would cause compilation error.

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.

Overriding and Polymorphism Example

Overridden method invocation is based on the object referred to. It is not based on the Type of the Reference variable. This is called Polymorphism. Consider the example below:

```
class Animal{
    public void bark(){
System.out.println("Animal Bark");
    }
}

class Dog extends Animal{
    public void bark(){
System.out.println("Dog Bark");
    }
}

public class PolymorphismExample {
    public static void main(String[] args) {

Animal[] animals = {new Dog(),new Animal()};

animals[0].bark();//Dog bark
animals[1].bark();//Animal bark
    }
```

```
}
```

animals[0] contains a reference to Dog Object. When animals[0].bark() method is called method in Dog class is invoked even though the type of reference variable is Animal. animals[1] contains a reference to Animal Object. When animals[1].bark() method is called method in Animal class is invoked.

Covariant Returns

A sub class is considered to be of same type as its super class. So, in interfaces or abstract class, it is fine to provide implementations using the Sub Class Types as Return Types. (com.in28minutes.SameType)

Class Modifiers

- Let us learn about a few Java Class Modifiers.

Access Modifiers

Access modifier for a class can be public or (default), It cannot be private or protected.

```
public class PublicClass{

}

class DefaultClass{

}

protected class Error{//COMPILER ERROR

}

private class Error{//COMPILER ERROR

}
```

Non-access modifiers

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

Class Access Modifiers

- Lets learn about a few Java Class Access Modifiers.

public class modifier

A public class is visible to all other classes.

default class modifier

A class is called a Default Class is when there is no access modifier specified on a class. Default classes are visible inside the same package only. Default access is also called Package access.

Default Class Modifier Examples

Default Access Class Example

package com.in28minutes.classmodifiers.defaultaccess.a;

```
/* No public before class. So this class has default access*/
class DefaultAccessClass {
//Default access is also called package access
}
```

Another Class in Same Package: Has access to default class

```
package com.in28minutes.classmodifiers.defaultaccess.a;

public class AnotherClassInSamePackage {
    //DefaultAccessClass and AnotherClassInSamePackage
    //are in same package.
    //So, DefaultAccessClass is visible.
    //An instance of the class can be created.
    DefaultAccessClass defaultAccess;
}
```

Class in Different Package: NO access to default class

```
package com.in28minutes.classmodifiers.defaultaccess.b;

public class ClassInDifferentPackage {
    //Class DefaultAccessClass and Class ClassInDifferentPackage
    //are in different packages (*.a and *.b)
    //So, DefaultAccessClass is not visible to ClassInDifferentPackage

    //Below line of code will cause compilation error if uncommented
    //DefaultAccessClass defaultAccess; //COMPILE ERROR!!
}
```

Method and Variable Access Modifiers

- Method and variable access modifiers can be public, protected, private or (default)

Two Access Modifier Questions

When we talk about access modifiers, we would discuss two important questions

Is Accessible through reference/instance variable?

We create an instance of the class and try to access the variables and methods declared in the class.

```
ExampleClass example = new ExampleClass();

example.publicVariable = 5;
example.publicMethod();
```

Is Accessible through Inheritance?

Can we access the super class variables and methods from a Sub Class?

```
public class SubClass extends ExampleClass {
    void subClassMethod(){
publicVariable = 5;
protectedVariable = 5;
    }
}
```

Important Access Things to Remember

A sub class trying to access through reference/instance variables, will have the same access as a normal class (non sub class). Access modifiers cannot be applied to local variables

Access Modifiers Example

Let's consider the following class with variables and methods declared with all 4 access modifiers:

```
package com.in28minutes.membermodifiers.access;

public class ExampleClass {
    int defaultVariable;
    public int publicVariable;
    private int privateVariable;
    protected int protectedVariable;

    void defaultMethod(){

    }

    public void publicMethod(){

    }

    private void privateMethod(){

    }
```

```
    protected void protectedMethod(){

    }
}
```

Method Access Modifiers

Let's discuss about access modifiers in order of increasing access.

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

Access Modifier Example: Class in Same Package

Look at the code below to understand what can be accessed and what cannot be.

```
package com.in28minutes.membermodifiers.access;

public class TestClassInSamePackage {
    public static void main(String[] args) {
ExampleClass example = new ExampleClass();

example.publicVariable = 5;
example.publicMethod();

//privateVariable is not visible
//Below Line, uncommented, would give compiler error
//example.privateVariable=5; //COMPILE ERROR
//example.privateMethod();

example.protectedVariable = 5;
example.protectedMethod();
```

```
example.defaultVariable = 5;
example.defaultMethod();
    }
}
```

Access Modifier Example: Class in Different Package

Look at the code below to understand what can be accessed and what cannot be.

```
package com.in28minutes.membermodifiers.access.different;

import com.in28minutes.membermodifiers.access.ExampleClass;

public class TestClassInDifferentPackage {
    public static void main(String[] args) {
ExampleClass example = new ExampleClass();

example.publicVariable = 5;
example.publicMethod();

//privateVariable,privateMethod are not visible
//Below Lines, uncommented, would give compiler error
//example.privateVariable=5; //COMPILE ERROR
//example.privateMethod();//COMPILE ERROR

//protectedVariable,protectedMethod are not visible
//Below Lines, uncommented, would give compiler error
//example.protectedVariable = 5; //COMPILE ERROR
//example.protectedMethod();//COMPILE ERROR

//defaultVariable,defaultMethod are not visible
//Below Lines, uncommented, would give compiler error
//example.defaultVariable = 5;//COMPILE ERROR
//example.defaultMethod();//COMPILE ERROR
    }
}
```

Access Modifier Example: Sub Class in Same Package

Look at the code below to understand what can be accessed and what cannot be.

```
package com.in28minutes.membermodifiers.access;

public class SubClassInSamePackage extends ExampleClass {

    void subClassMethod(){
publicVariable = 5;
publicMethod();
```

```
//privateVariable is not visible to SubClass
//Below Line, uncommented, would give compiler error
//privateVariable=5; //COMPILE ERROR
//privateMethod();

protectedVariable = 5;
protectedMethod();

defaultVariable = 5;
defaultMethod();
    }
}
```

Access Modifier Example: Sub Class in Different Package

Look at the code below to understand what can be accessed and what cannot be.

```
package com.in28minutes.membermodifiers.access.different;

import com.in28minutes.membermodifiers.access.ExampleClass;

public class SubClassInDifferentPackage extends ExampleClass {

    void subClassMethod(){
publicVariable = 5;
publicMethod();

//privateVariable is not visible to SubClass
//Below Line, uncommented, would give compiler error
//privateVariable=5; //COMPILE ERROR
//privateMethod();

protectedVariable = 5;
protectedMethod();

//privateVariable is not visible to SubClass
//Below Line, uncommented, would give compiler error
//defaultVariable = 5; //COMPILE ERROR
//defaultMethod();
    }
}
```

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 public class FinalClass {
}
```

Below class will not compile if uncommented. FinalClass cannot be extended.

```
/*
class ExtendingFinalClass extends FinalClass{ //COMPILER ERROR

}
*/
```

Example of Final class in Java is the String class. Final is used very rarely as it prevents re-use of the class.

Final methods cannot be overriden.

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

```
public class FinalMemberModifiersExample {
    final void finalMethod(){
    }
}
```

Any SubClass extending above class cannot override the finalMethod().

```
class SubClass extends FinalMemberModifiersExample {
    //final method cannot be over-riddent
    //Below method, uncommented, causes compilation Error
    /*
    final void finalMethod(){

    }
    */
}
```

Final variable values cannot be changed.

Once initialized, the value of a final variable cannot be changed.

```
final int finalValue = 5;
//finalValue = 10; //COMPILER ERROR
```

Final arguments value cannot be modified.

Consider the example below:

```
void testMethod(final int finalArgument){
    //final argument cannot be modified
    //Below line, uncommented, causes compilation Error
    //finalArgument = 5;//COMPILER ERROR
}
```

Other Non access Modifiers

- A class cannot be both abstract and final

strictfp

This modifier can be used on a class and a method. This (strictfp) cannot be used on a variable. IEEE standard for floating points would be followed in the method or class where strictfp modifier is specified.

volatile

Volatile can only be applied to instance variables. A volatile variable is one whose value is always written to and read from "main memory". Each thread has its own cache in Java. The volatile variable will not be stored on a Thread cache.

native

Can be applied only to methods. These methods are implemented in native languages (like C)

Static Variables and Methods

- Static variables and methods are class level variables and methods. There is only one copy of the static variable for the entire Class. Each instance of the Class (object) will NOT have a unique copy of a static variable. Let's start with a real world example of a Class with static variable and methods.

Static Variable/Method , Example

count variable in Cricketer class is static. The method to get the count value getCount() is also a static method.

```
public class Cricketer {
    private static int count;

    public Cricketer() {
count++;
    }

    static int getCount() {
return count;
    }

    public static void main(String[] args) {
```

```
Cricketer cricketer1 = new Cricketer();
Cricketer cricketer2 = new Cricketer();
Cricketer cricketer3 = new Cricketer();
Cricketer cricketer4 = new Cricketer();

System.out.println(Cricketer.getCount());//4
    }
}
```

4 instances of the Cricketer class are created. Variable count is incremented with every instance created in the constructor.

Static Variables and Methods Example 2

Example class below explains all the rules associated with access static variables and static methods.

```
public class StaticModifierExamples {
    private static int staticVariable;
    private int instanceVariable;

    public static void staticMethod() {
//instance variables are not accessible in static methods
//instanceVariable = 10; //COMPILER ERROR

//Also, this Keyword is not accessible. this refers to object.
//static methods are class methods

//static variables are accessible in static methods
staticVariable = 10;
    }

    public void instanceMethod() {
//static and instance variables are accessible in instance methods
instanceVariable = 10;
staticVariable = 10;
    }

    public static void main(String[] args) {
//static int i =5; //COMPILER ERROR
StaticModifierExamples example = new StaticModifierExamples();

//instance variables and methods are only accessible through object references
example.instanceVariable = 10;
example.instanceMethod();
//StaticModifierExamples.instanceVariable = 10;//COMPILER ERROR
//StaticModifierExamples.instanceMethod();//COMPILER ERROR
```

```
//static variables and methods are accessible through object references and
Class Name.
example.staticVariable = 10;
example.staticMethod();
StaticModifierExamples.staticVariable = 10;
StaticModifierExamples.staticMethod();

    }
}
```

In a static method, instance variables are not accessible. Keyword this is also not accessible. However static variables are accessible.

```
    public static void staticMethod() {
//instance variables are not accessible in static methods
//instanceVariable = 10; //COMPILER ERROR

//Also, this Keyword is not accessible. this refers to object.
//static methods are class methods

//static variables are accessible in static methods
staticVariable = 10;
    }
```

In instance methods, both static and instance variables are accessible.

```
    public void instanceMethod() {
instanceVariable = 10;
staticVariable = 10;
    }
```

Instance variables and methods are only accessible through object references.

```
example.instanceVariable = 10;
example.instanceMethod();
//StaticModifierExamples.instanceVariable = 10;//COMPILER ERROR
//StaticModifierExamples.instanceMethod();//COMPILER ERROR
```

Static variables and methods are accessible through object references and Class Name.

```
example.staticVariable = 10;
example.staticMethod();
StaticModifierExamples.staticVariable = 10;
StaticModifierExamples.staticMethod();
```

It is always recommended to use Class Name to access a static variable or method. This is because static methods are class level methods. It is not appropriate to use instance references to call static methods (even though it compiles and works).

Static methods cannot be overridden

Consider the example below:

```
class Animal{
    static void StaticMethod(){
System.out.println("Animal Static Method");
    }
}

class Dog extends Animal{
    static void StaticMethod(){
System.out.println("Dog Static Method");
    }
}
```

When code below is run, static method in Animal is executed. Static method invocation is based on the type of reference variable. It does not depend on the type of object referred to.

```
Animal animal = new Dog();
animal.StaticMethod();//Animal Static Method
```

Local variables cannot be declared as static

Example below:

```
public static void main(String[] args) {
    //static int i =5; //COMPILER ERROR
}
```

Class Contents

- Let's discuss what a Java class can contain and what it cannot.

Section

Java Source File Rules A Java Source File can contain a.0 or 1 Public Classes b.0 or 1 or More Non Public Classes

Order should be

1. Package Statement
2. Imports
3. Class Declarations

Comments can be anywhere in the file. If there is a public class, file name should the (name of public class) + ".java". If name of public class is Scorer, name of file should be Scorer.java. If there is no public class, there are no restrictions on file name.

Example Class

```
/* Comments Anywhere*/
package com.in28minutes.classcontent;
class DefaultClass1{

}
/* Comments Anywhere*/
class DefaultClass2{

}
/* Comments Anywhere*/
public class PublicClass1 {
}

/* Cannot have another Public Class.
public class PublicClass2 {
}
*/
```

Nested Class

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

Nested Class Example

Consider the following example:

```
class OuterClass {

    public class InnerClass {
    }

    public static class StaticNestedClass {
    }

    public void exampleMethod() {
class MethodLocalInnerClass {
};
    }

}
```

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

- Inner Class is a very important Java Concept. Let's learn about it in this tutorial.

Inner Class Example

Consider the following Example:

```
class OuterClass {
    private int outerClassInstanceVariable;

    public class InnerClass {
private int innerClassVariable;

public int getInnerClassVariable() {
    return innerClassVariable;
}

public void setInnerClassVariable(
int innerClassVariable) {
    this.innerClassVariable = innerClassVariable;
}

public void privateVariablesOfOuterClassAreAvailable() {
    outerClassInstanceVariable = 5; // we can access the value
    System.out.println("Inner class ref is " + this);

    //Accessing outer class reference variable
    System.out.println("Outer class ref is " + OuterClass.this);
}
    }

    public void createInnerClass(){
//Just use the inner class name to create it
InnerClass inner = new InnerClass();
    }
```

```
}

public class InnerClassExamples {
    public static void main(String[] args) {

// COMPILER ERROR! You cannot create an inner class on its own.
// InnerClass innerClass = new InnerClass();
OuterClass example = new OuterClass();

// To create an Inner Class you need an instance of Outer Class
OuterClass.InnerClass innerClass = example.new InnerClass();

    }
}
```

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();
```

Creating an Inner Class instance in outer class

Consider the method createInnerClass from the example above: This method shows how to create an inner class instance.

```
public void createInnerClass(){
    //Just use the inner class name to create it
    InnerClass inner = new InnerClass();
}
```

Instance variables of Outer Class are available in inner class

Consider the method privateVariablesOfOuterClassAreAvailable from InnerClass declared above:

```
public void privateVariablesOfOuterClassAreAvailable() {
    outerClassInstanceVariable = 5; // we can access the value
    System.out.println("Inner class ref is " + this);

    //Accessing outer class reference variable
    System.out.println("Outer class ref is " + OuterClass.this);
}
```

Static Inner Nested Class

- Let's learn about Static Inner Nested Class in this Java tutorial.

Static Inner Nested Class Example

Consider the example below:

```
class OuterClass {
    private int outerClassInstanceVariable;

    public static class StaticNestedClass {
private int staticNestedClassVariable;

public int getStaticNestedClassVariable() {
    return staticNestedClassVariable;
}

public void setStaticNestedClassVariable(
int staticNestedClassVariable) {
    this.staticNestedClassVariable = staticNestedClassVariable;
}

public void privateVariablesOfOuterClassAreNOTAvailable() {
    // outerClassInstanceVariable = 5; //COMPILE ERROR
}
    }

}

public class InnerClassExamples {
    public static void main(String[] args) {
// Static Nested Class can be created without needing to create its
// parent. Without creating NestedClassesExample, we created
// StaticNestedClass
OuterClass.StaticNestedClass staticNestedClass1 = new
OuterClass.StaticNestedClass();
staticNestedClass1.setStaticNestedClassVariable(5);

OuterClass.StaticNestedClass staticNestedClass2 = new
OuterClass.StaticNestedClass();
staticNestedClass2.setStaticNestedClassVariable(10);

// 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
    }
}
```

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.

```
public void privateVariablesOfOuterClassAreNOTAvailable() {
    // outerClassInstanceVariable = 5; //COMPILE ERROR
}
```

Method Inner Class

- Let us learn about Method Inner Class in this tutorial.

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

Look at the commented code below exampleMethod. MethodLocalInnerClass can be instantiated only in the method where it is declared.

Method inner class can access class instance variables

```
//Can access class instance variables
System.out.println(outerClassInstanceVariable);
```

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

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

Variable Arguments

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

Variable Arguments Example

```
    //int(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.
//number can be treated as if it is declared as int[] numbers;
int sum = 0;
for (int number: numbers) {
    sum += number;
}
return sum;
    }

    public static void main(String[] args) {
```

```
VariableArgumentExamples example = new VariableArgumentExamples();
//3 Arguments
System.out.println(example.sum(1, 4, 5));//10
//4 Arguments
System.out.println(example.sum(1, 4, 5, 20));//30
//0 Arguments
System.out.println(example.sum());//0
    }
```

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. Below example gives a compilation error

```
    public int sum(int... numbers, float value) {//COMPILER ERROR
    }
```

Variable Argument of Type Custom Class

Even a class can be used a variable argument. In the example below, bark method is overloaded with a variable argument method.

```
    class Animal {
void bark() {
    System.out.println("Bark");
}
void bark(Animal... animals) {
    for (Animal animal: animals) {
animal.bark();
    }
}
    }
```

Exception Handling

- In this tutorial, let's understand the need for exception handling and learn how to handle exceptions.

Example without Exception Handling

Let's first look an example without exception handling. Method main throws an exception because toString method is invoked on a null object.

```
    public static void main(String[] args) {
String str = null;
str.toString();
    }
```

Output of above program is

```
Exception in thread "main" java.lang.NullPointerException at
com.in28minutes.exceptionhandling.ExceptionHandlingExample1.main(ExceptionHand
lingExample1.java:6)
```

Exception Example 2 , Propagation of an Exception

In this example, main invokes method1, which invokes method2 which throws a NullPointerException. Check the output of this program.

```
    public static void main(String[] args) {
method1();
    }

    private static void method1() {
method2();
    }

    private static void method2() {
String str = null;
str.toString();
```

```
    }
//Output - Exception in thread "main" java.lang.NullPointerException at
com.in28minutes.exceptionhandling.ExceptionHandlingExample1.method2(ExceptionH
andlingExample1.java:15)
at
com.in28minutes.exceptionhandling.ExceptionHandlingExample1.method1(ExceptionH
andlingExample1.java:10)
at
com.in28minutes.exceptionhandling.ExceptionHandlingExample1.main(ExceptionHand
lingExample1.java:6)
```

Look at the stack trace. Exception which is thrown in method2 is propagating to method1 and then to main. This is because there is no exception handling in all 3 methods - main, method1 and method2

Exception Example 3: Execution of method stopped

Look at the example below: A println method call is added after every method call.

```
    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() {
String str = null;
str.toString();
System.out.println("Line after Exception - Method 2");
    }
//Output - Exception in thread "main" java.lang.NullPointerException
at
com.in28minutes.exceptionhandling.ExceptionHandlingExample1.method2(ExceptionH
andlingExample1.java:18)
at
com.in28minutes.exceptionhandling.ExceptionHandlingExample1.method1(ExceptionH
andlingExample1.java:12)
at
com.in28minutes.exceptionhandling.ExceptionHandlingExample1.main(ExceptionHand
lingExample1.java:7)
```

Note that none of the lines with text "Line after Exception - " are executed. If an exception occurs, lines after the line where exception occurred are not executed. Since all three methods main, method1() and method2() do not have any Exception Handling, exception propagates from method2 to method1 to main.

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 5: Need for Finally

Consider the example below: In method2, a connection is opened. However, because of the exception thrown, connection is not closed. This results in unclosed connections.

```
package com.in28minutes.exceptionhandling;

class Connection {
```

```
    void open() {
System.out.println("Connection Opened");
    }

    void close() {
System.out.println("Connection Closed");
    }
}

public class ExceptionHandlingExample1 {

    // Exception Handling Example 1
    // 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 {
    Connection connection = new Connection();
    connection.open();

    // LOGIC
    String str = null;
    str.toString();

    connection.close();
} catch (Exception e) {
    // NOT PRINTING EXCEPTION TRACE- BAD PRACTICE
    System.out.println("Exception Handled - Method 2");
}
    }
}
```

Output

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

Connection that is opened is not closed. Because an exception has occurred in method2, connection.close() is not run. This results in a dangling (un-closed) connection.

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.

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

} catch (Exception e) {
    // NOT PRINTING EXCEPTION TRACE - BAD PRACTICE
    System.out.println("Exception Handled - Method 2");
} finally {
    connection.close();
}
    }
```

Output

```
Connection Opened
Exception Handled - Method 2
Connection Closed
Line after Exception - Method 1
Line after Exception - Main
```

Connection is closed even when exception is thrown. This is because connection.close() is called in the finally block. Finally block is always executed (even when an exception is thrown). So, if we want some code to be always executed we can move it to finally block.

Code in finally is NOT executed only in two situations. If exception is thrown in finally. If JVM Crashes in between (for example, System.exit()).

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

```
private static void method2() {
Connection connection = new Connection();
connection.open();
try {
    // LOGIC
    String str = null;
    str.toString();
    return;
} catch (Exception e) {
```

```
    // NOT PRINTING EXCEPTION TRACE - BAD PRACTICE
    System.out.println("Exception Handled - Method 2");
    return;
} finally {
    connection.close();
}
    }
```

Exception Handling Syntax

Let's look at a few quirks about Exception Handling syntax.

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();
}
    }
```

Output:

```
Connection Opened
Connection Closed
Exception in thread "main" java.lang.NullPointerException at
com.in28minutes.exceptionhandling.ExceptionHandlingExample1.method2(ExceptionH
andlingExample1.java:33) at
com.in28minutes.exceptionhandling.ExceptionHandlingExample1.method1(ExceptionH
andlingExample1.java:22) at
com.in28minutes.exceptionhandling.ExceptionHandlingExample1.main(ExceptionHand
lingExample1.java:17)
```

Try without a catch is useful when you would want to do something (close a connection) even if an exception occurred without handling the exception.

Try without both catch and finally is not allowed.

Below method would give a Compilation Error!! (End of try block)

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

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{}
```

Below class definitions show creation of a programmer defined exception in Java.

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

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

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. For Example: RuntimeException,UnCheckedException,UnCheckedException2 are unchecked or RunTime Exceptions. There are subclasses of RuntimeException (which means they are subclasses of Exception also.)

Checked Exception

Other Exception Classes (which don't fit the earlier definition). These are also called Checked Exceptions. Exception, CheckedException1,CheckedException2 are checked exceptions. They are subclasses of Exception which are not subclasses of RuntimeException.

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(ExceptionHandlingExam
ple2.java:17)
at
com.in28minutes.exceptionhandling.ExceptionHandlingExample2.main(ExceptionHand
lingExample2.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

Throws Exception Example

Let's look at how to declare throwing an exception from a method.

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

Look at the line "static Amount addAmounts(Amount amount1, Amount amount2) throws Exception". This is how we declare that a method throws Exception. This results in compilation error in main method. This is because Main method is calling a method which is declaring that it might throw Exception. Main method again has two options a. Throw b. Handle

Code with main method throwing the exception below

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

Output

```
Exception in thread "main" java.lang.Exception: Currencies don't match
at
com.in28minutes.exceptionhandling.AmountAdder.addAmounts(ExceptionHandlingExam
ple2.java:17)
at
com.in28minutes.exceptionhandling.ExceptionHandlingExample2.main(ExceptionHand
lingExample2.java:28)
```

Handling an Exception

main can also handle the exception instead of declaring throws. Code for it below.

```
public static void main(String[] args) {
try {
    AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR",5));
} catch (Exception e) {
    System.out.println("Exception Handled in Main");
}
    }
```

Output

```
Exception Handled in Main
```

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());
}
    }
}
```

Output:

```
Exception Handled in Mainclass
com.in28minutes.exceptionhandling.CurrenciesDoNotMatchException
```

Let's change main method to handle Exception instead of CurrenciesDoNotMatchException

```
public class ExceptionHandlingExample2 {
    public static void main(String[] args) {
try {
    AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR",5));
} catch (Exception e) {
    System.out.println("Exception Handled in Main" + e.getClass());
}
    }
}
```

Output:

```
Exception Handled in Mainclass
com.in28minutes.exceptionhandling.CurrenciesDoNotMatchException
```

There is no change in output from the previous example. This is because Exception catch block can catch Exception and all subclasses of Exception.

Extend RuntimeException

Let's change the class CurrenciesDoNotMatchException to extend RuntimeException instead of Exception

```
class CurrenciesDoNotMatchException extends RuntimeException{
}
```

Output:

```
Exception Handled in Mainclass
com.in28minutes.exceptionhandling.CurrenciesDoNotMatchException
```

Change methods addAmounts in AmountAdder to remove the declaration " throws CurrenciesDoNotMatchException"

No compilation error occurs since RuntimeException and subclasses of RuntimeException are not Checked Exception's. So, they don't need to be handled or declared. If you are interested in handling them, go ahead and handle them. But, java does not require you to handle them.

Remove try catch from main method. It is not necessary since CurrenciesDoNotMatchException is now a RuntimeException.

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

Output:

```
Exception in thread "main"
com.in28minutes.exceptionhandling.CurrenciesDoNotMatchException at
com.in28minutes.exceptionhandling.AmountAdder.addAmounts(ExceptionHandlingExam
ple2.java:21)
at
com.in28minutes.exceptionhandling.ExceptionHandlingExample2.main(ExceptionHand
lingExample2.java:30)
```

Multiple Catch Blocks

Now, let's add two catch blocks to the main

```
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("Handled CurrenciesDoNotMatchException");
} catch (Exception e) {
    System.out.println("Handled Exception");
}
    }
}
```

Output:

```
Handled CurrenciesDoNotMatchException
```

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. For example, CurrenciesDoNotMatchException should be before Exception. Below code gives a compilation error.

```
    public static void main(String[] args) {
try {
    AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR",
    5));
} catch (Exception e) { // COMPILER ERROR!!
    System.out.println("Handled Exception");
} catch (CurrenciesDoNotMatchException e) {
    System.out.println("Handled CurrenciesDoNotMatchException");
}
    }
```

Catch block handles only specified Exceptions (and sub types)

A catch block of type ExceptionType can only catch types ExceptionType and sub classes of ExceptionType. For Example: Let us change the main method code as shown below. Main method throws a NullPointerException.

```
    public static void main(String[] args) {
try {
    AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("RUPEE",
    5));
    String string = null;
    string.toString();
} catch (CurrenciesDoNotMatchException e) {
    System.out.println("Handled CurrenciesDoNotMatchException");
}
    }

  //Output : Exception in thread "main" java.lang.NullPointerException at
com.in28minutes.exceptionhandling.ExceptionHandlingExample2.main(ExceptionHand
lingExample2.java:34)
```

Since NullPointerException is not a sub class of CurrenciesDoNotMatchException it wouldn't be handled by the catch block. Instead a NullPointerException would be thrown out by the main method.

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.

```
    public static void main(String[] args) {
try {
    AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("RUPEE",
    5));
    String string = null;
    string.toString();
} catch (CurrenciesDoNotMatchException e) {
    System.out.println("Handled CurrenciesDoNotMatchException");
    e.printStackTrace();
}
    }
```

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
```

For floating point numbers, precision can be specified after dot(.). Below example uses a precision of 2, so .5678 gets changed to .57

```
System.out.printf("%5.2f", 1234.5678); //prints 1234.57
```

An error in specifying would give a RuntimeException. In below example a string is passed to %d argument.

```
System.out.printf("%5d","Test");
//Throws java.util.IllegalFormatConversionException

//To change the order of printing and passing of arguments, argument index can be used
System.out.printf("%3$.1f %2$s %1$d", 123, "Test", 123.4); //prints 123.4 Test 123

//format method has the same behavior as printf method
System.out.format("%5.2f", 1234.5678);//prints 1234.57
```

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.

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.

Creating Date Object

```
//Creating Date Object
Date now = new Date();
System.out.println(now.getTime());
```

Manipulating Date Object

Lets now look at adding a few hours to a date object. All date manipulation to date needs to be done by adding milliseconds to the date. For example, if we want to add 6 hour, we convert 6 hours into millseconds. 6 hours = 6 * 60 * 60 * 1000 milliseconds. Below examples shows specific code.

```
Date date = new Date();

//Increase time by 6 hrs
date.setTime(date.getTime() + 6 * 60 * 60 * 1000);
System.out.println(date);

//Decrease time by 6 hrs
date = new Date();
date.setTime(date.getTime() - 6 * 60 * 60 * 1000);
System.out.println(date);
```

Formatting Dates

Formatting Dates is done by using DateFormat class. Let's look at a few examples.

```
//Formatting Dates
System.out.println(DateFormat.getInstance().format(
date));//10/16/12 5:18 AM
```

Formatting Dates with a locale

```
System.out.println(DateFormat.getDateInstance(
DateFormat.FULL, new Locale("it", "IT"))
.format(date));//martedÒ 16 ottobre 2012

System.out.println(DateFormat.getDateInstance(
DateFormat.FULL, Locale.ITALIAN)
.format(date));//martedÒ 16 ottobre 2012

//This uses default locale US
System.out.println(DateFormat.getDateInstance(
DateFormat.FULL).format(date));//Tuesday, October 16, 2012

System.out.println(DateFormat.getDateInstance()
.format(date));//Oct 16, 2012
System.out.println(DateFormat.getDateInstance(
DateFormat.SHORT).format(date));//10/16/12
System.out.println(DateFormat.getDateInstance(
DateFormat.MEDIUM).format(date));//Oct 16, 2012

System.out.println(DateFormat.getDateInstance(
DateFormat.LONG).format(date));//October 16, 2012
```

Format Date's using SimpleDateFormat

Let's look at a few examples of formatting dates using SimpleDateFormat.

```
System.out.println(new SimpleDateFormat("yy-MM-dd")
```

```
.format(date));//12-10-16
System.out
.println(new SimpleDateFormat("yy-MMM-dd")
.format(date));//12-Oct-16
System.out.println(new SimpleDateFormat(
"yyyy-MM-dd").format(date));//2012-10-16

//Parse Dates using DateFormat
Date date2 = DateFormat.getDateInstance(
DateFormat.SHORT).parse("10/16/12");
System.out.println(date2);//Tue Oct 16 00:00:00 GMT+05:30 2012

//Creating Dates using SimpleDateFormat
Date date1 = new SimpleDateFormat("yy-MM-dd")
.parse("12-10-16");
System.out.println(date1);//Tue Oct 16 00:00:00 GMT+05:30 2012
```

Default Locale

```
Locale defaultLocale = Locale.getDefault();

System.out.println(defaultLocale
.getDisplayCountry());//United States
System.out.println(defaultLocale
.getDisplayLanguage());//English
```

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);
```

Calendar 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
```

Creating calendar: Example 2

```
Calendar gregorianCalendar = new GregorianCalendar(
2011, 7, 15);
```

Formatting Calendar object.

Done by getting the date using calendar.getTime() and using the usual formatting of dates.

```
System.out.println(DateFormat.getInstance().format(
calendar.getTime()));//3/29/12 11:39 AM
```

Number Format

- Number format is used to format a number to different locales and different formats.

Format number Using Default locale

```
System.out.println(NumberFormat.getInstance().format(321.24f));//321.24
```

Format number using locale

Formatting a number using Netherlands locale

```
System.out.println(NumberFormat.getInstance(new
Locale("nl")).format(4032.3f));//4.032,3
```

Formatting a number using Germany locale

```
System.out.println(NumberFormat.getInstance(Locale.GERMANY).format(4032.3f));/
/4.032,3
```

Formatting a Currency using Default locale

```
System.out.println(NumberFormat.getCurrencyInstance().format(40324.31f));//$40
,324.31
```

Format currency using locale

```
formatting a Currency using Netherlands locale
System.out.println(NumberFormat.getCurrencyInstance(new
Locale("nl")).format(40324.31f));//? 40.324,31
```

Setting maximum fraction digits for a float

```
numberFormat numberFormat = NumberFormat.getInstance();
System.out.println(numberFormat.getMaximumFractionDigits());//3
numberFormat.setMaximumFractionDigits(5);
System.out.println(numberFormat.format(321.24532f));//321.24533
```

Parsing using NumberFormat

Parsing a float value using number format

```
System.out.println(numberFormat.parse("9876.56"));//9876.56
```

Parsing only number value using number format

```
numberFormat.setParseIntegerOnly(true);
System.out.println(numberFormat.parse("9876.56"));//9876
```

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

Most important methods declared in the collection interface are the methods to add and remove an element. add method allows adding an element to a collection and delete method allows deleting an element from a collection. size() methods returns number of elements in the collection. Other important methods defined as part of collection interface are shown below.

```
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<Integer> integers = new ArrayList<Integer>();
```

Code like below is permitted because of auto boxing. 5 is auto boxed into Integer object and stored in ArrayList.

```
Integers.add(5);//new Integer(5)
```

Add method (by default) adds the element at the end of the list.

ArrayList of String Example

Below example shows how to create and use a String ArrayList. ArrayList can have duplicates (since List can have duplicates). size() method gets number of elements in the ArrayList. contains(Object) method checks if an element is present in the arraylist.

```
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.

```
//example1 - value is present
System.out.println(arraylist.indexOf("Dravid"));//2
//example2 - value is not present
System.out.println(arraylist.indexOf("Bradman"));//-1
```

get() function - get value at specified index.

```
System.out.println(arraylist.get(1));//Sachin
```

remove() function

remove() function has two variations.

```
//Using the object as parameter
//Dravid is removed from the list
arraylist.remove("Dravid");//[Ganguly, Sachin, Sachin]

//Using index as argument.
//Object at index 1 (Sachin) is removed
arraylist.remove(1);//[Ganguly, Sachin]
```

Sorting Collections

```
List<String> numbers = new ArrayList<String>();
numbers.add("one");
numbers.add("two");
numbers.add("three");
numbers.add("four");
System.out.println(numbers);//[one, two, three, four]

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

System.out.println(numbers);//[four, one, three, two]
```

List of Objects of a Custom Class

Consider the following class Cricketer.

```
class Cricketer{
    int runs;
    String name;

    public Cricketer(String name, int runs) {
super();
this.name = name;
this.runs = runs;
    }

    @Override
    public String toString() {
return name + " " + runs;
    }
}
```

Let's now try to sort a list containing objects of Cricketer class.

```
List<Cricketer> cricketers = new ArrayList<Cricketer>();
cricketers.add(new Cricketer("Bradman", 9996));
cricketers.add(new Cricketer("Sachin", 14000));
cricketers.add(new Cricketer("Dravid", 12000));
cricketers.add(new Cricketer("Ponting", 11000));
System.out.println(cricketers);
//[Bradman 9996, Sachin 14000, Dravid 12000, Ponting 11000]

//Cricketer class does not implement Comparable Interface
//Collections.sort(cricketers); //COMPILER ERROR
```

We get a compiler error since Cricketer class does not implement Comparable interface. We were able to sort numbers in earlier example because String class implements Comparable. Let's make the Cricketer class implement the Comparable Interface.

```
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) {
if (this.runs > that.runs) {
    return 1;
}
if (this.runs < that.runs) {
    return -1;
}
return 0;
    }

}
```

Now let's try to sort the cricketers.

```
Collections.sort(cricketers);
System.out.println(cricketers);
//[Bradman 9996, Ponting 11000, Dravid 12000, Sachin 14000]
```

Other option to sort collections is by creating a separate class which implements Comparator interface. Example below:

```
class DescendingSorter implements Comparator<Cricketer> {

    //compareTo returns -1 if cricketer1 < cricketer2
```

```
    //   1 if cricketer1 > cricketer2
    //   0 if cricketer1 = cricketer2

    //Since we want to sort in descending order,
    //we should return -1 when runs are more
    @Override
    public int compare(Cricketer cricketer1,
    Cricketer cricketer2) {
if (cricketer1.runs > cricketer2.runs) {
    return -1;
}
if (cricketer1.runs < cricketer2.runs) {
    return 1;
}
return 0;
    }

}
```

Let's now try to sort the previous defined collection:

```
Collections
.sort(cricketers, new DescendingSorter());

System.out.println(cricketers);
//[Sachin 14000, Dravid 12000, Ponting 11000, Bradman 9996]
```

Convert List to Array

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

```
List<String> numbers1 = new ArrayList<String>();
numbers1.add("one");
numbers1.add("two");
numbers1.add("three");
numbers1.add("four");
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]
```

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.

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. Let's look at examples of these collection classes.

HashSet

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

HashSet Example

```
Set<String> hashset = new HashSet<String>();

hashset.add("Sachin");
System.out.println(hashset);//[Sachin]

hashset.add("Dravid");
System.out.println(hashset);//[Sachin, Dravid]
```

Let's try to add Sachin to the Set now. Sachin is Duplicate. So will not be added. returns false.

```
hashset.add("Sachin");//returns false since element is not added
System.out.println(hashset);//[Sachin, Dravid]
```

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.

```
Set<String> treeSet = new TreeSet<String>();

treeSet.add("Sachin");
System.out.println(treeSet);//[Sachin]
```

Notice that the list is sorted after inserting Dravid.

```
//Alphabetical order
treeSet.add("Dravid");
System.out.println(treeSet);//[Dravid, Sachin]
```

Notice that the list is sorted after inserting Ganguly.

```
treeSet.add("Ganguly");
System.out.println(treeSet);//[Dravid, Ganguly, Sachin]

//Sachin is Duplicate. So will not be added. returns false.
treeSet.add("Sachin");//returns false since element is not added
System.out.println(treeSet);//[Dravid, Ganguly, Sachin]
```

Objects that are inserted into a TreeSet should be comparable.

TreeSet - NavigableSet interface examples 1

TreeSet implements this interface. Let's look at an example with TreeSet. Note that elements in TreeSet are sorted.

```
TreeSet<Integer> numbersTreeSet = new TreeSet<Integer>();
numbersTreeSet.add(55);
numbersTreeSet.add(25);
numbersTreeSet.add(35);
numbersTreeSet.add(5);
numbersTreeSet.add(45);
```

NavigableSet interface has following methods. Lower method finds the highest element lower than specified element. Floor method finds the highest element lower than or equal to specified element. Corresponding methods for finding lowest number higher than specified element are higher and ceiling. A few examples using the Set created earlier below.

```
//Find the highest number which is lower than 25
System.out.println(numbersTreeSet.lower(25));//5

//Find the highest number which is lower than or equal to 25
System.out.println(numbersTreeSet.floor(25));//25

//Find the lowest number higher than 25
System.out.println(numbersTreeSet.higher(25));//35

//Find the lowest number higher than or equal to 25
System.out.println(numbersTreeSet.ceiling(25));//25
```

NavigableSet subSet,headSet,tailSet Methods in TreeSet

```
TreeSet<Integer> exampleTreeSet = new TreeSet<Integer>();
exampleTreeSet.add(55);
exampleTreeSet.add(25);
exampleTreeSet.add(105);
exampleTreeSet.add(35);
exampleTreeSet.add(5);

System.out.println(exampleTreeSet);//[5, 25, 35, 55, 105]
```

All the three methods - subSet,headSet,tailSet - are inclusive with Lower Limit and NOT inclusive with higher limit. In the sub set below, Lower Limit is inclusive - 25 included. Higher limit is not inclusive - 55 excluded.

```
//Get sub set with values >=25 and <55
SortedSet<Integer> subTreeSet = exampleTreeSet
.subSet(25, 55);
System.out.println(subTreeSet);//[25, 35]
```

In the sub set below, Higher limit not inclusive - 55 excluded.

```
//Get sub set with values <55
SortedSet<Integer> headTreeSet = exampleTreeSet
.headSet(55);
System.out.println(headTreeSet);//[5, 25, 35]

//Get sub set with values >=35
SortedSet<Integer> tailTreeSet = exampleTreeSet
.tailSet(35);
```

```
System.out.println(tailTreeSet);//[35, 55, 105]
//In the sub set, Lower limit inclusive - 35 included.

//Get sub set with value >=25 and <=55 (both inclusive parameters - true)
SortedSet<Integer> subTreeSetIncl = exampleTreeSet
.subSet(25, true, 55, true);
System.out.println(subTreeSetIncl);//[25, 35, 55]

//Get sub set with value >25 and <55 (both inclusive parameters - false)
SortedSet<Integer> subTreeSetNotIncl = exampleTreeSet
.subSet(25, false, 55, false);
System.out.println(subTreeSetNotIncl);//[35]

//Get sub set with values <=55. Inclusive set to true.
SortedSet<Integer> headTreeSetIncl = exampleTreeSet
.headSet(55, true);
System.out.println(headTreeSetIncl);//[5, 25, 35, 55]

//Get sub set with values >35. Inclusive set to false.
SortedSet<Integer> tailTreeSetNotIncl = exampleTreeSet
.tailSet(35, false);
System.out.println(tailTreeSetNotIncl);//[55, 105]
```

All the sub set methods - subSet,headSet,tailSet - return dynamic sub sets. When original set is modified (addition or deletion), corresponding changes can affect the sub sets as well.

```
System.out.println(exampleTreeSet);//[5, 25, 35, 55, 105]
System.out.println(subTreeSet);//[25, 35]
System.out.println(headTreeSet);//[5, 25, 35]
System.out.println(tailTreeSet);//[35, 55, 105]
```

Let's now insert a value 30 into the exampleTreeSet. Remember that subTreeSet, headTreeSet, tailTreeSet are sub sets of exampleTreeSet.

```
exampleTreeSet.add(30);

System.out.println(exampleTreeSet);//[5, 25, 30, 35, 55, 105]
System.out.println(subTreeSet);//[25, 30, 35]
System.out.println(headTreeSet);//[5, 30, 25, 35]
System.out.println(tailTreeSet);//[35, 55, 105]
```

30 is in the range of subTreeSet and headTreeSet. So, it is printed as part of exampleTreeSet, subTreeSet and headTreeSet.

```
//Let's now add 65 to the set
exampleTreeSet.add(65);

System.out.println(exampleTreeSet);//[5, 25, 30, 35, 55, 65, 105]
System.out.println(subTreeSet);//[25, 30, 35]
System.out.println(headTreeSet);//[5, 30, 25, 35]
System.out.println(tailTreeSet);//[35, 55, 65, 105]
```

65 is printed as part of exampleTreeSet and tailTreeSet. Key thing to remember is that all the sub sets are dynamic. If you modify the original set,the sub set might be affected.

TreeSet - NavigableSet interface methods - pollFirst, pollLast and more

```
TreeSet<Integer> treeSetOrig = new TreeSet<Integer>();
treeSetOrig.add(55);
treeSetOrig.add(25);
treeSetOrig.add(35);
treeSetOrig.add(5);

System.out.println(treeSetOrig);//[5, 25, 35, 55]
//descendingSet method returns the tree set in reverse order
TreeSet<Integer> treeSetDesc = (TreeSet<Integer>) treeSetOrig
.descendingSet();
System.out.println(treeSetDesc);//[55, 35, 25, 5]
```

pollFirst returns the first element and removes it from the set.

```
System.out.println(treeSetOrig);//[5, 25, 35, 55]
System.out.println(treeSetOrig.pollFirst());//5
System.out.println(treeSetOrig);//[25, 35, 55]
//In above example element 5 is removed from the set and also removed from the
tree set.
```

pollLast returns the last element and removes it from the set.

```
System.out.println(treeSetOrig);//[25, 35, 55]
System.out.println(treeSetOrig.pollLast());//55
System.out.println(treeSetOrig);//[25, 35]
```

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.

HashMap Example

```
Map<String, Cricketer> hashmap = new HashMap<String, Cricketer>();
hashmap.put("sachin",
new Cricketer("Sachin", 14000));
hashmap.put("dravid",
new Cricketer("Dravid", 12000));
hashmap.put("ponting", new Cricketer("Ponting",
11500));
hashmap.put("bradman", new Cricketer("Bradman",
9996));
```

Hash Map Methods

get method gets the value of the matching key.

```
System.out.println(hashmap.get("ponting"));//Ponting 11500

//if key is not found, returns null.
System.out.println(hashmap.get("lara"));//null
```

If existing key is reused, it would replace existing value with the new value passed in.

```
//In the example below, an entry with key "ponting" is already present.
//Runs are updated to 11800.
hashmap.put("ponting", new Cricketer("Ponting",
11800));

//gets the recently updated value
System.out.println(hashmap.get("ponting"));//Ponting 11800
```

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.

```
Map<String, Cricketer> treemap = new TreeMap<String, Cricketer>();
treemap.put("sachin",
new Cricketer("Sachin", 14000));
System.out.println(treemap);
//{sachin=Sachin 14000}
```

We will now insert a Cricketer with key dravid. In sorted order,dravid comes before sachin. So, the value with key dravid is inserted at the start of the Map.

```
treemap.put("dravid",
new Cricketer("Dravid", 12000));
System.out.println(treemap);
//{dravid=Dravid 12000, sachin=Sachin 14000}
```

We will now insert a Cricketer with key ponting. In sorted order, ponting fits in between dravid and sachin.

```
treemap.put("ponting", new Cricketer("Ponting",
11500));
System.out.println(treemap);
//{dravid=Dravid 12000, ponting=Ponting 11500, sachin=Sachin 14000}

treemap.put("bradman", new Cricketer("Bradman",
9996));
System.out.println(treemap);
//{bradman=Bradman 9996, dravid=Dravid 12000, ponting=Ponting 11500,
sachin=Sachin 14000}
```

NavigableMap Interface Examples (TreeMap) Set I

Let's look at an example with TreeMap. Note that keys in TreeMap are sorted.

```
TreeMap<Integer, Cricketer> numbersTreeMap = new TreeMap<Integer, Cricketer>
();
numbersTreeMap.put(55, new Cricketer("Sachin",
14000));
numbersTreeMap.put(25, new Cricketer("Dravid",
12000));
numbersTreeMap.put(35, new Cricketer("Ponting",
12000));
numbersTreeMap.put(5,
new Cricketer("Bradman", 9996));
numbersTreeMap
.put(45, new Cricketer("Lara", 10000));
```

lowerKey method finds the highest key lower than specified key. floorKey method finds the highest key lower than or equal to specified key. Corresponding methods for finding lowest key higher than specified key are higher and ceiling. A few examples using the Map created earlier below.

```
//Find the highest key which is lower than 25
System.out.println(numbersTreeMap.lowerKey(25));//5

//Find the highest key which is lower than or equal to 25
System.out.println(numbersTreeMap.floorKey(25));//25

//Find the lowest key higher than 25
System.out.println(numbersTreeMap.higherKey(25));//35

//Find the lowest key higher than or equal to 25
System.out.println(numbersTreeMap.ceilingKey(25));//25
```

NavigableMap Interface Examples (TreeMap) Set II

Methods similar to subSet,headSet,tailSet (of TreeSet) are available in TreeMap as well. They are called subMap, headMap, tailMap.They have the similar signatures and results as the corresponding TreeSet Methods. They are inclusive with Lower Limit and NOT inclusive with higher limit - unless the (optional) inclusive flag is passed. The resultant sub map's are dynamic. If original map get modified, the sub map might be affected as well.

```
TreeMap<Integer, Cricketer> exampleTreeMap = new TreeMap<Integer, Cricketer>
();
exampleTreeMap.put(55, new Cricketer("Sachin",
14000));
exampleTreeMap.put(25, new Cricketer("Dravid",
12000));
exampleTreeMap.put(5,
new Cricketer("Bradman", 9996));
exampleTreeMap
.put(45, new Cricketer("Lara", 10000));

//Lower limit (5) inclusive, Uppper Limit(25) NOT inclusive
System.out.println(exampleTreeMap.subMap(5, 25));//{5=Bradman 9996}

System.out.println(exampleTreeMap.headMap(30));
//{5=Bradman 9996, 25=Dravid 12000}

System.out.println(exampleTreeMap.tailMap(25));
//{25=Dravid 12000, 45=Lara 10000, 55=Sachin 14000}
```

NavigableMap Interface Examples (TreeMap) Set III

Consider the next set of method examples below:

```
TreeMap<Integer, Cricketer> treeMapOrig = new TreeMap<Integer, Cricketer>();
treeMapOrig.put(55, new Cricketer("Sachin", 14000));
treeMapOrig.put(25, new Cricketer("Dravid", 12000));
treeMapOrig.put(5, new Cricketer("Bradman", 9996));
treeMapOrig.put(45, new Cricketer("Lara", 10000));

System.out.println(treeMapOrig);
//{5=Bradman 9996, 25=Dravid 12000, 45=Lara 10000, 55=Sachin 14000}
```

descendingMap method returns the tree set in reverse order.

```
NavigableMap<Integer, Cricketer> treeMapDesc = treeMapOrig
.descendingMap();
System.out.println(treeMapDesc);
//{55=Sachin 14000, 45=Lara 10000, 25=Dravid 12000, 5=Bradman 9996}
```

pollFirstEntry returns the first entry in the map and removes it from the map.

```
System.out.println(treeMapOrig);
//{5=Bradman 9996, 25=Dravid 12000, 45=Lara 10000, 55=Sachin 14000}
System.out.println(treeMapOrig.pollFirstEntry());//5=Bradman 9996
System.out.println(treeMapOrig);
//{25=Dravid 12000, 45=Lara 10000, 55=Sachin 14000}
//In above example element 5 is removed from the set and also removed from the
tree set.
```

pollLastEntry returns the last entry from the map and removes it from the map.

```
System.out.println(treeMapOrig);
//{25=Dravid 12000, 45=Lara 10000, 55=Sachin 14000}
System.out.println(treeMapOrig.pollLastEntry());//55=Sachin 14000
System.out.println(treeMapOrig);
//{25=Dravid 12000, 45=Lara 10000}
```

PriorityQueue

- PriorityQueue implements the Queue interface.

PriorityQueue Example

```
//Using default constructor - uses natural ordering of numbers
//Smaller numbers have higher priority
PriorityQueue<Integer> priorityQueue = new PriorityQueue<Integer>();
```

Adding an element into priority queue - offer method

```
priorityQueue.offer(24);
priorityQueue.offer(15);
priorityQueue.offer(9);
priorityQueue.offer(45);

System.out.println(priorityQueue);//[9, 24, 15, 45]
```

Peek method examples

```
//peek method get the element with highest priority.
System.out.println(priorityQueue.peek());//9
//peek method does not change the queue
System.out.println(priorityQueue);//[9, 24, 15, 45]

//poll method gets the element with highest priority.
System.out.println(priorityQueue.poll());//9
//peek method removes the highest priority element from the queue.
```

```
System.out.println(priorityQueue);//[24, 15, 45]

//This comparator gives high priority to the biggest number.
Comparator reverseComparator = new Comparator<Integer>() {
    public int compare(Integer paramT1,
    Integer paramT2) {
return paramT2 - paramT1;
    }

};
```

Priority Queue and Comparator

We can create priority queue using a comparator class i.e. custom defined priority.

```
PriorityQueue<Integer> priorityQueueDesc = new PriorityQueue<Integer>(
20, reverseComparator);

priorityQueueDesc.offer(24);
priorityQueueDesc.offer(15);
priorityQueueDesc.offer(9);
priorityQueueDesc.offer(45);

//45 is the largest element. Our custom comparator gives priority to highest
number.
System.out.println(priorityQueueDesc.peek());//45
```

Collections static methods

```
static int binarySearch(List, key) //Can be used only on sorted list
static int binarySearch(List, key, Comparator)
static void reverse(List)//Reverse the order of elements in a List.
static Comparator reverseOrder();
//Return a Comparator that sorts the reverse of the collection current sort
sequence.
static void sort(List)
static void sort(List, Comparator)
```

Generics

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

Need for Generics , Example

Consider the class below:

```
class MyList {
    private List<String> values;

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

    void remove(String value) {
values.remove(value);
    }
}
```

MyList can be used to store a list of Strings only.

```
MyList myList = new MyList();
myList.add("Value 1");
myList.add("Value 2");
```

To store integers, we need to create a new class. This is problem that Generics solve. Instead of hard-coding String class as the only type the class can work with, we make the class type a parameter to the class.

Generics Example

Let's replace String with T and create a new class.

```
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);
    }
}
```

Note the declaration of class:

```
class MyListGeneric<T>
```

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. For Example: See how T is used as a parameter and return type in the class MyListGeneric. Now the MyListGeneric class can be used to create a list of Integers or a list of Strings

```
MyListGeneric<String> myListString = new MyListGeneric<String>();
myListString.add("Value 1");
myListString.add("Value 2");

MyListGeneric<Integer> myListInteger = new MyListGeneric<Integer>();
myListInteger.add(1);
myListInteger.add(2);
```

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;

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

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

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

In declaration of the class, we specified a constraint "T extends Number". We can use the class MyListRestricted with any class extending Number - Float, Integer, Double etc.

```
MyListRestricted<Integer> restrictedListInteger = new
MyListRestricted<Integer>();
restrictedListInteger.add(1);
restrictedListInteger.add(2);
```

String not valid substitute for constraint "T extends Number".

```
//MyListRestricted<String> restrictedStringList =
//new MyListRestricted<String>();//COMPILER ERROR
```

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);
```

Generics and Collections Example 1

Consider the classes below:

```
class Animal {
}

class Dog extends Animal {
}
```

Let's create couple of Arrays and Lists as shown below:

```
Animal[] animalsArray = { new Animal(), new Dog() };
Dog[] dogsArray = { new Dog(), new Dog() };

List<Animal> animalsList = Arrays.asList(animalsArray);
List<Dog> dogsList = Arrays.asList(dogsArray);
```

Let's create a couple of static methods as shown below:

```
    static void doSomethingArray(Animal[] animals) {
//do Something with Animals
    }

    static void doSomethingList(List<Animal> animals) {
//do Something with Animals
    }
```

Array method can be called with Animal[] or Dog[]

```
doSomethingArray(animalsArray);
doSomethingArray(dogsArray);
```

List method works with List. Gives compilation error with List.

```
doSomethingList(animalsList);
//List<Dog> not compatible with List<Animal>
//doSomethingList(dogsList);//COMPILER ERROR
```

Summary : List not compatible with List even thought Dog extends Animal. However, Dog[] is compatible with Animal[].

Generics and Collections Example 2 - extends

Consider the methods below:

```
    static void doSomethingListModified(List<? extends Animal> animals) {
//Adding an element into  a list declared with ? is prohibited.

//animals.add(new Animal());//COMPILER ERROR!
//animals.add(new Dog());//COMPILER ERROR!
    }
```

Method declared with List<? extends Animal> compiles with both List and List

```
doSomethingListModified(animalsList);
doSomethingListModified(dogsList);
```

Generics and Collections Example 3 - Super

Method declared with List<? super Dog> compiles with both List and List.

```
    static void doSomethingListModifiedSuper(List<? super Dog> animals) {
//Adding an element into  a list declared with ? is prohibited.
//animals.add(new Animal());//COMPILER ERROR!
//animals.add(new Dog());//COMPILER ERROR!
    }
```

List of any super class of Dog is fine. List of any Subclass of Dog is not valid parameter.

```
doSomethingListModifiedSuper(animalsList);
doSomethingListModifiedSuper(dogsList);
```

Generics and Collections Example 4 , extends with interface

Below method can be called with a List declared with any type implementing the interface Serializable.

```
    static void doSomethingListInterface(List<? extends Serializable> animals)
{
//Adding an element into  a list declared with ? is prohibited.

//animals.add(new Animal());//COMPILER ERROR!
//animals.add(new Dog());//COMPILER ERROR!
    }
```

Generics and Collections , Few more Examples and Rules

A method declared with List can only be called with a List declared with type Object. None of the other classes are valid. A method declared with List<?> can be called with a List of any type. //A method declared with List<? extends Object> can be called with a List of any type - since all classes are sub classes of Object. ? can only be used in Declaring a type. Cannot be used as part of definition.

```
List<? extends Animal> listAnimals = new ArrayList<Dog>(); //COMPILES
//List<?> genericList = new ArrayList<? extends Animal>(); //COMPILER ERROR
```

Generics and Collection , Compatibility with old code

Consider the method below: It is declared to accept a Generic ArrayList. The method adds a string to the arraylist.

```
    static void addElement(ArrayList something){
something.add(new String("String"));
    }
```

Consider the code below:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(5);
numbers.add(6);
addElement(numbers);
```

We are passing a ArrayList to a method accepting ArrayList as parameter. We are trying to a add a string to it as well, inside the method. Compiling this class would give a warning: javac gives warning because multiplyNumbersBy2(ArrayList) is invoked with a Specific ArrayList and in the method addElement an element is added to ArrayList.

```
//com/rithus/generics/GenericsExamples.java uses unchecked or unsafe
operations.
//Recompile with -Xlint:unchecked for details.
```

To get more details run javac specifying the parameter Xlint:unchecked.

```
javac -Xlint:unchecked com/rithus/generics/GenericsExamples.java
//com/rithus/generics/GenericsExamples.java:21: warning: [unchecked]
//unchecked call to add(E) as a member of the raw type java.util.ArrayList
//something.add(new String("String"));
//     ^
```

Files

- Let us first look at the File class which helps us to create and delete files and directories. File class cannot be used to modify the content of a file.

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());
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");
```

File Class - Directory

A File class in Java represents a file and directory.

```
File directory = new File("src/com/rithus");
```

Print full directory path

```
System.out.println(directory.getAbsolutePath());
System.out.println(directory.isDirectory());//true
```

This does not create the actual file.

```
File fileInDir = new File(directory,"NewFileInDirectory.txt");
```

Actual file is created when we invoke createNewFile method.

```
System.out.println(fileInDir.createNewFile()); //true - First Time
```

Print the files and directories present in the folder.

```
System.out.println(Arrays.toString(directory.list()));
```

Creating a directory

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

Creating a file in a new directory

```
File notExistingDirectory = new File("notexisting");
File newFile = new File(notExistingDirectory,"newFile");

//Will throw Exception if uncommented: No such file or directory
//newFile.createNewFile();

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. Let's write an example to write and read from a file using FileReader and FileWriter.

FileWriter Class

We can write to a file using FileWriter class.

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

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.

```
FileWriter fileWriter2 = new FileWriter("FileName.txt");
fileWriter2.write("How are you doing Buddy?");
//Always flush before close. Writing to file uses Buffering.
fileWriter2.flush();
fileWriter2.close();
```

FileReader Class

File Reader can be used to read entire content from a file at one go.

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

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

```
FileReader fileReader2 = new FileReader("FileName.txt");
System.out.println(fileReader2.read(temp));//24
System.out.println(Arrays.toString(temp));//output below
```

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. Let's write an example to write and read from a file using BufferedReader and BufferedWriter.

BufferedWriter Class

BufferedWriter class helps writing to a class with better buffering than FileWriter.

BufferedWriter Constructors

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

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

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();
```

Reading the file created using BufferedReader

```
FileReader fileReader4 = new FileReader("PrintWriterFileName.txt");
BufferedReader bufferedReader2 = new BufferedReader(fileReader4);

String line2;
//readLine returns null when reading the file is completed.
while((line2=bufferedReader2.readLine()) != null){
    System.out.println(line2);
}
```

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.

```
class Rectangle implements Serializable {
    public Rectangle(int length, int breadth) {
this.length = length;
this.breadth = breadth;
area = length * breadth;
    }

    int length;
    int breadth;
    int area;
}
```

Serializing an object - Example

Below example shows how an instance of an object can be serialized. We are creating a new Rectangle object and serializing it to a file Rectangle.ser.

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

De-serializing an object , Example

Below example show how a object can be deserialized from a serialized file. A rectangle object is deserialized from the file Rectangle.ser

```
FileInputStream fileInputStream = new FileInputStream("Rectangle.ser");
ObjectInputStream objectInputStream = new ObjectInputStream(
fileInputStream);
Rectangle rectangle = (Rectangle) objectInputStream.readObject();
objectInputStream.close();
System.out.println(rectangle.length);// 5
System.out.println(rectangle.breadth);// 6
System.out.println(rectangle.area);// 30
```

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;)

```
//Modified Rectangle class

class Rectangle implements Serializable {
    public Rectangle(int length, int breadth) {
this.length = length;
this.breadth = breadth;
area = length * breadth;
    }

    int length;
    int breadth;
    transient int area;
}
```

If you run the program again, you would get following output

```
System.out.println(rectangle.length);// 5
System.out.println(rectangle.breadth);// 6
System.out.println(rectangle.area);// 0
```

Note that the value of rectangle.area is set to 0. Variable area is marked transient. So, it is not stored into the serialized file. And when de-serialization happens area value is set to default value i.e. 0.

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.

```
    private void readObject(ObjectInputStream is) throws IOException,
    ClassNotFoundException {
// Do whatever java does usually when de-serialization is called
is.defaultReadObject();
// In addition, calculate area also
area = this.length * this.breadth;
    }
```

When an object of Rectangle class is de-serialized, Java invokes the readObject method. The area is recalculated in this method. If we run the program again, we get the calculated area value back. Remember that area is not part of the serialized file. It is re-calculated in the readObject method.

```
System.out.println(rectangle.length);// 5
System.out.println(rectangle.breadth);// 6
System.out.println(rectangle.area);// 30
```

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.

```
    private void writeObject(ObjectOutputStream os) throws IOException {
//Do whatever java does usually when serialization is called
os.defaultWriteObject();
    }
```

If you run the above program again, you would get following output

```
System.out.println(rectangle.length);//5
System.out.println(rectangle.breadth);//6
System.out.println(rectangle.area);//30
```

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. Look at the class below. An object of class House contains an object of class Wall.

```
class House implements Serializable {
    public House(int number) {
super();
this.number = number;
```

```
    }


    Wall wall;
    int number;
}

class Wall{
    int length;
    int breadth;
    int color;
}
```

House implements Serializable where Wall doesn't.

Let's run this example program:

```
public class SerializationExamples2 {
    public static void main(String[] args)
    throws IOException {

FileOutputStream fileStream = new FileOutputStream(
"House.ser");
ObjectOutputStream objectStream = new ObjectOutputStream(
fileStream);
House house = new House(10);
house.wall = new Wall();
//Exception in thread "main" java.io.NotSerializableException:
//com.in28minutes.serialization.Wall
objectStream.writeObject(house);
objectStream.close();
    }
}

//Output:
//Exception in thread "main" java.io.NotSerializableException:
com.in28minutes.serialization.Wall
//    at java.io.ObjectOutputStream.writeObject0(Unknown Source)
//    at java.io.ObjectOutputStream.defaultWriteFields(Unknown Source)
```

This is because Wall is not serializable. Two solutions are possible. Make Wall transient => wall will not be serialized. This causes the wall object state to be lost. Make Wall implement Serializable => wall object will also be serialized and the state of wall object along with the house will be stored.

Serialization Program 1: Make Wall transient

```
class House implements Serializable {
    public House(int number) {
super();
this.number = number;
    }

    transient Wall wall;
    int number;
}
```

Serialization Program 2: Make Wall implement Serializable

```
class Wall implements Serializable {
    int length;
    int breadth;
    int color;
}
```

With both these programs, earlier main method would run without throwing an exception.

If you try de-serializing, In Example2, state of wall object is retained whereas in Example1, state of wall object is lost.

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. Let's consider the example code below:

Hero class extends Actor and Hero class implements Serializable. However, Actor class does not implement Serializable.

```
class Actor {
    String name;

    public Actor(String name) {
super();
this.name = name;
    }

    Actor() {
name = "Default";
    }
}
```

```
class Hero extends Actor implements Serializable {
    String danceType;

    public Hero(String name, String danceType) {
super(name);
this.danceType = danceType;
    }

    Hero() {
danceType = "Default";
    }
}
```

Let's run the code below:

```
FileOutputStream fileStream = new FileOutputStream("Hero.ser");
ObjectOutputStream objectStream = new ObjectOutputStream(fileStream);

Hero hero = new Hero("Hero1", "Ganganam");

// Before -> DanceType:Ganganam Name:Hero1
System.out.println("Before -> DanceType:" + hero.danceType + " Name:"
+ hero.name);

objectStream.writeObject(hero);
objectStream.close();

FileInputStream fileInputStream = new FileInputStream("Hero.ser");
ObjectInputStream objectInputStream = new ObjectInputStream(
fileInputStream);
hero = (Hero) objectInputStream.readObject();
objectInputStream.close();

// After -> DanceType:Ganganam Name:Default
System.out.println("After -> DanceType:" + hero.danceType + " Name:"
+ hero.name);
```

Code executes successfully but after de-serialization, the values of super class instance variables are not retained. They are set to their initial values. 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. Let's first understand the need for threading and

then look into how to create a thread and what is synchronization?

Need for Threads

We are creating a Cricket Statistics Application. Let's say the steps involved in the application are STEP I: Download and Store Bowling Statistics => 60 Minutes STEP II: Download and Store Batting Statistics => 60 Minutes STEP III: Download and Store Fielding Statistics => 15 Minutes STEP IV: Merge and Analyze => 25 Minutes Steps I, II and III are independent and can be run in parallel to each other. Run individually this program takes 160 minutes. We would want to run this program in lesser time. Threads can be a solution to this problem. Threads allow us to run STEP I, II and III in parallel and run Step IV when all Steps I, II and III are completed.

Need for Threads Example

Below example shows the way we would write code usually , without using Threads.

```
ThreadExamples example = new ThreadExamples();
example.downloadAndStoreBattingStatistics();
example.downloadAndStoreBowlingStatistics();
example.downloadAndStoreFieldingStatistics();

example.mergeAndAnalyze();
```

downloadAndStoreBowlingStatistics starts only after downloadAndStoreBattingStatistics completes execution. downloadAndStoreFieldingStatistics starts only after downloadAndStoreBowlingStatistics completes execution. What if I want to run them in parallel without waiting for the others to complete? This is where Threads come into picture.

Creating a Thread Class

Creating a Thread class in Java can be done in two ways. Extending Thread class and implementing Runnable interface. Let's create the BattingStatisticsThread extending Thread class and BowlingStatisticsThread implementing Runnable interface.

Creating a Thread By Extending Thread class

Thread class can 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();
```

Thread created implementing RunnableInterface.

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. Let's first look at an example where this problem can occur.

Example Program:

Consider the SpreadSheet class below. It consists of three cells and also a method setAndGetSum which sets the values into the cell and sums them up.

```
class SpreadSheet {
    int cell1, cell2, cell3;

    int setandGetSum(int a1, int a2, int a3) {
cell1 = a1;
sleepForSomeTime();
cell2 = a2;
sleepForSomeTime();
cell3 = a3;
sleepForSomeTime();
return cell1 + cell2 + cell3;
    }

    void sleepForSomeTime() {
try {
    Thread.sleep(10 * (int) (Math.random() * 100));
} catch (InterruptedException e) {
    e.printStackTrace();
}
    }
}
```

Serial Run

Let's first run the above example in a serial way and see what the output would be.

```
public static void main(String[] args) {
    SpreadSheet spreadSheet = new SpreadSheet();
    for (int i = 0; i < 4; i++) {
System.out.print(spreadSheet.setandGetSum(i, i * 2, i * 3) + " ");
    }
}
//Output - 0 6 12 18
```

Output would contain a multiple of 6 always because we are calling with i, i*2 and i*3 and summing up. So, the result should generally be i*6 with i running from 0 to 3.

Example Program using Threads

Let's now run the SpreadSheet class in a Thread. Example Code below:

```
public class ThreadExampleSynchronized implements Runnable {

    SpreadSheet spreadSheet = new SpreadSheet();

    public void run() {
for (int i = 0; i < 4; i++) {
    System.out.print(
    spreadSheet.setandGetSum(i,i * 2, i * 3)
    + " ");
}
    }

    public static void main(String[] args) {
ThreadExampleSynchronized r = new ThreadExampleSynchronized();
Thread one = new Thread(r);
Thread two = new Thread(r);
one.start();
two.start();
    }
}
```

We are creating 2 instances of the Thread using the interface , one and two. And start method is invoked to run the thread. Both threads share the instance of SpreadSheet class , spreadsheet. Output

```
FIRST RUN    : 0 1 6 9 12 15 18 18
SECOND RUN    : 0 3 6 6 12 15 18 18
THIRD RUN    : 0 3 6 9 12 15 18 18
```

Output Discussion What we see is that different runs have different results. That's expected with threads. What is not expected is to see numbers like 1, 9, 15 and 3 in the output. We are expecting to see multiples of 6 in the output(as in the earlier serial run) but we see numbers which are not multiples of 6. Why is this happening? This is a result of having two threads run in parallel without synchronization. Consider the code in the setAndGetSum method.

```
int setandGetSum(int a1, int a2, int a3) {
    cell1 = a1;
    sleepForSomeTime();
    cell2 = a2;
    sleepForSomeTime();
    cell3 = a3;
    sleepForSomeTime();
    return cell1 + cell2 + cell3;
}
```

After setting the value to each cell, there is a call for the Thread to sleep for some time. After Thread 1 sets the value of cell1, it goes to Sleep. So, Thread2 starts executing. If Thread 2 is executing Òreturn cell1 + cell2 + cell3;Ó, it uses cell1 value set by Thread 1 and cell2 and cell3 values set by Thread 2. This results in the unexpected results that we see when the method is run in parallel. What is explained is one possible scenario. There are several such scenarios possible. 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 int setandGetSum(int a1, int a2, int a3) {
    cell1 = a1;
    sleepForSomeTime();
    cell2 = a2;
    sleepForSomeTime();
    cell3 = a3;
    sleepForSomeTime();
    return cell1 + cell2 + cell3;
}
```

Output of the program now is Ò0 0 6 6 12 12 18 18Ó. This is expected output , all numbers are multiples of 6.

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 method Example

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

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;
    }
```

Static synchronized block Example

Static blocks are synchronized on the class.

```
    static int getCount2(){
synchronized (SynchronizedSyntaxExample.class) {
    return count;
}
    }
```

Static and non static synchronized methods and blocks

Static methods and block are synchronized on the class. Instance methods and blocks are synchronized on the instance of the class i.e. an object of the class. Static synchronized methods and instance synchronized methods don't affect each other. This is because they are synchronized on two different things.

States of a Thread

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

- NEW;
- RUNNABLE;
- RUNNING;
- BLOCKED/WAITING;
- TERMINATED/DEAD; Let's consider the example that we discussed earlier.

Example Program

```
LINE 1: BattingStatisticsThread battingThread1 = new
BattingStatisticsThread();
LINE 2: battingThread1.start();

LINE 3: BowlingStatisticsThread battingInterfaceImpl = new
BowlingStatisticsThread();
LINE 4: Thread battingThread2 = new Thread(battingInterfaceImpl);
LINE 5: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
```

States of a Thread - Examples

A thread is in NEW state when an object of the thread is created but the start method is not yet called. At the end of line 1, battingThread1 is in NEW state. A thread is in RUNNABLE state when it is eligible to run, but not running yet. (A number of Threads can be in RUNNABLE state. Scheduler selects which Thread to move to RUNNING state). In the above example, sometimes the Batting Statistics thread is running and at other time, the Bowling Statistics Thread is running. When Batting Statistics thread is Running, the Bowling Statistics thread is ready to run. It's just that the scheduler picked Batting Statistics thread to run at that instance and vice-versa. When Batting Statistics thread is Running, the Bowling Statistics Thread is in Runnable state (Note that the Bowling Statistics Thread is not waiting for anything except for the Scheduler to pick it up and run it). A thread is RUNNING state when it's the one that is currently , what else to say, Running. 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.

Thread Priority Example

Consider the thread example declared below:

```
class ThreadExample extends Thread {
    public void run() {
for (int i = 0; i < 1000; i++)
    System.out
    .println( this.getName() + " Running "
    + i);
    }
}
```

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

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

Java also provides predefined constants Thread.MAX_PRIORITY(10), Thread.MIN_PRIORITY(1), Thread.NORM_PRIORITY(5) which can be used to assign priority to a thread.

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

Look at the example code below:

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

thread3.join() method call force the execution of main method to stop until thread3 completes execution. After that, thread4.start() method is invoked, putting thread4 into a Runnable State.

Overloaded Join method

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

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

In above example, main method thread would wait for 2000 ms or the end of execution of thread4, whichever is minimum.

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.

Deadlock Example

Consider the example classes below: Resource represents any resource that you need access to. A network connection, database connection etc. Operation represents an operation that can be done on these resources. Let's say that Operation need two resources, resource1 and resource2 and offer two operations method1 and method2. Look at the program below:

```
class Resource {
```

```
}

class SomeOperation {
    Resource resource1 = new Resource();
    Resource resource2 = new Resource();

    void method1() throws InterruptedException {
synchronized (resource1) {
    Thread.sleep(1000);
    //code using resource1
    synchronized (resource2) {
//code using resource2
    }
}
    }

    void method2() throws InterruptedException {
System.out.println(Thread.currentThread().getName()
+ "is in method2");
synchronized (resource2) {
    //code using resource2
    Thread.sleep(1000);
    synchronized (resource1) {
//code using resource1
    }
}
    }

}
```

Method1 executes some code on resource1 first and then executes some code on resource2. Method2 does the reverse. We use the sleep method call to simulate the fact that these operation could take some time to complete. Let's create two threads sharing the above operation using the code below: Threads one and two now share object operation. The thread code runs both operations method1 and method2.

```
public class ThreadDeadlock implements Runnable {

    SomeOperation operation = new SomeOperation();

    @Override
    public void run() {
try {
    operation.method1();
    operation.method2();
} catch (InterruptedException e) {
    e.printStackTrace();
}
    }
```

```
    public static void main(String[] args) {
ThreadDeadlock r = new ThreadDeadlock();
Thread one = new Thread(r);
Thread two = new Thread(r);
one.start();
two.start();
    }
}
```

When executed this program just hangs, because of a deadlock. To make what is happening behind the screens more clear, Let's add in a few sysout's in the Operation class.

```
class SomeOperation {
    Resource resource1 = new Resource();
    Resource resource2 = new Resource();

    void method1() throws InterruptedException {
synchronized (resource1) {
    System.out.println("Method1 - got resource1");
    Thread.sleep(1000);
    //code using resource1
    System.out.println("Method1 - waiting for resource2");
    synchronized (resource2) {
System.out.println("Method1 - got resource2");
//code using resource2
    }
}
    }

    void method2() throws InterruptedException {
synchronized (resource2) {
    System.out.println("Method2 - got resource2");
    //code using resource2
    Thread.sleep(1000);
    System.out.println("Method2 - waiting for resource1");
    synchronized (resource1) {
System.out.println("Method2 - got resource1");
//code using resource1
    }
}
    }

}
```

Output:

```
Method1 - got resource1
Method1 - waiting for resource2
Method1 - got resource2
Method1 - got resource1
Method2 - got resource2
Method1 - waiting for resource2
Method2 - waiting for resource1
HANGSÉÉÉÉÉÉÉÉ
```

Now we have two threads waiting for resources held by one another. This results in a deadlock.

Thread - wait, notify and notifyAll methods

Consider the example below: Calculator thread calculates two values: Sum upto Million and Sum upto 10 Million. Main program uses the output of sum upto million.

Example 1

```
class Calculator extends Thread {
    long sumUptoMillion;
    long sumUptoTenMillion;
    public void run() {
calculateSumUptoMillion();
calculateSumUptoTenMillion();
    }

    private void calculateSumUptoMillion() {
for (int i = 0; i < 1000000; i++) {
    sumUptoMillion += i;
}
    }
    private void calculateSumUptoTenMillion() {
for (int i = 0; i < 10000000; i++) {
    sumUptoTenMillion += i;
}
    }
}

public class ThreadWaitAndNotify {
    public static void main(String[] args) {
Calculator thread = new Calculator();
thread.start();
System.out.println(thread.sumUptoMillion);
    }
}
```

Output

```
0
```

Output printed is 0. This is because the thread has not finished calculating the value of sumUptoMillion when the main method prints the value to the output. We have to find a way to stop the main method from running until sumUptoMillion is calculated. One option is to use the join method. But, join method would make the main method wait until both the operations (sumUptoMillion and sumUptoTenMillion) are completed. But, we want main method to wait only for sumUptoMillion. We can achieve this using wait and notify methods. wait and notify methods can only be used in a synchronized context.

Example with wait and notify methods

```
package com.in28minutes.threads;

class Calculator extends Thread {
    long sumUptoMillion;
    long sumUptoTenMillion;

    public void run() {
synchronized (this) {
    calculateSumUptoMillion();
    notify();
}
calculateSumUptoTenMillion();
    }

    private void calculateSumUptoMillion() {
for (int i = 0; i < 1000000; i++) {
    sumUptoMillion += i;
}
System.out.println("Million done");
    }

    private void calculateSumUptoTenMillion() {
for (int i = 0; i < 10000000; i++) {
    sumUptoTenMillion += i;
}
System.out.println("Ten Million done");
    }
}

public class ThreadWaitAndNotify {
    public static void main(String[] args) throws InterruptedException {
Calculator thread = new Calculator();
synchronized(thread){
    thread.start();
    thread.wait();
}
```

```
System.out.println(thread.sumUptoMillion);
    }
}
```

Output

```
Million done
499999500000
Ten Million done
```

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){
    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) {
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.

Assert Details

To compile code using 1.3 you can use the command below javac -source 1.3 OldCode.java => assert can be used as identifier with -source 1.4,1.5,5,1.6,6 => assert cannot be used as identifier

Assertions can easily be enabled and disabled. Assertions are disabled by default.

Enabling Assertions

Enable assertions: java -ea com.in28minutes.AssertExamples (OR) java -enableassertions com.in28minutes.AssertExamples

Disable Assertions

Disable assertions: java -da com.in28minutes.AssertExamples (OR) java -disableassertions com.in28minutes.AssertExamples

Enable Assertions in specific packages

Selectively enable assertions in a package only java -ea:com.rithus

Selectively enable assertions in a package and its subpackages only java -ea:com.in28minutes...

Enable assertions including system classes

java -ea -esa

Basic assert condition example

Basic assert is shown in the example below

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

Assert with debugging information: Example

If needed - debugging information can be added to an assert. Look at the example below.

```
private int computeCompoundInterest(int principal,float interest,int years){
    //condition is always boolean
    //second parameter can be anything that converts to a String.
    assert(principal>0): "principal is " + principal;
    return 100;
}

public static void main(String[] args) {
    AssertExamples examples = new AssertExamples();
    System.out.println(examples.computerSimpleInterest(-1000,1.0f,5));
}
```

Asserts - Not for validation

Assertions should not be used to validate input data to a public method or command line argument. IllegalArgumentException would be a better option.

In public method, only use assertions to check for cases which are never supposed to happen.

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.

Garbage Collection Example

Let's say the below method is called from a function.

```
void method(){
    Calendar calendar = new GregorianCalendar(2000,10,30);
    System.out.println(calendar);
}
```

An object of the class GregorianCalendar is created on the heap by the first line of the function with one reference variable calendar.

After the function ends execution, the reference variable calendar is no longer valid. Hence, there are no references to the object created in the method.

JVM recognizes this and removes the object from the heap. This is called Garbage Collection.

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();

Initialization Blocks

- Initialization Blocks - Code which runs when an object is created or a class is loaded

Types of Initialization Blocks

There are two types of Initialization Blocks Static Initializer: Code that runs when a class is loaded. Instance Initializer: Code that runs when a new object is created.

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.

Example Output

```
Static Initializer
Count when Static Initializer is run is 0
```

Even though three instances are created static initializer is run only once.

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();
    }

}
```

Code within instance initializer is run every time an instance of the class is created.

Example Output

```
Instance Initializer
Count when Instance Initializer is run is 1
Instance Initializer
Count when Instance Initializer is run is 2
Instance Initializer
Count when Instance Initializer is run is 3
```

Java Bean Conventions

- When is a java class called a bean? Let's find an answer to this question.

Java Bean Example

Consider the code example below:

```
public class JavaBeansStandards {

private String name;
private boolean good;

public String getName() {
    return name;
}

public boolean isGood() {
    return good;
}

public void setName(String name) {
    this.name = name;
}

public void setGood(boolean isGood) {
    this.good = isGood;
}
```

```
public void addSomeListener(MyListener listener){
}

public void removeSomeListener(MyListener listener){
}

}
```

Private Member Variables

Good practice is to have all member variables in a class declared as private.

```
private String name;
private boolean good;
```

Naming setter and getter methods

- To modify and access values of properties we use setter and getter methods. Getters and setters should be public.
- Getters should not have any arguments passed.
- Setters should take one argument (the property value) with same type as the return value of getter.
- Non boolean getter name should be (get + PropertyName)
- boolean getter name can be (get + PropertyName) or (is + PropertyName)
- All setters should be named (set + PropertyName)

Examples

```
public String getName() {
    return name;
}

public boolean isGood() {
    return good;
}

public void setName(String name) {
    this.name = name;
}
public void setGood(boolean isGood) {
    this.good = isGood;
}
```

Listener naming conventions

Methods to register/add a listener should use prefix "add" and suffix "Listener". They should accept 1 parameter - the listener object to be added.

Example

```
public void addSomeListener(MyListener listener){
}
```

Methods to de-register/remove a listener should use prefix "remove" and suffix "Listener". They should accept 1 parameter - the listener object to be removed

Example

```
public void removeSomeListener(MyListener listener){
}
```

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.

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>]
```

Output shows the matching strings 12, 12. Also, shown in the output is the starting index of each match. First 12 is present starting at index 0. The next 12 in the string starts at index 7.

Creating Regular Expressions for Java

Let's test regular expressions by using the method we created earlier: regex().

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< >]
```

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>]
```

Look for one or more characters from a to z (only small case).

```
regex("[a-z]+", "abcZ2xyzN1yza");//[0<abc>, 5<xyz>, 10<yza>]
//0123456789012
```

Regular Expressions , Look for Repetitions

Regular expressions can be joined together to look for a combination. a+b+ looks 1 or more a's and 1 or more b's next to each other. Notice that only a's or only b's do not match.

```
regex("a+b+", "aabcacaabbbcbb");//[0<aab>, 6<aabbb>]
```

* - 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+b*c? looks for 1 or more a's followed by 0 or more b's followed by 0 or 1 c's. a => matches. ab => matches. abc=>matches. abcc => does not match (only 0 or 1 c's)

```
regex("a+b*c?", "adabdabcdabccd");//[0<a>, 2<ab>, 5<abc>, 9<abc>]
```

^a looks for anything other than a

```
regex("[^a]+", "bcadefazyx");//[0<bc>, 3<def>, 7<zyx>]
```

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

```
regex("[^abcd]+a", "efgazyazyzb");//[0<efga>, 4<zya>]
```

. matches any character

a.c looks for Ôa' followed by any character followed by Ôc'. abc => match abbc => no match (. matches 1 character only)

```
regex("a.c", "abca ca!cabbc");//[0<abc>, 3<a c>, 6<a!c>]
```

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);
}
```

Example

```
regexUsingScanner("a+?", "aaaaab");//[a, a, a, a, a]
```

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.

Example method for Tokenizing

```
private static void tokenize(String string,String regex) {
    String[] tokens = string.split(regex);
    System.out.println(Arrays.toString(tokens));
}
```

Example:

```
tokenize("ac;bd;def;e",";");//[ac, bd, def, e]
```

Tokenizing using Scanner Class

```
private static void tokenizeUsingScanner(String string,String regex) {
    Scanner scanner = new Scanner(string);
    scanner.useDelimiter(regex);
    List<String> matches = new ArrayList<String>();
    while(scanner.hasNext()){
matches.add(scanner.next());
    }
    System.out.println(matches);
}
```

Example:

```
tokenizeUsingScanner("ac;bd;def;e",";");//[ac, bd, def, e]
```

Scanner Class: Other Functions

```
private static void lookForDifferentThingsUsingScanner(
String string) {
    Scanner scanner = new Scanner(string);
    while(scanner.hasNext()){
if(scanner.hasNextBoolean()){
    System.out.println("Found Boolean:" + scanner.nextBoolean());
} else if(scanner.hasNextInt()){
    System.out.println("Found Integer:" + scanner.nextInt());
} else {
    System.out.println("Different thing:" + scanner.next());
}
    }
}
```

Scanner has more useful functions other than just looking for a delimiter

Example:

```
lookForDifferentThingsUsingScanner("true false 12 3 abc true 154");
```

Output

```
//Found Boolean:true
//Found Boolean:false
//Found Integer:12
//Found Integer:3
//Different thing:abc
//Found Boolean:true
//Found Integer:154
```

Expressions

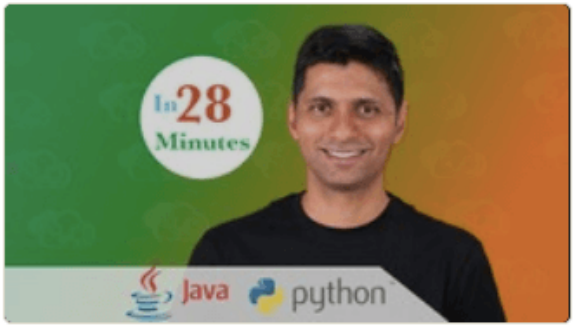
Where are objects created? Where are strings created?

TODO

- replace `with` java at start of code
- //Getters and Setters are eliminated to keep the example short
- Java SE vs ME vs EE
- Check for long lines which are cut off in pdf
- Notes from Venkat's Talk
    - Designed for a different world - PEnguins
    - OOPS - Terrible - 1970s
    - FP - Lambda calculus-1929
    - Complexity from problems vs Complexity from solutions
    - Structured Programming - One Start One Exit????
    - Functional Programming
        - Assignment LEss
        - Statements vs Expressions
        - Pure Function - No side effects and Zero dependencies that change
        - Referential Transparency
        - Pure Functions
            - idempotent
            - referenctial transparency
            - memorizable
            - easier to testt
            - may be lazily evaluated
        - Higher order function
            - take/create/return object vs take/create/return function
        - Java Future - imperative + oops -> functional + oops

        - Stream - Does not evaluate the function on all the data. It takes a collection of functions (fusing - intermediate operations combined) and executes them on each piece of data..
            - Functional Pipeline, Function
            - "Coolection pipeline pattern" -> Martin Fowler
        - Lambda - Stateless
        - Closure has State
    - Code is Liability not an asset!
    - Exception Handling - Promise
    - Reactive Programming
        - Error, Data and End channels
        - Error is a first class citizen
        - Handle errors down stream

Complete Java Course



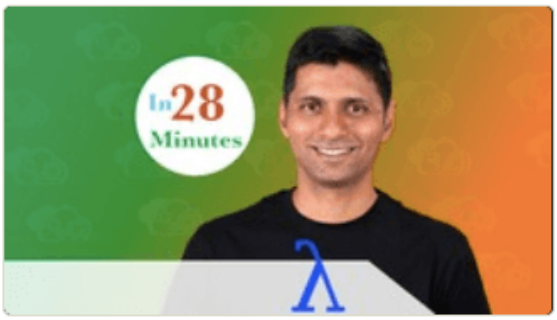
Python for Beginners - Go from Java to Python in 100 Steps

Learn Python Programming using Your Java Skills. For Beginner Python Programmers.

in28Minutes Official

4.5 ★★★★½ (531)

8 total hours • 103 lectures • All Levels



Learn Java Functional Programming with Lambdas & Streams

Learn Java Functional Programming with Lambdas & Streams. Solve Java Functional Programming Puzzles & Exercises.

in28Minutes Official

4.5 ★★★★½ (499)

4.5 total hours • 44 lectures • All Levels

Bestseller

Troubleshooting

- Refer our TroubleShooting Guide - https://github.com/in28minutes/in28minutes-initiatives/tree/master/The-in28Minutes-TroubleshootingGuide-And-FAQ

Youtube Playlists - 500+ Videos

Click here - 30+ Playlists with 500+ Videos on Spring, Spring Boot, REST, Microservices and the Cloud

Keep Learning in28Minutes

in28Minutes is creating amazing solutions for you to learn Spring Boot, Full Stack and the Cloud - Docker, Kubernetes, AWS, React, Angular etc. - Check out all our courses here