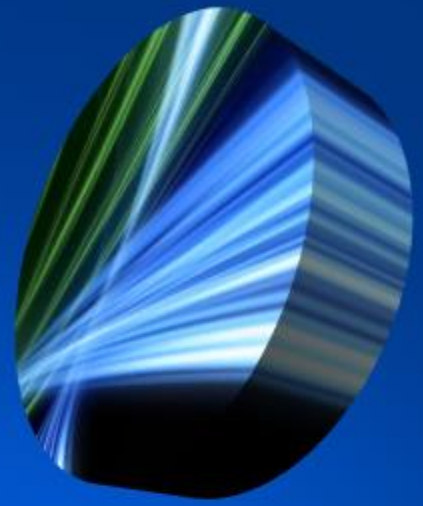
Java fundamentals to advance



By Krishantha Dinesh
(HDIT, PGDIT, MscIT, MBCS, MIEEE)

Prerequisites

- Basic java understanding and programming experionse
- Mobilephone.soundmode=soundmode.completesilent && soundmode.completesilent != soundmode.vibrate

What is java

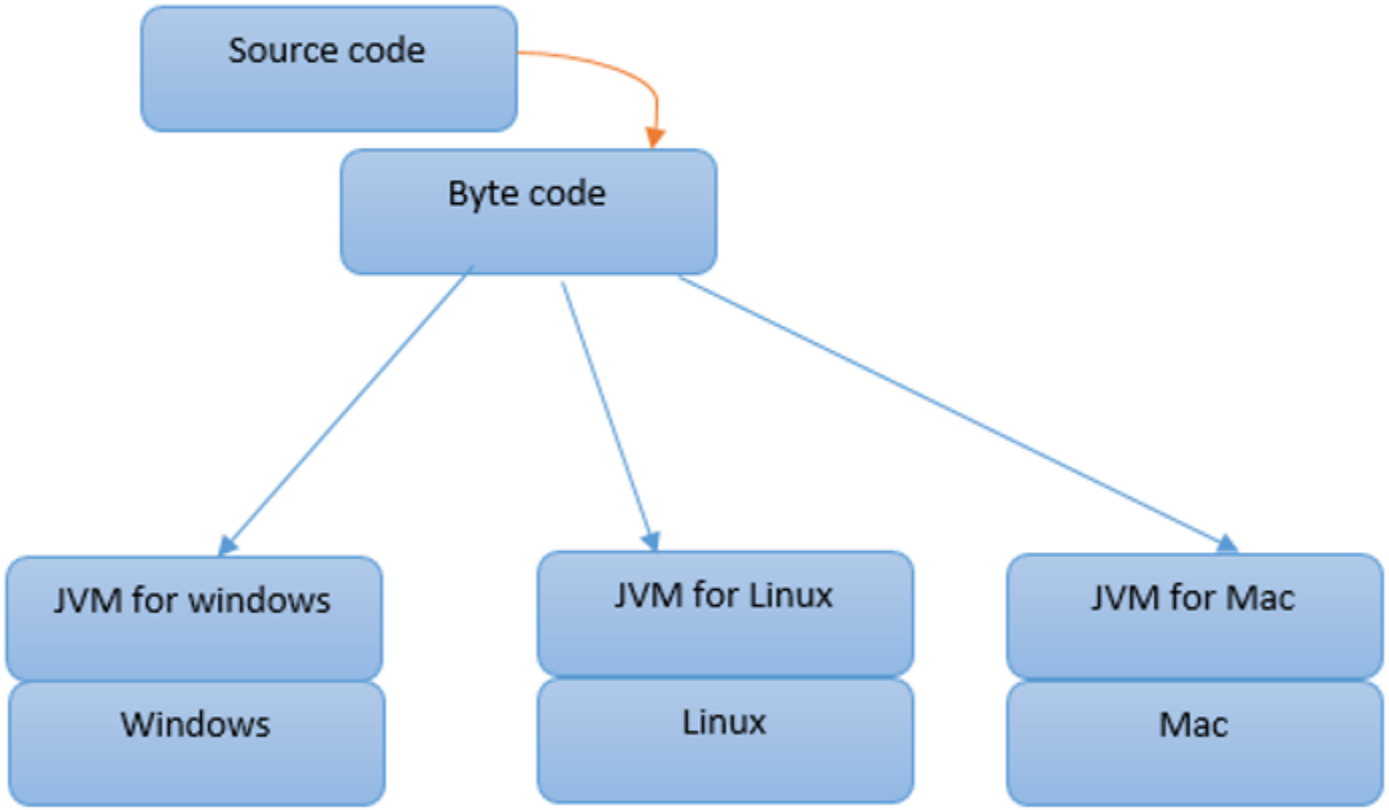
- it is simple question but most of peoples are struggled to answer.
- simple answer (which is not complete answer) is java is a programming language based on C and C++.
- java is designed to run on virtual machine based with the concept call “write once run anywhere”.
- if you are coming from C# background you will find this is very much similar.
- obviously java is object oriented programming language also NOT platform dependent.

History

- father of java is James Gosling and first release made in 1995.
- in 2004 java 1.5 has been released as java 5 with significant changes
- current version when this slides are produced is java 8

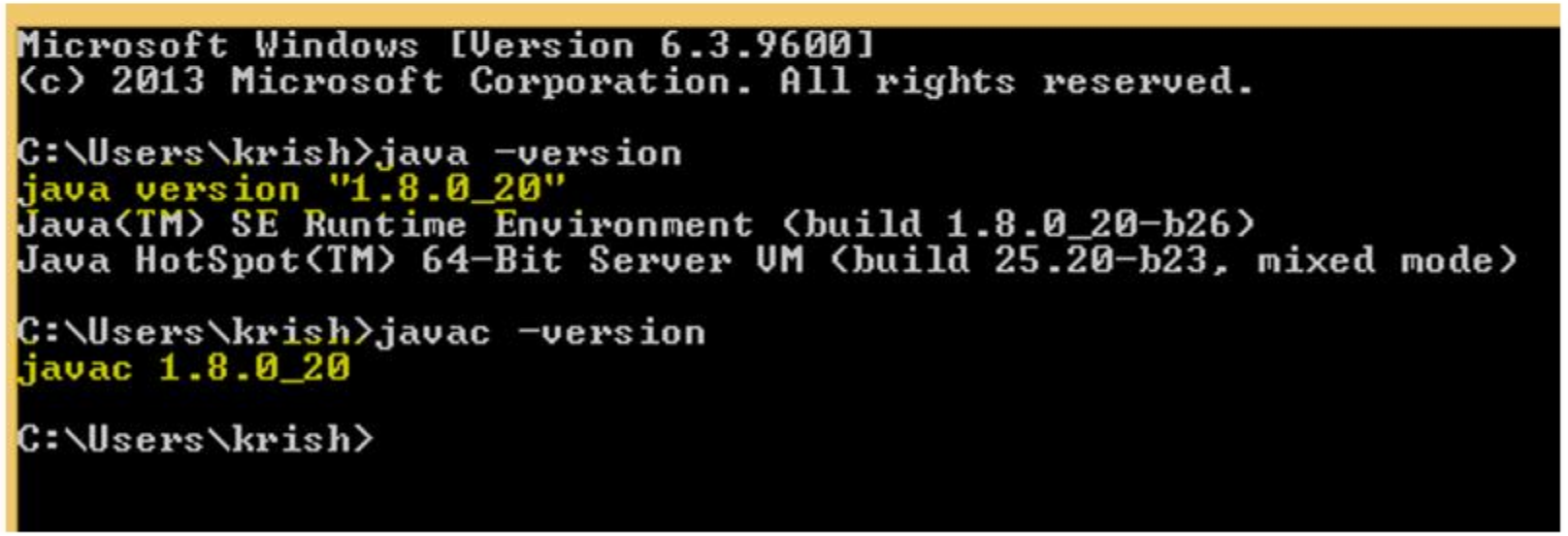
JVM

- even though java considered as platform **independent** JVM is highly platform **dependent**. there are separate JVM for each operation systems.
- first java compile source code to byte code which can read by any JVM. now JVM convert that byte code the machine code according to each operations systems handle things like networking, storage.
- biggest advantage here is you are writing code to java specification and no need to worry about how operation system deal with it.


Source code
Byte code
JVM for windows
Windows
JVM for Linux
Linux
JVM for Mac
Mac

Write first java program

- now you are ready to write first java program. this slide will not describe how to install java. once you installed
  - java -version
  - javac -version
- output should be equal as following image shows. but these number can be different according to your installed version.



- Go to command prompt and move to location you want to store your program then type
  - notepad HelloWorld.java
- write following code in notepad and save it.

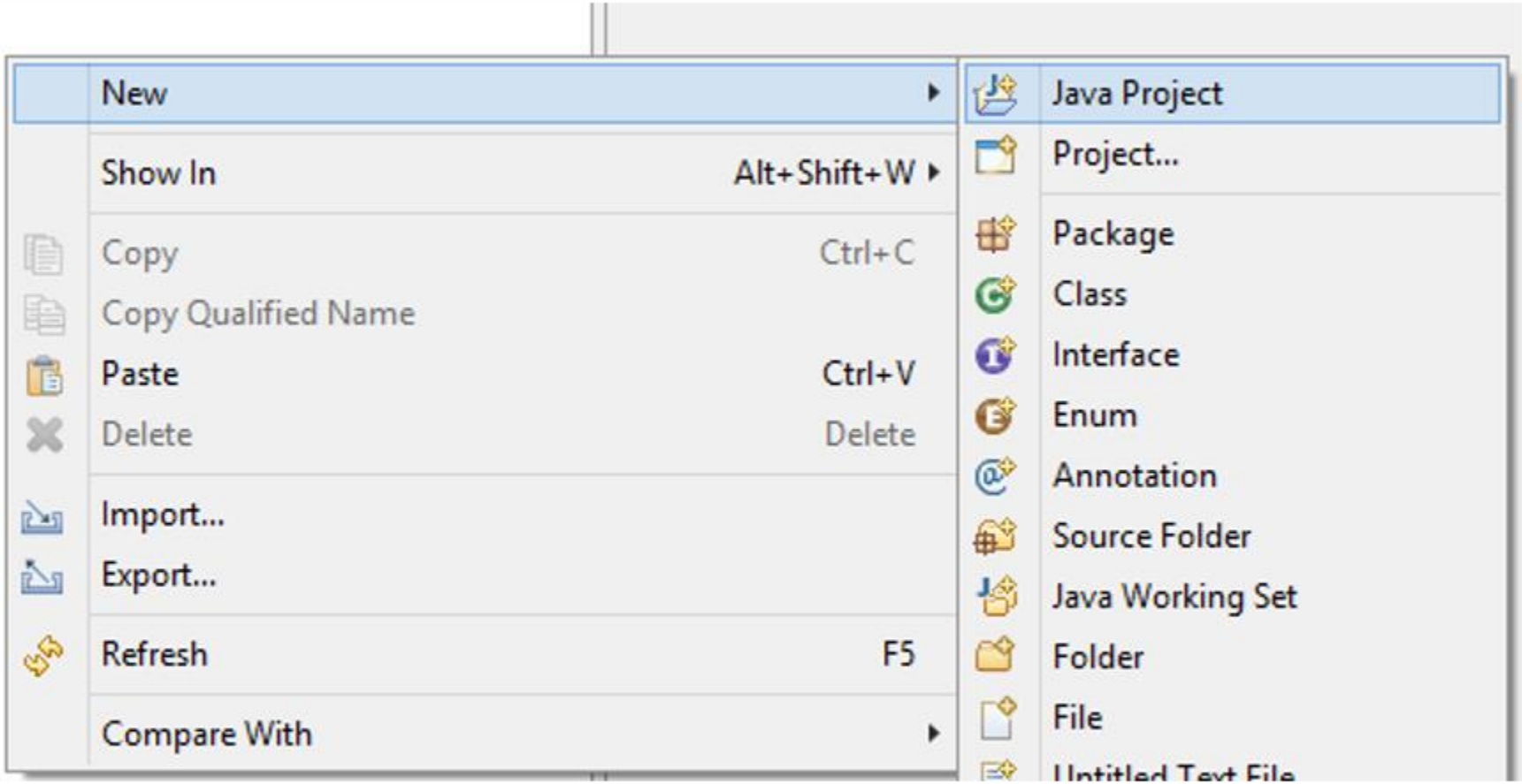
```
public class HelloWorld{
        public static void main (String[] args){
                System.out.println("Hello World 2");
        }
}
```

- now from command prompt type
  - javac HelloWorld.java
- it will create new file call HelloWorld.class now you can run your program as
  - java HelloWorld

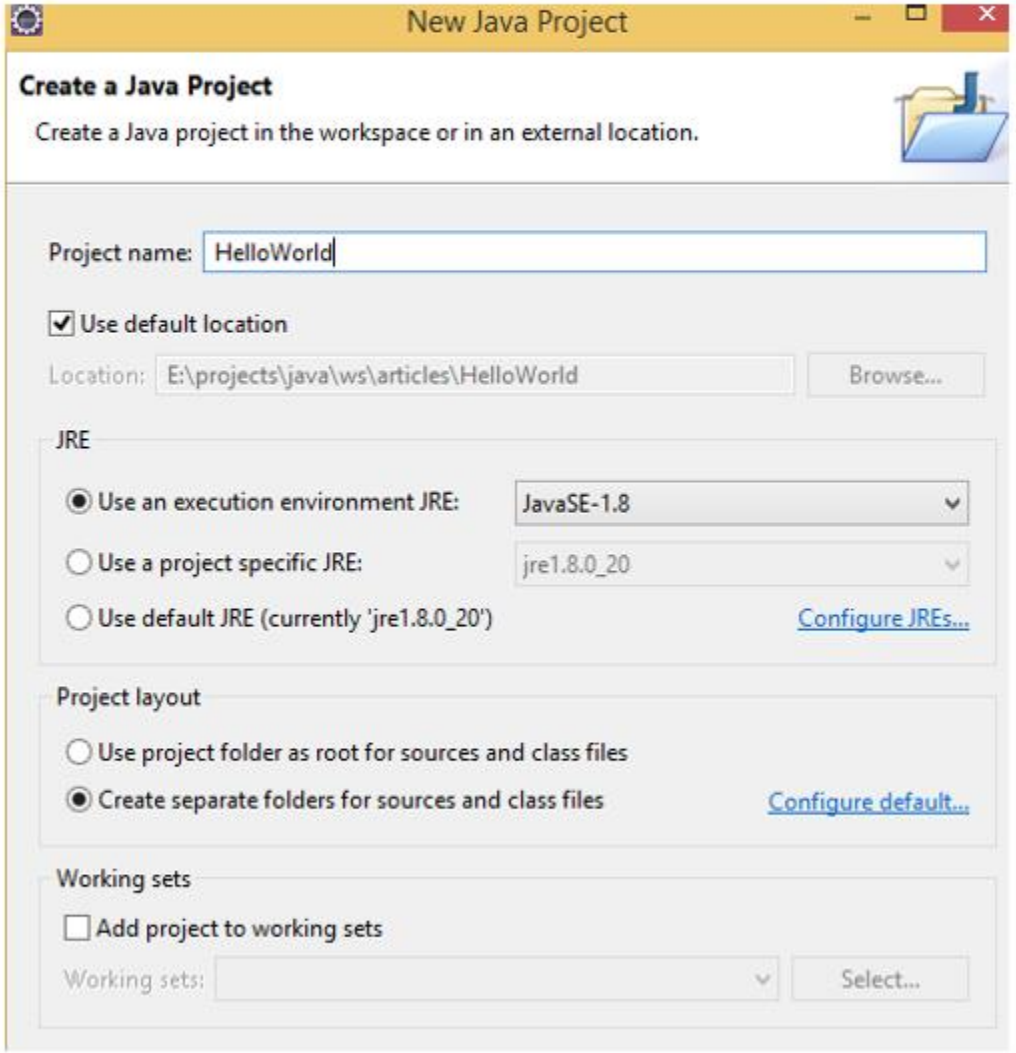
IDE

- there are many IDE you can use to develop java program.
- Here we consider eclipse which is more famous IDE among java developers.
- you can download eclipse from its official web site but you need to make sure you are downloading correct version for you jdk. for example if you are using 32 bit jdk then you need to download 32 bit eclipse version
- eclipse is workspace based IDE and it will ask to create workspace in first time execute. you can use default one or your customized one. eclipse will create all projects inside that workspace. also you can use multiple workspace based on your requirement.

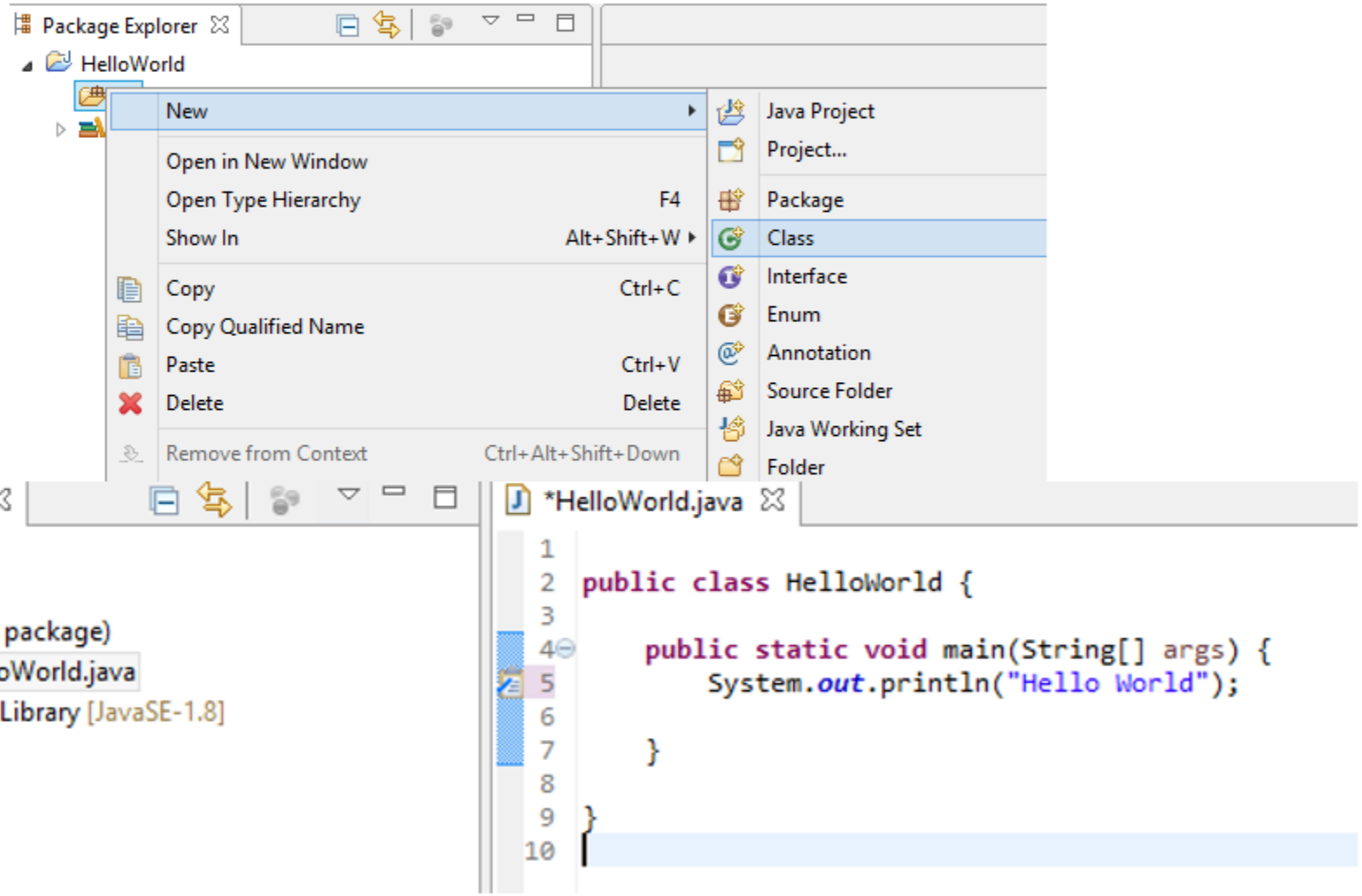
create new java project



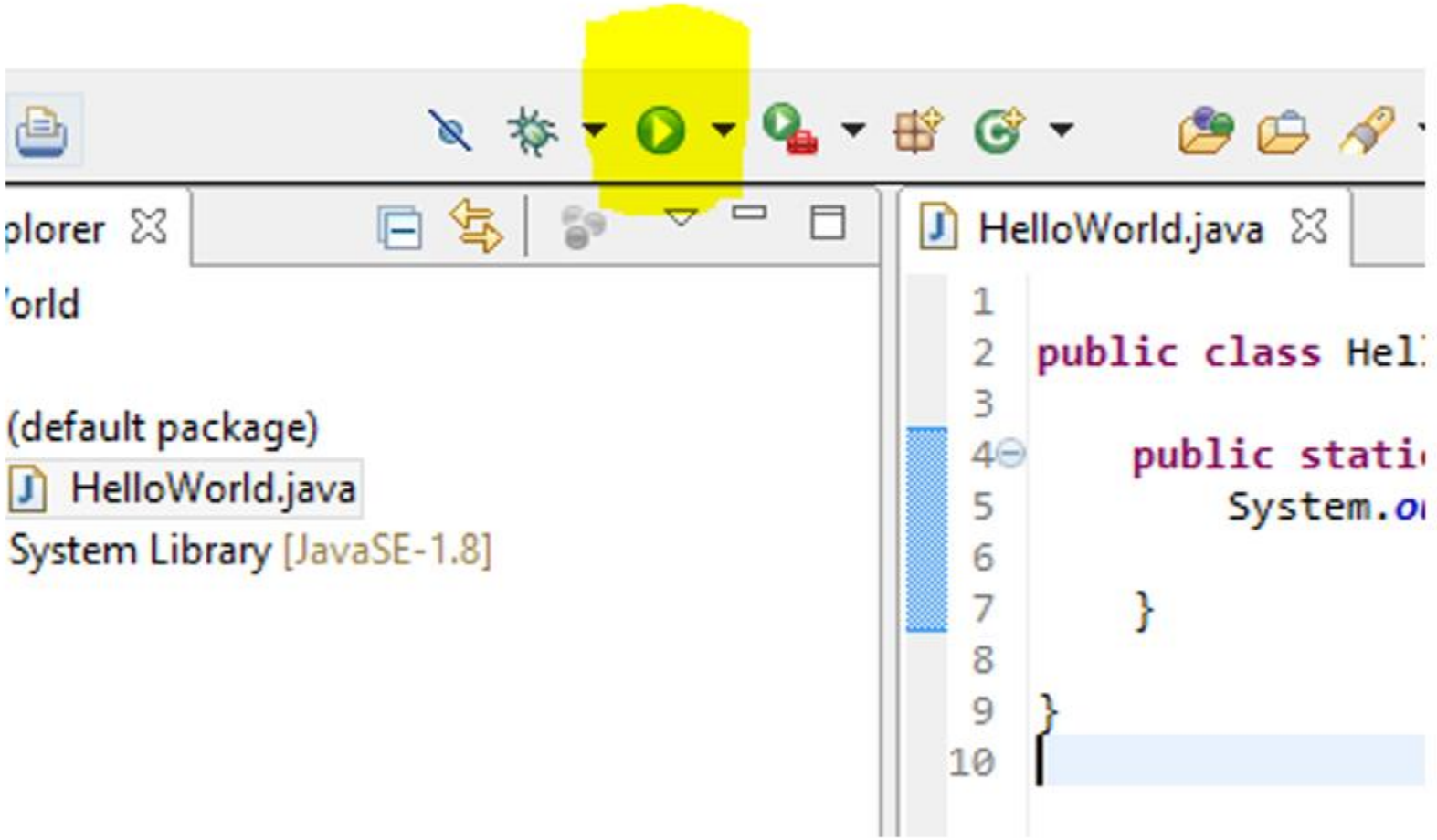
Name your project



Add new class and implement

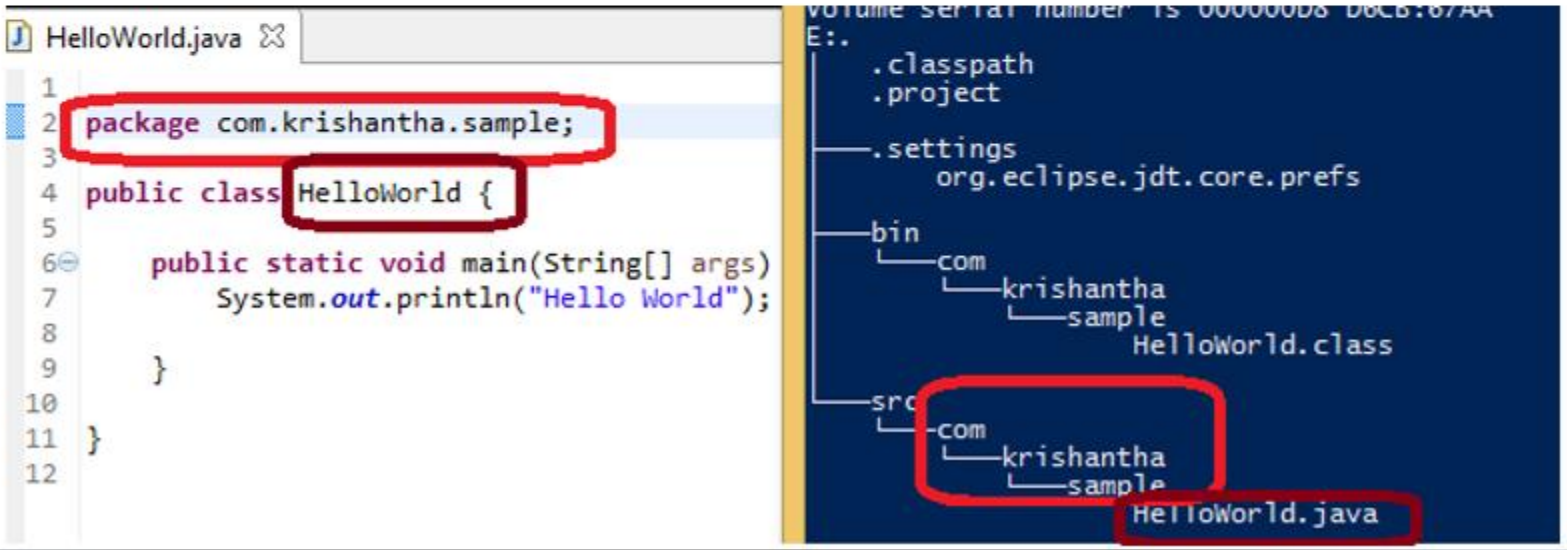


Execute java program



packages

- package is use to group similar behavior classes together.
- if you not created package its belongs to default package (no package). standerds way is reverse order is official web site. eg:
  - org.apache
  - com.sun
  - org.w3c
- when you create package it will store in respective folder in file system as well



variables

- Variable is a container that holds values that are used in a Java program. in other words variable is reserved memory location to store data.
- Every variable must be declared to use a data type. you need to understand that variable and constant are different. for example
- 10+2=12 this is constant (**not java constant**) . because you know answer is always 12
- 10+k= ???? here answer is unpredictable as its highly depends on the value of "k" meas variable
- previous introduction said variable is must declared with type. what is type?

Types

- as we learnt variable is reserved memory locations. so operation should know what going to store in order to reserve space. therefore by giving name for certain data you can advice operating system / JVM to how much space you required to store data. if you only have numbers it can be int. but if you expect to store characters you need to define type as String.
- by the way java is strongly type. that means two things.
  - you need to define data type when you declare variable
  - once you defined you cannot change it

```
package com.krishantha.sample;
 public class HelloWorld {
     public static void main(String[] args) {
        String name="Krishantha";
        System.out.println("Hello "+ name);
     }
 }
```

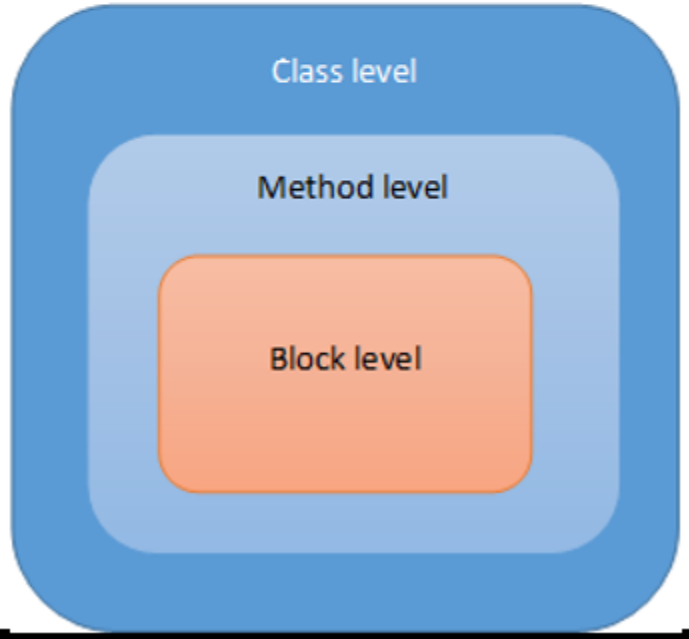
Naming variables

- Most of developers do not concern about naming variable. too much descriptive name as well as no descriptive variable name are bad. for example

  - int d=0; // no descriptive
  - int studentTotalMarkForTheGivenExam=0; // this also bad as it too much descriptive
- so you need to use some mean full name which not break above rule

scope of variable

- in simple term scope of variable mean visibility of the variable. varible can have
  - class level scope
  - method level scope
  - block level scope
- following diagram will give more clear understanding about scope of variable



Variable scope –demo

```
public class HelloWorld {

    int classLevel;

    public void demo() {

        int methodLevel;

        for (int x = 0; x < 10; x++) {
            int blockLevel;

            // all variables can be access from here
            classLevel = 0;
            methodLevel = 0;
            blockLevel = 0;

        }
        // block level not visible to upper level
        blockLevel = 0;

        // class level visible as its supper for method
        classLevel = 0;

    }
    // method level not visible to upper level
    methodLevel = 0;

}
```

Scope rule and block rule

- Scope rule
  - variable from super scope is visible to child (inner) scope but variable from child (inner) scope are not visible to super (outer) scope.
- Block rule
  - this is irrelevant to variable scope. but variable scope explain thing call “block scope” what is block? in java you can set block for any set of code that treat as single statement

Block rule - demo

```
package com.krishantha.sample;

public class HelloWorld {

    static int classLevel = 10;

    public static void main(String[] args) {

        System.out.println("before block " + classLevel);

//this is how you can all block
        {
            int classLevel = 20;
            System.out.println("inside block " + classLevel);
        }
        System.out.println("outside block " + classLevel);
    }

}
```

since “outside block” statement does not have access to value assignment inside block its still has class level assigned value.

```
before block 10
inside block 20
outside block 10
```

Primitive type

- What is primitive type is really confused question for beginners. easiest way to answer is non object data types are called as primitive types. if you need more technical answer, when you assign value to primitive type that memory hold exactly what you assigned. while object type variable hold the memory reference.

primitive types and limitations

- Integer (4)
  - byte: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays. They can also be used in place of int where their limits help to clarify your code.
  - short: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive).
  - int: By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -2^31 and a maximum value of 2^31-1. In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 [zero] and a maximum value of 2^32-1.
  - long: The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^63 and a maximum value of 2^63-1. In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 [zero] and a maximum value of 2^64-1.

- Floating points (2)
  - float: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead. Numbers and Strings covers BigDecimal and other useful classes provided by the Java platform.
  - double: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.

- Boolean (1)
  - boolean: The boolean data type has only two possible values: true and false.

- Character (1)
  - char: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).
  - there is other type call String. not a primitive type but that also treat special which plan to discuss later.

Auto boxing and Unboxing

- there are certain containers as well as method which required non primitive data types. means object types. to serve this purpose we have non primitive type belongs to each primitive types. for example int -> Integer or double -> Double. this wrapper act like box. we can put our primitive type to wrapper box and send to destination and when return came can unbox it and get primitive type back. java can handle this automatically. its call Autobox and Autounbox. means when primitive type assigned to its relevant wrapper type you don't need manually cast and java automatically do that. its call auto boxing.

Operators – by example

```
package com.krishantha.samples;
        public class OperatorDemo {
                public static void main(String[] args) {

    System.out.println("source values are 8 and 12\n");
    int result = 8 + 12;

    //+
    System.out.println("Demo \"+\" --> 8 + 12 = " + result);
    int original_result = result;

    result = result - 1;

    System.out.println("Demo \"-\" --> "+original_result + " - 1 = " + result);
    original_result = result;

    result = result * 2;

    System.out.println("Demo \"*\" --> "+original_result + " * 2 = " + result);
    original_result = result;
```

```
result = result / 2;

        System.out.println("Demo \"/\" --> "+original_result + " / 2 = " + result);
        original_result = result;

        result = result % 7;

        System.out.println("Demo \"%\" --> "+original_result + " % 7 = " + result);

        original_result=result;
        result++;
        System.out.println("Demo \"++\" --> "+original_result + "++ = " + result);

        original_result=result;
        result--;
        System.out.println("Demo \"--\" --> "+original_result + "-- = " + result);
    }

}
```

```
source values are 8 and 12

Demo "+" --> 8 + 12 = 20
Demo "-" --> 20 - 1 = 19
Demo "*" --> 19 * 2 = 38
Demo "/" --> 38 / 2 = 19
Demo "%" --> 19 % 7 = 5
Demo "++" --> 5++ = 6
Demo "--" --> 6-- = 5
```

Assignment operators – by example

```
package com.krishantha.samples;

public class AssignmentOperators {
    public static void main(String[] args){
        System.out.println("source value is 12\n");
        int result = 12;

        //+
        System.out.println("Demo \"=\" -->" + result);

        int original_result = result;
        result += 2;
        System.out.println("Demo \"+=\" --> "+original_result + " += 2 -> " + result + " (same as 12+2)");

        original_result = result;
        result -= 2;
        System.out.println("Demo \"-=\" --> "+original_result + " -= 2 -> " + result + " (same as 14-2)");

        original_result = result;
        result *= 3;
        System.out.println("Demo \"*=\" --> "+original_result + " *= 3 -> " + result + " (same as 12*3)");

        original_result = result;
        result /= 2;
        System.out.println("Demo \"/=\" --> "+original_result + " /= 2 -> " + result + " (same as 36/2)");

        original_result = result;
        result %= 7;
        System.out.println("Demo \"%=\" --> "+original_result + " %= 7 -> " + result + " (same as 18%7)");

    }
}
```

```
source value is 12

Demo "=" -->12
Demo "+=" --> 12 += 2 -> 14 (same as 12+2)
Demo "-=" --> 14 -= 2 -> 12 (same as 14-2)
Demo "*=" --> 12 *= 3 -> 36 (same as 12*3)
Demo "/=" --> 36 /= 2 -> 18 (same as 36/2)
Demo "%=" --> 18 %= 7 -> 4 (same as 18%7)
```

String & Array

- it is the most special member in java family. mean special type. String is an object type. however in java its consider something like primitive data type. also string is immutable. immutable means by default its not allowed to change and if you tried to do so it will create new object. also java has provided easy access to string means easy to use. as an example you can create string object without new keyword. (treat like primitive type). however since String is object type it has its own methods such as length() or toUpper() etc
- Array
  Array may not need to explain in details as its common to almost all programming languages. but still array is the basic concept. however since its cannot re size once created most of the time we use collections rather than arrays

Data Conversion

- since we discussed about wide range of data types we need to discuss about how we can convert those to different type.

- For example, in a particular situation we may want to treat an integer as a floating point value. impotent point is these conversions are do not change the type of a variable or the value that's stored in it. they only convert a value. need to be very careful on this as this can lead to loose the information.
- Widening conversions are always safe because they tend to go from a small data type to a larger one (such as a short to an int) so there will be enough space.
- Narrowing conversions can lose information because they tend to go from a large data type to a smaller one (such as an int to a short)

- There are 3 ways to do this

- assignment
- promotion
- casting

Assignment

- Assignment conversion means value of one type is assigned to a variable of another. for an example if distance is a float variable and radius is an int variable, the following assignment converts the value in radius to a float
- distance = radius
- Only widening conversions can happen via assignment. Note that the value or type of radias does not change.

```
package com.krishantha.samples.conversion;

public class Assignment {

    public static void main(String[] args) {
        float distance = 123_645_3344;
        int radius = 10;

        // assignment is valid
        distance = radius;

        // this is not correct
        radius = distance;
    }
}
```

Promotion

- Promotion happens automatically when operators in expressions convert their operands
- example, if sum is a float and count is an int, the value of count is converted to a floating point value to perform the following calculation:
- result = sum / count;

```
package com.krishantha.samples.conversion;

public class Promotion {

    public static void main(String[] args) {

        float sum = 123_455_344;
        int count = 10;
        // promotion
        float result = sum / count;

    }
}
```

Casting

- his is devil of conversion. means most dangerous one but its very powerfull. Both widening and narrowing conversions can be accomplished by explicitly casting a value. To cast, the type is put in parentheses in front of the value being converted As example, if total and count are integers, but we want a floating point result when dividing them, we can cast total:

- result = (float) total / count;

Class in practical

- Following class has name, attribute and methods

```java
package com.krishantha.training.oop;

import java.util.Date;

public class Employee {

    private int empId;
    private String firstName;
    private Date dateOfBirth;


    // methods

    public void attendToWork() {
        System.out.println("Employee is working");
    }

    public void applyLeave() {
        System.out.println("Employee apply for leave");
    }

}
```

Abstraction and Inheritance

```java
package com.krishantha.training.oop;

import java.util.Calendar;

public class Human {
    protected Calendar dateOfBirth;
}
```

You can see that common attribute
Moved to super class and it can
Use in child class by extend it

When this all method become abstract
We can create It as interface.

```java
package com.krishantha.training.oop;

import java.util.Calendar;

public class Employee extends Human {

    private int empId;
    private String firstName;


    // methods
    public boolean checkAge() {

        if (dateOfBirth.get(Calendar.YEAR) <= 1980) {
            return false;
        } else {
            return true;
        }

    }
```

Interface

- No any implementation details
- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.
- An interface can extend another interface, similarly to the way that a class can extend another class.

Composition – has a

```java
public class Car {

    private String model;
    private Engine engine;

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public Engine getEngine() {
        return engine;
    }

    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public Car() {
        this.engine = new Engine(1000, false);
        this.engine.start();
    }

}
```

```java
public class Engine {
    private int capasity;
    private boolean start;

    public Engine(int capasity, boolean start) {
        if (start) {
            this.start = true;
            this.capasity = capasity;

        } else {
            this.start = false;
        }
    }

    public void start() {
        start = true;
        System.out.println("Started");
    }

}
```

Composition or inheritance

- If super class changed that effect will ripple out all over sub classes. It is easier to change the interface of a back-end class (composition) than a superclass (inheritance)
- Composition allows you to delay the creation of back-end objects until (and unless) they are needed
- It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition), because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not purely true of composition

Access modifiers - default

- Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.
- A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.
- Private

Access modifiers - private

- Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.
- Private access modifier is the most restrictive access level. Class and interfaces cannot be private.
- Variables that are declared private can be accessed outside the class if public getter methods are present in the class.
- Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

Access modifiers - protected

- Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

- The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

- Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Access modifiers - public

- A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

- However if the public class we are trying to access is in a different package, then the public class still need to be imported.

- Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Overload

```
package com.krishantha.training.oop;

public class Printer {

    private String model;
    private boolean state;

    public void on() {

        if (!state) {
            state = true;
        }
    }

    // overload
    public void on(String model) {

        if (!state) {
            state = true;
            System.out.println("Model " + model + " is on now");
        }
    }

}
```

Override

```java
package com.krishantha.training.oop;

import java.util.Calendar;

public class Human {
    protected Calendar dateOfBirth;

    protected void move() {
        System.out.println("Human can move");
    }
}
```

```java
package com.krishantha.training.oop;

public class Employee extends Human {

    private int empId;
    private String firstName;

    public Employee() {
        move();
    }

}
```

Problems @ Javac

<terminated> Applicati

Human can move

```java
package com.krishantha.training.oop;


public class Employee extends Human {

    private int empId;
    private String firstName;


    public Employee() {
     move();
    }

    @Override
    public void move(){
        System.out.println("Employee can walk");
    }

}
```

Problems @ Javado

<terminated> Applicatio

Employee can walk

Note:
you can give high level of access modifiers. But you cant reduce it.

Super and this

- Super use to reference to parent class
- This mean current class

```
@Override
public void move() {

    super.move();
    System.out.println("Employee can walk");
}
```

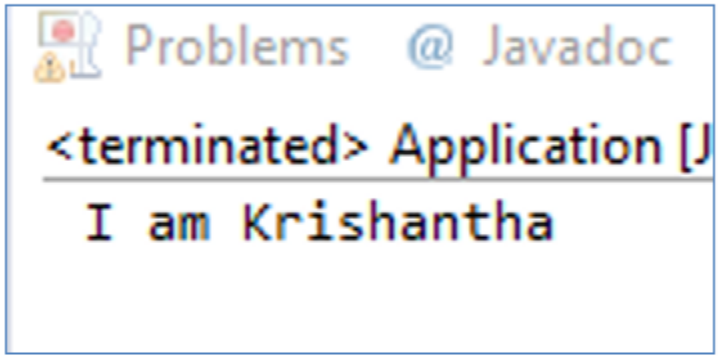
```
<terminated> Applicatic
Human can move
Employee can walk
```

```
public Employee() {

    this("no name");
}

public Employee(String name) {
    move();
}
```

Note:
You cant use super() and this () on same method as both of them wants to be first line ☹

Constructor

```
private int empId;
private String firstName;

public Employee() {

    this("no name");
}

public Employee(String name) {
    this.firstName = name;
}

public void getIdentity() {
    System.out.println(" I am " + firstName);
}
```



Note:
You can use constructor to initialize the class

Control statements

- If else
- Switch
- For
- For each
- While
- Break / continue

In simple term control statement allow to control the flow using

- Conditional
- Iterative
- jumping

If else and else if

```
If (condition){
        //Do this if true
}
Else if (this this condition){
        // do if true
}
Else{
        //do neither true
}
```

```java
package com.krishantha.training.oop;

public class ControlStatement {

    public static void main(String args[]) {

        oddEvenCalculator(4);

    }

    static void oddEvenCalculator(int number) {
        if (number == 0) {
            System.out.println("zero is invalid to calculate");
        }
        else if (number % 2 == 0) {
            System.out.println("this is even number");
        } else {
            System.out.println("this is odd number");
        }

    }

}
```

switch

- From java 7 string also allowed in switch statement
- We must break the statement once condition is true. Other wise all rest of will execute
- Also you can use multiple choices

```
static void printDistrict(String code) {

    switch (code) {
    case "CMB":
        System.out.println("Colombo");
        break;
    case "GMP":
    case "GPH":
        System.out.println("Gampaha");
        break;
    default:
        System.out.println("Code Error");
        break;
    }
```

For loop

For(initial_value; condition; modifier){

    // do what

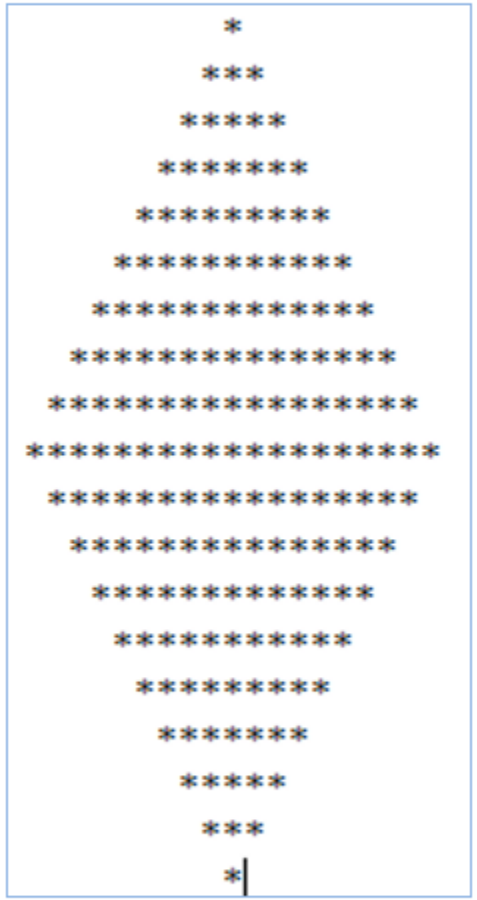
}

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);

}
```

But this is wrong.. Why?

```
for (float i = 10f; i != 0; i-=.1) {
    System.out.println(i);

}
```

Ex:



	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

For each

- For each almost like for. Difference is it going through some collection
- : is used to assign value to variable from collection

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

    String[] color = new String[] { "Red", "Green", "Blue", "Black" };

    for (String clr : color) {
        System.out.println(clr.toUpperCase());
    }

}
```

While loop

- Syntax like this

```
while(Boolean_expression)
{
  //Statements
}
```

```
public static void main(String args[]) {
    int x = 5;

    while (x < 10) {
        System.out.print("value of x : " + x);
        x++;
        System.out.print("\n");
    }
}
```

Do while

- Same as while loop. But this guaranteed to execute at least 1 time

```
do
{
  //Statements
}while(Boolean_expression);
```

```
public static void main(String args[]) {
    int x = 5;

    do {
        System.out.print("value of x : " + x);
        x++;
        System.out.println("");
    } while (x > 10);
}
```

Break and continue

- break leaves a loop, continue jumps to the next iteration.

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

    for (int i = 0; i < 10; i++) {

        if(i%2==0){
            continue;
        }
        System.out.println(i);

    }

}
```

```
1
3
5
7
9
```

Generics

- It is the way to write code in independence of type
- As example if we need house for man , dog and bird we can create house in general and convert to the way we want when use it
- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
- All generic method declarations have a type parameter section delimited by angle brackets (< >) that precedes the method's return.

Naming conventions

- Naming convention helps us understanding code easily and having a naming convention is one of the best practices of java programming language. So generics also comes with it's own naming conventions. Usually type parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are

  - E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
  - K – Key (Used in Map)
  - N – Number
  - T – Type
  - V – Value (Used in Map)
  - S,U,V etc. – 2nd, 3rd, 4th types

Generic in class

```java
package com.krishantha.training.generic;

public class Vehicle<T> {

    private T t;

    public Vehicle(T t) {
        this.t = t;
    }

    public void drive() {
        System.out.println(t.toString());
    }
}
```

```java
package com.krishantha.training.generic;

public class Car {

    @Override
    public String toString(){
        return "This is Car";
    }
}
```

```java
package com.krishantha.training.generic;

public class Application {

    public static void main(String args[]) {
        Car car = new Car();
        Vehicle<Car> vehicle = new Vehicle<>(car);

        vehicle.drive();

        Bus bus = new Bus();
        Vehicle<Bus> bvehicle = new Vehicle<>(bus);
        bvehicle.drive();

    }

}
```

```java
package com.krishantha.training.generic;

public class Bus {
    @Override
    public String toString() {
        return "This is Bus";
    }
}
```

Generic Methods

```java
package com.krishantha.training.generic;

public class PrintArray {

    public  <E> void printArray(E[] array) {

        for (E element : array) {

            System.out.println(element);

        }

    }

}
```

```java
package com.krishantha.training.generic;

public class Application {

    public static void main(String args[]) {

        PrintArray array=new PrintArray();

        Integer intArray[] ={1,20,3,45,5,66};
        String stringArray[]={"dd","ee","ff","tt"};


        array.printArray(intArray);

        array.printArray(stringArray);

    }

}
```

Bounded parameter

- With this you can restrict type to particular class or its subclass

```
public class GenericExample<T> {

    T t;

    public GenericExample(T t) {
        this.t = t;
    }

    public void printIt() {

        System.out.println(t.getClass());

    }

}
```

```
public class Application {

    public static void main(String args[]) {

        GenericExample<Integer> example = new GenericExample<Integer>(
                new Integer(10));
        example.printIt();

        GenericExample<String> example2 = new GenericExample<String>("Krisha");
        example2.printIt();

    }

}
```

This is open type

```java
public class GenericExample<T extends Number> {

    T t;

    public GenericExample(T t) {
        this.t = t;
    }

    public void printIt() {

        System.out.println(t.getClass());

    }

}
```

Now its give compile time error

```java
public class Application {

    public static void main(String args[]) {

        GenericExample<Integer> example = new GenericExample<Integer>(
                new Integer(10));
        example.printIt();

        GenericExample<String> example2 = new GenericExample<String>("Krisha");
        example2.printIt();

    }

}
```

Collection framework

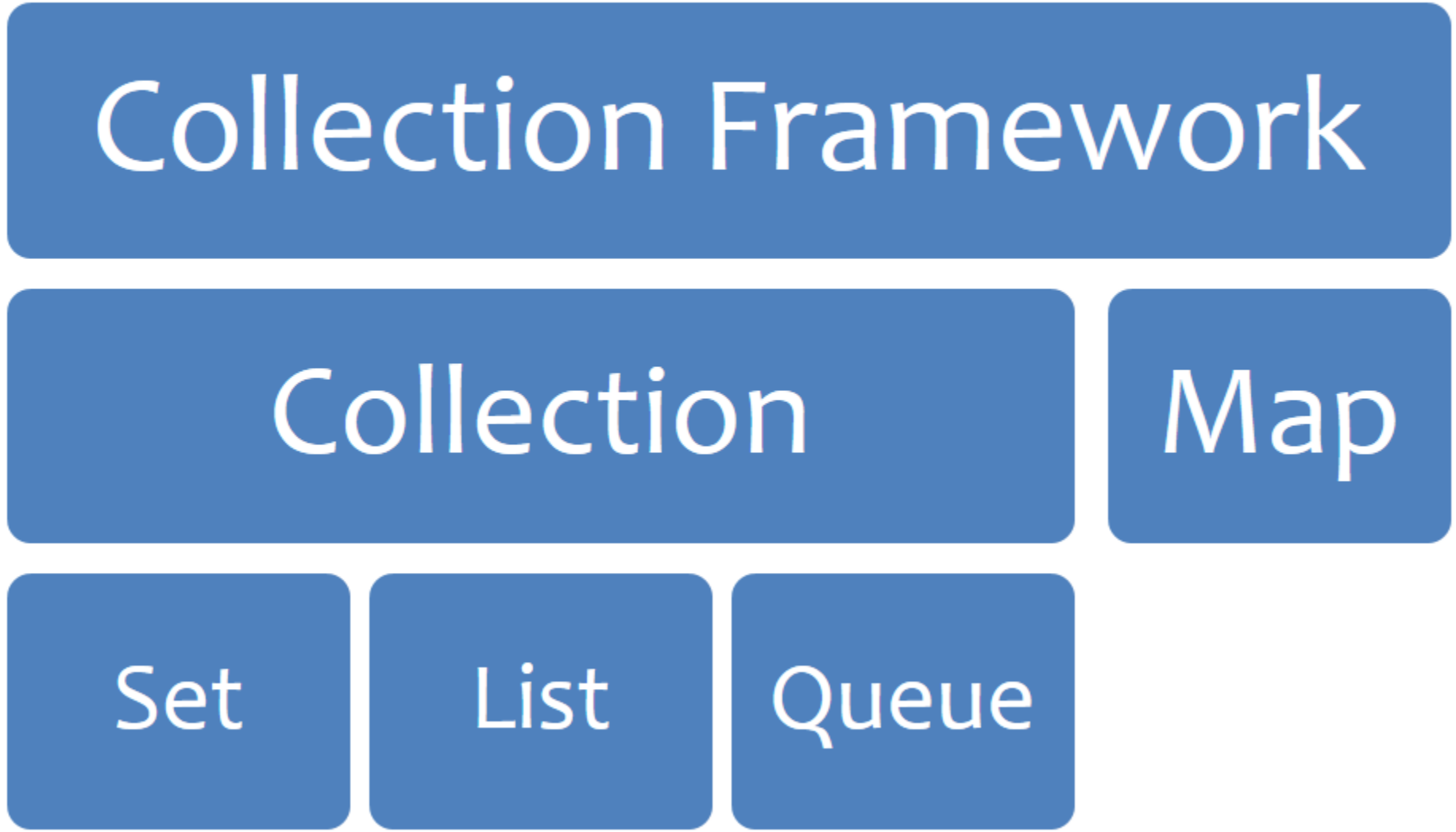
- Prior to Java 2, Java provided ad hoc classes such as Dictionary, Vector, Stack, and Properties to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used Vector was different from the way that you used Properties.

- The collections framework was designed to meet several goals.

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) are highly efficient.

- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.

- Extending and/or adapting a collection had to be easy.


Collection Framework
Collection
Map
Set
List
Queue

List

- Most common usage
- Ordered and sequential
- Can have duplicate

Array List

- The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.
- Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.
- Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.
- The ArrayList class supports three constructors. The first constructor builds an empty array list.

Array List sample

```
public class ArrayListSample {

    public static void main(String args[]) {

        List<String> list = new ArrayList<>();

        System.out.println("initial size " + list.size());

        list.add("A");
        list.add("E");
        list.add("I");
        list.add("O");
        list.add("U");

        System.out.println("after adding size " + list.size());

        list.remove(3);

        System.out.println("after remove size " + list.size());

        for (String string : list) {
            System.out.println(string);
        }

    }
```

```
initial size 0
after adding size 5
after remove size 4
A
E
I
U
```

Map

- interface
- Key value pair
- Key must be unique
- Its like dictionary
- There are implementation s such as hashMap

hashMap

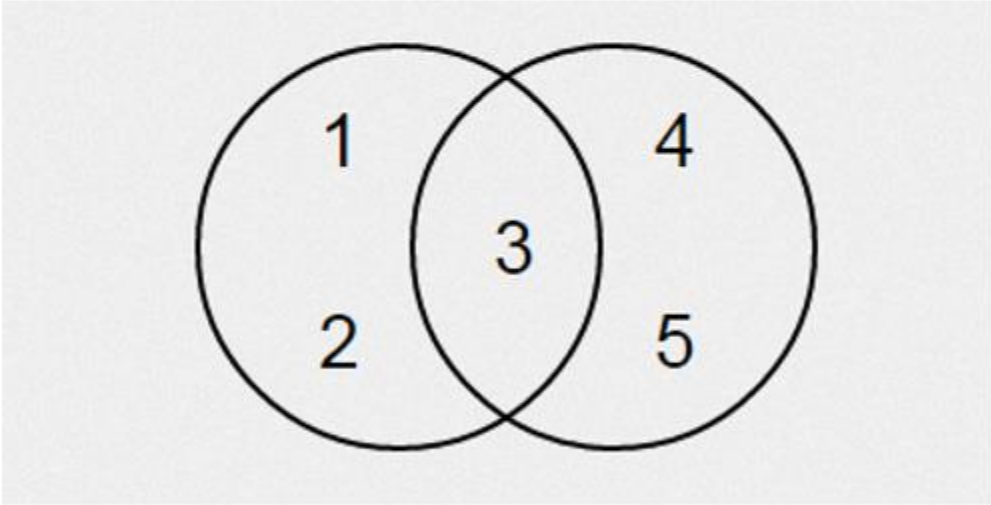
- The HashMap class uses a hashtable to implement the Map interface. This allows the execution time of basic operations, such as get( ) and put( ), to remain constant even for large sets.
- The HashMap class supports four constructors.
- No assign sequence comes and retrieve .

hashMap Sample

```
package com.krishantha.training.generic;

import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

public class HashMapSample {

    public static void main(String[] args) {

        Map<String, Integer> marks = new HashMap<>();

        marks.put("Saman", 10);
        marks.put("Kamal", 20);
        marks.put("Nimal", 40);

        for (Entry<String, Integer> mark : marks.entrySet()) {
            System.out.println(mark.getKey() + "-" + mark.getValue());
        }

    }

}
```

Set

- No duplicate allowed
- It is unique grouping
- Must be able to compare



treeSet

- TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.

- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

- The TreeSet class supports four constructors. The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements:

```java
package com.krishantha.training.generic;

import java.util.Set;
import java.util.TreeSet;

public class TreeSetSample {

    public static void main(String[] args) {

        Set<String> tree= new TreeSet<>();

        tree.add("Z");
        tree.add("A");
        tree.add("C");
        tree.add("K");
        tree.add("T");
        tree.add("M");

        System.out.println(tree);


    }

}
```

Queue

- Ordered list
- Designed for processing
- FIFO principle

Queue via linkedlist

```
package com.krishantha.training.generic;

import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {

    public static void main(String[] args) {

        Queue<String> queue = new LinkedList<>();

        queue.offer("A");
        queue.add("F");
        queue.offer("Z");
        queue.offer("B");
        queue.offer("G");
        queue.offer("K");

        while (queue.peek() != null) {
            System.out.println(queue.poll());
        }
    }
}
```

Since this is from Queue interface addFirst() addLast() methods will not be available.

Enumeration

- Its exactly listed set
- Its fully functional class and use as singleton
- This has been superseded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code.
- Should use enumeration when a variable (especially a method parameter) can only take one out of a small set of possible values.

Why enumerators

- More readable
- Type Safe
- Grouping things in Set
- Iterable

example

```java
package com.krishantha.training.enumerator;

public class EnumeratorExample {

    public static void main(String args[]) {
        System.out.println(Day.WEDNUSDAY.getDayCode());

    }

}
```

Enumerator cannot have public constructor

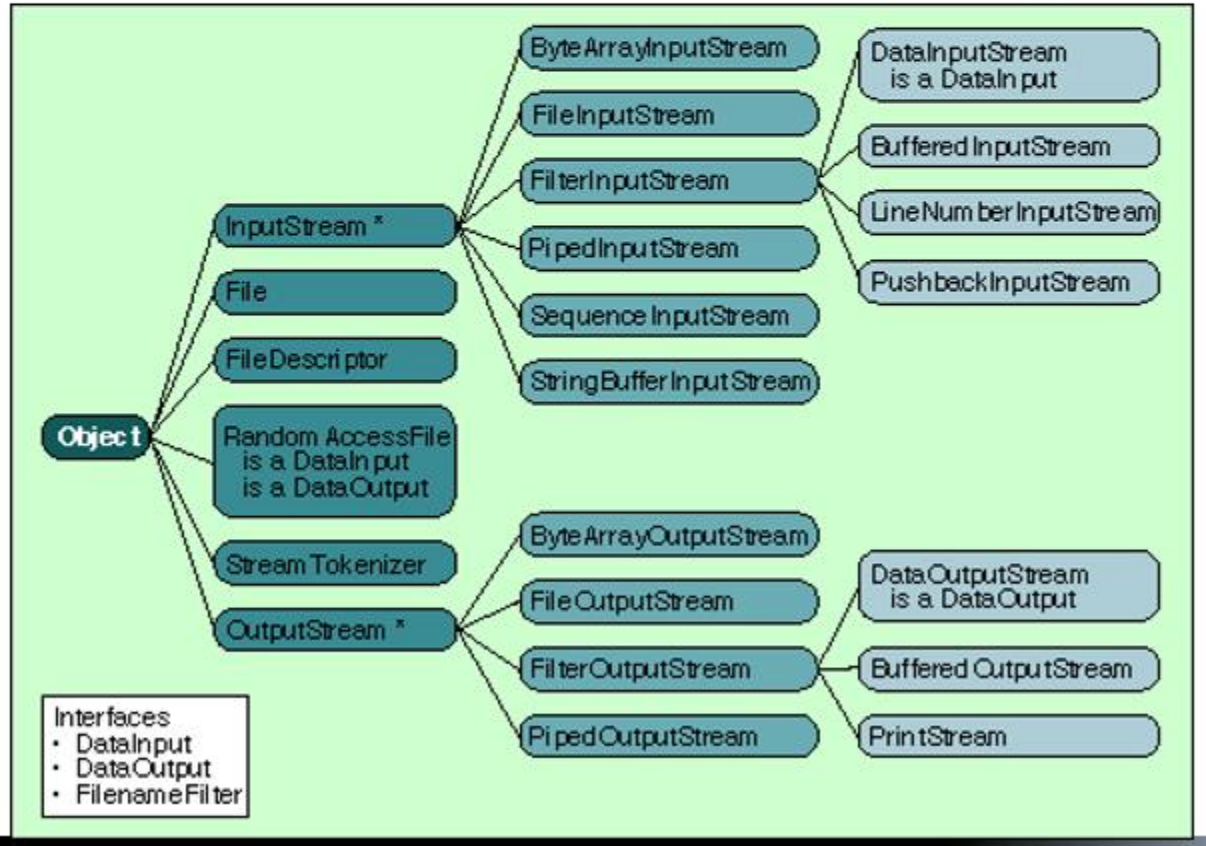
```java
package com.krishantha.training.enumerator;

public enum Day {
    SUNDAY("SUN"), MONDAY("MON"), TUESDAY("TUE"), WEDNUSDAY("WED"), THURSDAY(
            "THU"), FRIDAY("FRI"), SATURDAY("SAT");

    private String dayCode;

    private Day(String code) {
        this.dayCode = code;
    }

    public String getDayCode() {
        return dayCode;
    }

}
```

Java IO

- Instances may or may not denote an actual file-system object such as a file or a directory.
- A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
- Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

Stream

- It is like pipe. Path to flow data
- A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination



File read sample

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

    FileReader fileReader = null;
    BufferedReader bufferedReader = null;
    try {
        fileReader = new FileReader("c:\\temp\\sample.txt");

        bufferedReader = new BufferedReader(fileReader);

        String thisLine;
        while ((thisLine = bufferedReader.readLine()) != null) {
            System.out.println(thisLine);
        }
    } finally {
        if (bufferedReader != null) {
            bufferedReader.close();
        }
        if (fileReader != null) {
            fileReader.close();
        }
    }

}
```

Write sample

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

    FileWriter fileWriter = null;
    PrintWriter printWriter = null;
    try {
        fileWriter = new FileWriter("c:\\temp\\sample.txt");

        printWriter = new PrintWriter(fileWriter);

        for (int i = 0; i <= 10; i++) {
            printWriter.write("line number " + i + "\n");
        }

    } finally {
        if (fileWriter != null) {
            fileWriter.close();
        }
        if (printWriter != null) {
            printWriter.close();
        }
    }

}
```

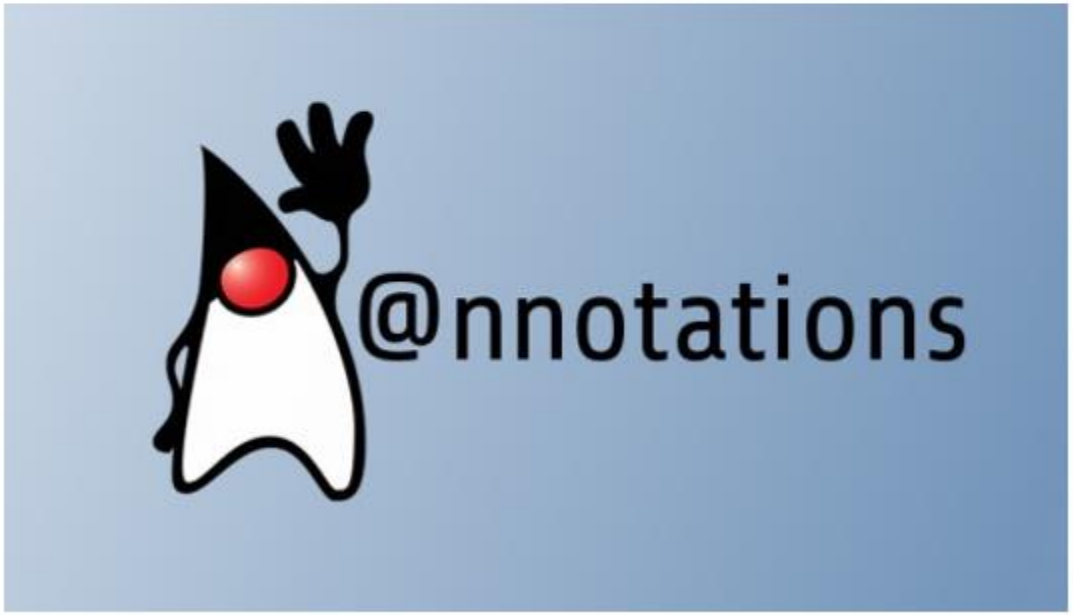
Own implementation

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

    FileReader freader = null;
    LowercaseReader reader = null;
    try {

        freader = new FileReader("c:\\temp\\sample.txt");

        reader = new LowercaseReader(freader);

        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }

    } finally {
        if (freader != null) {
            freader.close();
        }
        if (reader != null) {
            reader.close();
        }
    }

}
```

```
public class LowercaseReader extends BufferedReader {

    public LowercaseReader(Reader reader) {
        super(reader);

    }

    @Override
    public String readLine() throws IOException {
        String thisLine = super.readLine();
        if (thisLine == null)
            return null;
        else
            return thisLine.toLowerCase();
    }

}
```

Annotation

- An annotation, in the Java computer programming language, is a form of syntactic metadata that can be added to Java source code. Classes, methods, variables, parameters and packages may be annotated. (WIKI)
- Annotation is like meta data.
- Its data about data which is not directly effect to code



What are the annotation

- Data holding in class
- It can use with
  - Class
  - Field
  - Method
  - Statement
- @Repository(name="customerRepository")

Practical usage of annotation via reflection

```java
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

//this mean it available after compile
@Retention(RetentionPolicy.RUNTIME)
public @interface CarConfiguration {

    int defaultSpeed();

    String defaultMethod();
}
```

```java
@CarConfiguration(defaultMethod = "drive", defaultSpeed = 60)
public class Car {

    public void drive(int speed) {

        // your implimenation

        System.out.println("Driving at " + speed + " kmph of speed");
    }

    public void stop() {
        // your implimenation
    }

    public void start() {
        // your implimenation
    }
}
```

```java
public static void main(String[] args) throws NoSuchMethodException,
        SecurityException, IllegalAccessException,
        IllegalArgumentException, InvocationTargetException {


    Car car = new Car();

    CarConfiguration annotation = car.getClass().getAnnotation(CarConfiguration.class);
    if (annotation==null)
        System.out.println("null=====");
    //System.out.println("Start");

    //System.out.println("def:"+annotation.defaultMethod());

    Method method = car.getClass().getMethod(annotation.defaultMethod(), int.class);



    method.invoke(car, annotation.defaultSpeed());

}
```

Exceptions

- Exception handling is way to handle abnormal cases
- Exception is an object which contain data about exceptional situation
- In java all exception comes from Throwable (**what???**)
- Traditional code return error code. But this is advance than that and can handle the way we want
- Best part of this is we can throw exception to super class in case we cannot handle.
- We can use existing exception to throw our own custom exception.

existing exception to throw custom exception

- Following program will print unlimited prints if requested value falls as <0

```java
package com.krishantha.training.exception;

public class Printer {

    String model;

    public void print(int numberOfCopies){

        for(int i=numberOfCopies;i!=0;i--){
            System.out.println("printing copy "+i);
        }

    }

}
```

```java
package com.krishantha.training.exception;

public class Application {

    public static void main(String[] args) {

        Printer printer = new Printer();
        printer.print(10);

    }

}
```

Handle it

```
public class Printer {

    String model;

    public void print(int numberOfCopies)  {

        if (numberOfCopies <= 0)
            throw new IllegalArgumentException(
                    "Number of prints should be more than zero");
        for (int i = numberOfCopies; i != 0; i--) {
            System.out.println("printing copy " + i);
        }

    }

}
```

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

    try {
        Printer printer = new Printer();
        printer.print(0);
    } catch (IllegalArgumentException iae) {
        System.err.println("less than or equal zero copies not allowed");
    }

}
```

Bubble up exception

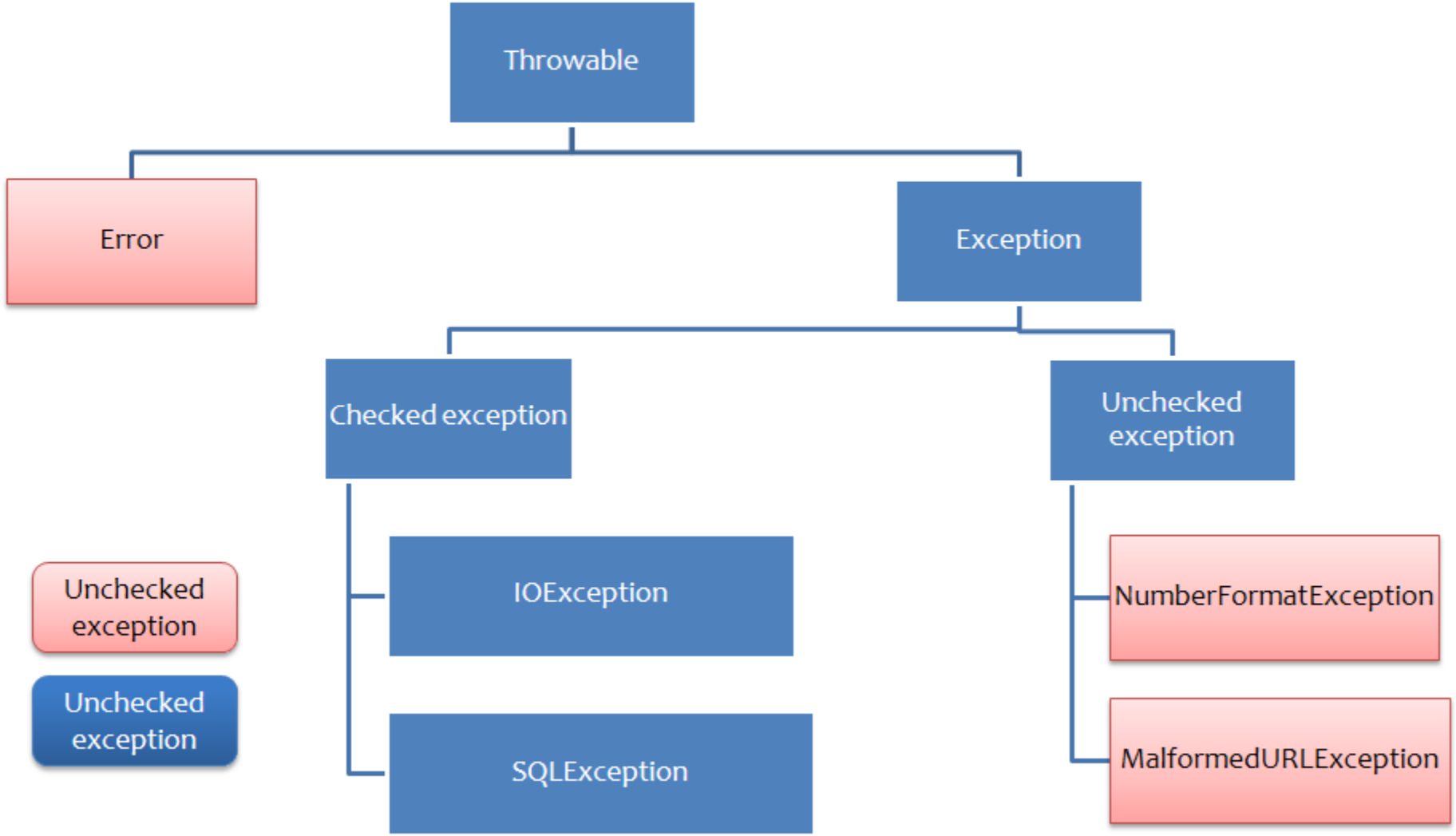
```
package com.krishantha.training.exception;

public class Printer {

    String model;

    public void print(int numberOfCopies)  {

        validateCopies(numberOfCopies);
        for (int i = numberOfCopies; i != 0; i--) {
            System.out.println("printing copy " + i);
        }

    }

    private void validateCopies(int numberOfCopies) {
        if (numberOfCopies <= 0)
            throw new IllegalArgumentException(
                    "Number of prints should be more than zero");
    }

}
```

Even though you change like this your output will be same. Because its bubble up exception Until it finds the catch or top most

finally

- This mean execute this no matter what happen.
- Its guaranteed this will execute unless program / jvm crashed
- Try is must and you can use try finally without catch block
- it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break
- The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is always recovered.
- If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

Family tree


Throwable
Error
Exception
Checked exception
Unchecked exception
IOException
SQLException
NumberFormatException
MalformedURLException
Unchecked exception
Unchecked exception

Checked and unchecked exceptions

- Checked exceptions must be explicitly caught or propagated. Unchecked exceptions do not have this requirement. They don't have to be caught or declared thrown.
- Checked exceptions in Java extend the java.lang.Exception class. Unchecked exceptions extend the java.lang.RuntimeException.
- There are many arguments for and against both checked and unchecked, and whether to use checked exceptions at all
- Checked and unchecked exceptions are functionally equivalent. There is nothing you can do with checked exceptions that cannot also be done with unchecked exceptions, and vice versa
- When you create custom exception is become checked or uncheked based on what you extent

Your exception is checked or unchecked?

```java
package com.krishantha.training.exception;

public class MyException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public MyException(String messgae) {
        super(messgae);
    }

}
```

```java
public class MyException extends Exception {

    private static final long serialVersionUID = 1L;

    public MyException(String messgae) {
        super(messgae);
    }

}
```

```java
public class Printer {

    String model;

    public void print(int numberOfCopies) {
        if (numberOfCopies <= 0)
            throw new MyException("Number of p
        for (int i = numberOfCopies; i != 0; i
            System.out.println("printing copy
        }

    }

}
```

```java
public class Printer {

    String model;

    public void print(int numberOfCopies) throws MyException {
        if (numberOfCopies <= 0)
            throw new MyException("Number of prints should be more
        for (int i = numberOfCopies; i != 0; i--) {
            System.out.println("printing copy " + i);
        }

    }

}
```

What to use? ☺ ☹ 😐

- Its highly depends with project architecture
- Nothing too best or bad
- When your exception extended from exception or when you deal with any build in checked exceptions you only have two choices.
- Either it catches the exception (MyException) or propagates it up the call stack
- Or it must handle it using catch block.
- When exception force to handle there is possibility to developer go with swallow exception
- Most of time advice you to use checked exceptions for all errors the application can recover from, and unchecked exceptions for the errors the application cannot recover from. In reality most applications will have to recover from pretty much all exceptions including NullPointerException, IllegalArgumentExceptions and many other unchecked exceptions.

Best practices

- Catch exact exception as much as possible. Don't catch super exception while child exception exist
- Exceptions are for exceptional scenarios only. So do not try to build logic or floor control with exceptions
- Don't catch errors
- Never swallow the exception in catch block (Empty catch)

Best practice detail

- **Convert checked exception to runtime exception (not always)**
  - This is one of the technique used to limit use of checked Exception in many of frameworks. This Java best practice provides benefits, in terms of restricting specific exception into specific modules, like SQLException into DAO layer and throwing meaningful RuntimeException to client layer

```
try {
 yourtask();
} catch (IOException ioe) {
 throw new CustomRuntimeException(ioe);
}
```

- Don’t catch throwable
  - Because java errors are also subclasses of the Throwable. Errors are irreversible conditions that can not be handled by JVM itself.

- When wrap exception make sure stack trace is not loss

```
//wrong
catch (NoSuchMethodException e) {
 throw new MyServiceException("<Additional info>: " + e.getMessage());
 }
 //correct
 catch (NoSuchMethodException e) {
 throw new MyServiceException("<Additional info>: " , e);
 }
```

- Either log or either throw. But don't do both
  - Logging and throwing will result in multiple log messages in log files, for a single problem in the code

- Don't catch exception if nothing going to do with it (don't eat exception)
  - Well this is most important concept. Don't catch any exception just for the sake of catching it. Catch any exception only if you want to handle it or, you want to provide additional contextual information in that exception.

```
catch (NoSuchMethodException e) {
 throw e;
}
```

- Never throw any exception from finally block
  - Then original exception will loss ☹
  - Below code is fine, as long as cleanUp() can never throw any exception. if someMethod() throws an exception, and in the finally block also, cleanUp() throws an exception, that second exception will come out of method and the original first exception (correct reason) will be lost forever. If the code that you call in a finally block can possibly throw an exception, make sure that you either handle it, or log it. Never let it come out of the finally block.

```
try {
 // Throws exceptionOne
 someMethod();
} finally {
   cleanUp();
}
```

- Declare the specific exception in throws not Exception
  - It simply defeats the whole purpose of having checked exception. Declare the specific checked exceptions that your method can throw. If there are just too many such checked exceptions, you should probably wrap them in your own exception and add information to in exception message. You can also consider code refactoring also if possible.

- Don’t use printStackTrace()
  - Since it does not have any contexual information it will not helps to anyone

- **Throw early catch later**
  - This is famous principle about Exception handling. It basically says that you should throw an exception as soon as you can, and catch it late as much as possible.
  - You should wait until you have all the information to handle it properly.
  - This principle implicitly says that you will be more likely to throw it in the low-level methods, where you will be checking if single values are null or not appropriate. And you will be making the exception climb the stack trace for quite several levels until you reach a sufficient level of abstraction to be able to handle the problem.

- **Use clean and mean full name for exception**
  - Name your checked exceptions stating the cause of the exception. You can have your own exception hierarchy by extending current Exception class. But for specific errors, throw an exception like “OrderStateInvalid” instead of “OrderException” to be more specific

- **Use relevant exception in meaningful way**
  - Relevancy is important to keep application clean. A method which tries to read a file, if throws NullPointerException when file name is empty then it will not give any relevant information to user. Validate it before fail and it will be better if such exception is wrapped inside custom exception e.g. NoSuchFileFoundException then it will be more useful for users of that method.

- Do not handle exception inside for loop

```
for (Message message:messageList) {
 try {
 // do something that can throw ex.
 } catch (YourException ue) {
 // handle exception
 }
}
//CORRECT WAY
try {
    for (Message message:messageList) {
    // do something that can throw ex.
    }
 } catch (SomeException e) {
 // handle exception
 }
```

- Don’t duplicate code. Use templates

```
public void saveData() {
 Connection conn = null;
 try{
 conn = getConnection();
 //do stuff
 } finally{
 DBUtil.closeConnection(conn);
 }
}

 public static void closeConnection(Connection conn){
 try{
 conn.close();
 } catch(SQLException ex){
 throw new RuntimeException("Cannot close connection", ex);
 }
 }
}
```

Static initialize

- Guaranteed only one time executed

```java
public class Rose {

    public static final long RED = 0xFF0000;

    public String name;
    public long color = RED;

    public Rose(String name, long color) {

        this.name = name;
        this.color = color;
    }

    public String toString() {
        return "name: " + this.name + "-" + "color: " + this.color;
    }
}
```

```java
public class RoseGarden {

    public static ArrayList<Rose> roses;

    static {
        System.out.println("init");
        // anycode here will execute class loader
        roses = new ArrayList<>();
        roses.add(new Rose("RedRose", 0xFF0000));
        roses.add(new Rose("GreenRose", 0x00FF00));
        roses.add(new Rose("BlueRose", 0x0000FF));
    }
}
```

```java
public class Application {

    public static void main(String args[]) {
        System.out.println("caller");
        ArrayList<Rose> roses = RoseGarden.roses;
        for (Rose rose : roses) {
            System.out.println("its " + rose.name);
        }
    }
}
```

```
caller
init
its RedRose
its GreenRose
its BlueRose
```

Before constructor

```
public class Rose {

    public static final long RED = 0xFF0000;

    public String name;
    public long color = RED;

    public Rose(String name, long color) {

        this.name = name;
        this.color = color;
    }

    public String toString() {
        return "name: " + this.name + "-" + "color: " + this.color;
    }
}
```

```
public class RoseGarden {

    public ArrayList<Rose> roses;

    {
        System.out.println("init");
        // anycode here will execute class loader
        roses = new ArrayList<>();
        roses.add(new Rose("BlueRose", 0x0000FF));
    }

    public RoseGarden() {
        System.out.println("constructor fired");
    }

    public RoseGarden(String name, int nor, long color) {

        for (int i = 0; i < nor; i++) {
            roses.add(new Rose(name, color));

        }
    }
```

```
public class Application {

    public static void main(String args[]) {
        System.out.println("caller");
        ArrayList<Rose> roses = new RoseGarden("Red",3,0x000000).roses;
        for (Rose rose : roses) {
            System.out.println("its " + rose.name);
        }
    }
}
```

```
caller
init
its BlueRose
its Red
its Red
its Red
```

If you need to execute some code which no matter which constructor fired. Then this is the way

Member classes (encapsulation)

- Add rose class in to garden class and add new method

```
public RoseGarden(String name, int nor, long color) {

    for (int i = 0; i < nor; i++) {
        roses.add(new Rose(name, color));
    }
}
public void addRose(String name, long color) {

    roses.add(new Rose(name, color));
}
public void getRoses() {

    for (Rose rose : roses) {
        System.out.println("that is " + rose.name);
    }
}

class Rose {
    public static final long RED = 0xFF0000;
    public String name;
    public long color = RED;

    public Rose(String name, long color) {

        this.name = name;
        this.color = color;
    }

    public String toString() {
        return "name: " + this.name + "-" + "color: " + this.color;
    }
}
```

```
public class Application {

    public static void main(String args[]) {
        System.out.println("caller");

        RoseGarden garden = new RoseGarden();
        garden.addRose("Red", 0xff0000);
        garden.addRose("Green", 0x00ff00);
        garden.addRose("Blue", 0x0000ff);

        garden.getRoses();

    }
}
```

If class is only available in other class we can use this

Class in side method. (inner class)

- This is sound crazy. But there are cases we need this
- This is follow the principle of encapsulation
- If you need to define class for build some complex process and you want to hide it from rest of the application also if this class use only one time this is the good way to do it
- You cannot use static members in this context
- You do not need access modifies are its by default private since it is inside the method

```java
public void getRoses() {

    class RoseValidator {

        public void validate(String name) {
            if (name != null && name != "") {
                if ("Green".equalsIgnoreCase(name)
                        || "Blue".equalsIgnoreCase(name)) {
                    System.out.println("Validation success");
                } else {
                    System.out.println("invalid color");
                }
            } else {
                System.out.println("Validation fail due to null or empty");
            }
        }
    }

    new RoseValidator().validate("Blue");

    for (Rose rose : roses) {
        System.out.println("that is " + rose.name);
    }
}
```

Anonymous class

- This is further step of previous example.
- In this case its further limiting to that class can be use only when just after it create
- Its based on inheritance principle and when you don't have super class you can use object

```java
public void getRoses() {

    new Object() {

        public void validate(String name) {
            if (name != null && name != "") {
                if ("Green".equalsIgnoreCase(name)
                        || "Blue".equalsIgnoreCase(name)) {
                    System.out.println("Validation success");
                } else {
                    System.out.println("invalid color");
                }
            } else {
                System.out.println("Validation fail due to null or empty");
            }
        }
    }.validate("BLUE");

    for (Rose rose : roses) {
        System.out.println("that is " + rose.name);
    }

}
```

Reflection API

- This can give the feature of dynamic instantiation
- Can use to highly dynamic development
- Also can use dynamic invocation and also able to inspect other code in the same system
- This is a relatively advanced feature **and should be used only by developers who have a strong grasp of the fundamentals of the language**
- Reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible

Disadvantages of Reflection

- Performance Overhead
  - Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
- Security Restrictions
  - Reflection requires a runtime permission which may not be present when running under a security manager. This is in an important consideration for code which has to run in a restricted security context, such as in an Applet.
- Exposure of Internals
  - Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.