**Source File Declaration Rules:**

●One public class per source code file

●If there is a public class in the file, the name of the file must match the public class name

●If the class is the part of a package, the package statement should be the first line of the source code before any import statements.’

●If there are import statements, they must go between the package name (if there is one) and class declaration

**Java Program Compilation And Execution:**

**Using The javac and java commands:**

**Compiling With Javac:**

The javac command is used to invoke java’s compiler. You can specify many options when running javac.

And what are those options like:

For instances, there are options to generate debugging information or compiler warning.

javac [options] [source-files]

Some of the examples of javac command:

Javac -help

Javac -version foo.java Bar.java

The first invocation does not compile any files, but prints a summary of valid options.

The second invocation passes the compiler an option, (-version, which prints the version of the compiler they are using) and passes the compiler two java files to compile, (foo.java and bar.java). Whenever, you are specifying multiple options, they must be separated by spaces.

**Launching Applications With Java: java command:**

The java command is used to invoke the java virtual machine.

**Import:**

In Java, the import statement is used to bring certain classes or the entire packages, into visibility. As soon as imported, a class can be referred to directly by using only its name. (Whether the package or class is library package or class, or user defined package or class, does not matter)

The import statement is a convenience to the programmer and is not technically needed to write complete Java program. If you are going to refer to some few dozen classes into your application, the import statement will save a lot of time and typing also.

**Static Import:**for instance, consider the following sample program:

**import static java.lang.System.out;**

**import static java.lang.Integer.\*;**

**public class TestStaticImport**

**{**

**public static void main(String[] args)**

**{**

**out.println(MAX\_VALUE);**

**out.println(toHexString(42));**

**}**

**}**

Let’s look what’s happening in the code that’s using the static import.

* Even through the feature is static import, the syntax must be import static followed by the fully qualified name of the static member you want to import. Or, you could use wildcard. (**like, import static java.lang.Integer.\*; here, \* is the wildcard)**
* What does static import actually mean? For instance, **import static java.lang.Integer.\*;** it says I want to do static imports of all the static members.
* Now, we are fully seeing the benefits of static import features. First, it imports all the static members, second, We did not have to type System in System.out.println. Second thing, we don’t need to type the Integer in Integer.MAX\_VALUE. So, in this line, we were able to use a short cut for a static\_method and a constant.

(However, in the last case, where we don’t have to import the integer in Integer.MAX\_VALUE, watch out for the ambiguously named static members. For instance, if your program does a static import for both the classes Integer and Long, referring to the MAX\_VALUE will cause a compiler error. Since, for both classes static definition of MAX\_VALUE is present and compiler does not know which MAX\_VALUE you are referring.)

**Wildcard Concept In import:**

As you have seen, when using import and import static statements, sometimes you can use the wildcard character \* to do the simple searching (for a function or constant) for you. **(with the use of \* character, you can search through a package or within a class). you can say this:**

**import java.util.\*; //ok, to search the whole java.util packages**

In a similar vein, if you want to search the java.lang.Integer class for static members, you can say that:

**import static java.lang.integer.\*;**

But, you cannot create broader searches. For instance, you cannot use an import to search through the entire java API;

**import java.\*;**

**Data Types In Java:**

**Primitive Data Types:**



Note, In java, char is of 2 bytes.

**User Defined Data Type:**Probably enum is one of the user defined datatype. As of java 5, java lets you restrict a variable to have one of the predefined values. In other words, one value from an enumerated list.

Using enum can help in reducing the bugs in your code.

For instance, in your coffee shop application, you might want to restrict your coffee size size selections to BIG, HUGE, and OVERWHELMING. If you let and order for a **LARGE** or **GRANDE** slip in, it might cause an error.

An enum can be defined as the following:

**enum coffeesize={BIG, HUGE, OVERWHEELMING};**

It’s not required that enum constants be all in caps, but borrowing from the Oracle code conventions, **that constants are named in caps**, it’s a good idea.

**Now, enum could be declared out side of a class, it could be declared within a class as a class member, or enum can be declared as their own separate class.**

**An Example: (when enum is declared outside of any class)**

enum CoffeeSize{BIG, HUGE, OVERWHELMING}

//no semicolon at the end

//this cannot be private or protected

class Coffee

{

CoffeeSize size;

}

public class CoffeeTest1

{

public static void main(String args[])

{

Coffee drink=new Coffee();

drink.size=CoffeeSize.BIG;

System.out.println("The drink size is: "+drink.size);

}

}

**Note the following things:**

1. Both class Coffee and CoffeeTest1 is in the same package. No access specifier is specified to the class Coffee as well as it’s members. Hence, the scope, the package default. For class Coffee as well as the variable size in it. Hence,   
     
   drink.size=CoffeeSize.BIG can be accessed.
2. We have a enum outside the class’s scope.
3. The way in which we can access one of the newly defined values.
4. note that thing. Java language designers make it optional to put a semicolon at the end of the enum declaration. So, what gets created when you make an enum? The most important thing to remember that an enum is not string or int. Each of the enumerated CoffeeSize typesare actually an instance of CoffeeSize. Think of an enum as a kind of class that looks something like this (not exactly though)

**class CoffeeSize**

**{**

**public static final CoffeeSize BIG=new CoffeeSize(“BIG”,0);**

**public static final CoffeeSize HUGE=new CoffeeSize(“HUGE”,1);**

**public static final CoffeeSize OVERWHELMING=new CoffeeSize(“OVERWHELMING”,”2”);**

**CoffeeSize(String enumName, int index)**

**{**

**//stuff here**

**}**

**}**

**How can we know more about it?**

public enum Constants {

ONE,

TWO,

THREE;

}

Compiling the above enum and disassembling resulting class file with javap gives the following: (Now, javap disassembles the machine code generated, as disassembling (Compiling the above enum and disassembling resulting class file with javap gives the following)

Compiled from "Constants.java"

**public final class Constants extends java.lang.Enum{**

**public static final Constants ONE;**

**public static final Constants TWO;**

**public static final Constants THREE;**

**public static Constants[] values();**

**public static Constants valueOf(java.lang.String);**

**static {};**

**}**

The disassemble shows that that each field of an enum is an instance of the Constants enum class. (Further analysis with javap will reveal that each field is initialized by creating a new object by calling the new Constants(String) constructor in the static initialization block.)

Therefore, we can tell that each enum field that we create will be at least as much as the overhead of creating an object in the JVM.

**Declaring Constructors, Methods, Variables In An Enum:**

Because, enum is a special kind of class, you can do more than just list the enumerated constant values. You can add constructors, instance variables, methods and something really strange known as a constant specific class body. To understand, why you might need more in your enum, think about the particular scenario: imagine you want to know the actual size, in ounces, that map to the three CoffeeSize constants. Now, you could make some kind of lookup table using some other data structures. But that will be a poor design and hard to maintain. The simplest way to treat your enum values as objects, each of which can have its own instance variables and own values.

enum CoffeeSize

{

BIG(8), HUGE(10), OVERWHELMING(16);

private int ounces;

CoffeeSize(int ounces)

{

this.ounces=ounces;

}

public int getOunces()

{

return ounces;

}

}

public class Coffee

{

CoffeeSize size;

public static void main(String[] args)

{

Coffee drink1=new Coffee();

drink1.size=CoffeeSize.BIG;

System.out.println("In "+drink1.size+" we get "+drink1.size.getOunces());

}

}

Which produces: In BIG we get 8

There are some points to know:

●You can never invoke an enum constructor directly. The enum constructor is invoked automatically, with the arguments you defined after the constant value.

●You can define more than one argument to the constructor, and you can overload the enum constructors. Just as you overload a normal class constructor.

**Variable Declarations:**there are two types of variables in java.   
  
**Primitives:** a **primitive** can be one of the eight types. Char, boolean, short, int, long, double or float. Once, a primitive has been declared, its primitive type can never be changed. Although in most cases, its value can be changed.

**Reference Variables:** a reference variable is used to refer to an object. A reference variable is declared to be a specific type and that type can never be changed. A reference variable can be used to refer to any other objects of the declared type or of a subtype of the declared type.

**Packages In Java:**

**There are two kinds of packages:**

1) User defined package: The package we create is called user-defined package.

2) Built-in package: The already defined package like java.io.\*, java.lang.\* etc are known as built-in packages.

**If a class is under a package, the statement package …; should be the first statement.**

**Consider the following example:**

package cert;

public class sludge

{

public void testit(){System.out.println(“Sludge”)};

}

In this class, **package cert;** should be the first statement. All imports come later.

**Sub packages in Java**

A package inside another package is known as sub package. For example If I create a package inside letmecalculate package then that will be called sub package.

Lets say I have created another package inside letmecalculate and the sub package name is multiply. So if I create a class in this subpackage it should have this package declaration in the beginning:

**package letmecalculate.multiply;**

**Note:** If you don’t declare a package for a class, it will be part of default package.

java is a package centric language, the developers assumed that for **good organization and for named scoping**, you would put all your classes into packages. And, this is right, Otherwise, consider the following situation. Three different programmers which are in the same company but working on different projects,define their own utilities class. Now, if those classes are not declared in any of the explicit package, and are in the class path, there is no way to tell JVM or compiler which one you are trying to reference.

Oracle suggests/recommends that developer use **reverse domain names appended with division and/or project names. For example, if your domain name is anonymous.com your package name should start with com.anonymous.**

**Class Declarations And Modifiers:**

Class modifiers are fall into two types:

**Access Modifiers. (public, private, protected)**

**Non access modifiers. (strictfp, final and abstract)**

**Access Modifier:**Now, though there is three access modifiers, public, private and protected, there are four levels of access controls. As the fourth one is default or package access when you don’t use any of the chosen access modifiers. (private, protected, public)

However, all the four access modifiers are for class variables and functions. For a class, there is two access modifiers. **Public and Default**

**What does that mean? A class has only two access specifiers, while a variable can have all the four access specifiers.**

**Public access modifier:**

When a method or variable member is declared public, it means all other classes, regardless of the package they belong to, can access the member. (Assuming that the class itself is visible)

**package cert;**

**public class sludge**

**{**

**public void testit(){System.out.println(“Sludge”)};**

**}**

**package book;**

**import cert.\*;  
class Goo**

**{**

**public static void main(String args[])**

**{**

**sludge o=new sludge();**

**O.testIt();**

**}**

**}**

As you can see, Goo and sludge are in different packages. However, Goo can invoke the method in sludge without problems, because, both the sludge class and it’s testIt() method are made public.

**But, if you make the following changes, it will not even compile:**

**Goo.java**

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package book;

import cert.\*;

public class Goo

{

public static void main(String[] args)

{

Sludge o=new Sludge();

o.testIt();

}

}

**Cert.java**

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package cert;

/\*\*

\*

\* @author Reve

\*/

public class Sludge

{

//it's access specifier is made package default

void testIt()

{

System.out.println("Sludge");

}

}

You cannot even compile it.

It will give the compilation error for following line in Goo.java

**o.testIt(); as testIt has default access.**

**Now, if you change the class definition to the following:**

**Goo.java**

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package book;

import cert.\*;

public class Goo

{

public static void main(String[] args)

{

Sludge o=new Sludge();

o.testIt();

}

}

**Cert.java**

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package cert;

/\*\*

\*

\* @author Reve

\*/

class Sludge

{

//it's access specifier is made package default

public void testIt()

{

System.out.println("Sludge");

}

}

It will give compilation error, too. But, in this line:

**Sludge o=new Sludge();**

Note: main must be contained under a public class. (Thus, every java project **(does not matter if it consists of even one file should contain at least one public class)**

**Private access modifiers:**

Members marked private cannot be accessed by code in other class other than the class in which the private member is declared.   
  
  
**Now, Note that,** a private method of a super class cannot be overridden by a subclass since, it is not inheriting it.

**Now, note that,** private access modifier cannot be applied to a class

**Protected Access modifiers:**

The protected and default access control levels are almost identical. But, with one critical difference. A default member may be accessed only if the class accessing the member belongs to thee same package. Whereas, a protected member can be accessed (through inheritance) by a subclass if the subclass is in a different package.

**Any class could only see the protected members through inheritance.**

**Non Access Modifiers Of A Class:**

strictfp, final and abstract. (what is native? Native is a modifier used on a function name. But it finally uses the native interface to reuse the functions defined in other languages)

**Final Class:**when used in a class declaration, the final keyword means the class cannot be sub classed. In other words, no other class can ever extend a final class. And trying to do so will generate **Compilation error. In other words, no other class can ever extend.** You should make a final class only if you need an absolute guarantee that none of the methods in that class will ever be overridden.

**Many classes in java core libraries are final. Like, String class.** Imagine the havoc if you could not guarantee how a string object would work on any given system your application is running on.

**However, in practice, we will almost never make a final class.** A final class obliterates a key benefit of OO -extensibility. So, unless if you have a serious safety or security issue, you should not do this.

**Final Functions:**

prevents a method from being overridden in a subclass.

**Final Arguments:**

Final keyword can also be used in case of an argument. The concept is similar to const argument in c++. **A new value cannot be assigned to the variable. If that is passed ass a final argument.**

**Abstract Class:**

An abstract class can never be instantiated. So, it’s purpose is it has to be extended.

**So, conceptually where it is useful?**

Imagine you have a class car that has generic methods common to all vehicles. But, you don’t want anyone to actually create a generic, abstract class object.

Or, the bank account example.

**abstract class Car**

**{**

**private double Price;**

**private String model;**

**private String year;**

**private abstract void goFast();**

**private abstract void goUpHill();**

**private abstract void impressNeighbours();**

**}**

**Some points about abstract class:**

* Even a single method is abstract in a normal class, that class has to be defined as abstract.
* However, you can have non abstract methods in a abstract class. For example, you might have methods that should not change from Car Type to Car Type such as getColor() or setPrice(). **By putting non-abstract methods in a abstract class, you give all concrete subclasses inherited method implementation.**
* **An interface cannot be abstract**
* **A variable cannot be abstract.**

**Strictfp modifier:**

**strictfp** is a keyword in java used for restricting floating-point calculations and ensuring same result on every platform while performing operations in the floating-point variable.  
Floating point calculations are platform dependent i.e. different output(floating-point values) is achieved when a class file is run on different platforms(16/32/64 bit processors). To solve this types of issue, strictfp keyword was introduced in JDK 1.2 version by following [IEEE 754](https://en.wikipedia.org/wiki/IEEE_floating_point) standards for floating-point calculations.

**Important points:**

* strictfp modifier is used with classes, interfaces and methods only.

**strictfp class Test**

**{**

**// all concrete methods here are**

**// implicitly strictfp.**

**}**

**strictfp interface Test**

**{**

**// all methods here becomes implicitly**

**// strictfp when used during inheritance.**

**}**

**class Car**

**{**

**// strictfp applied on a concrete method**

**strictfp void calculateSpeed(){}**

**}**

* strictfp modifier cannot be used with variables.
* When a class or an interface is declared with strictfp modifier, then all methods declared in the class/interface, and all nested types declared in the class, are implicitly strictfp.
* strictfp cannot be used with abstract methods. However, it can be used with abstract classes/interfaces. **(since, only concrete methods can be strictfp)**
* Since methods of an interface are implicitly abstract, strictfp cannot be used with any method inside an interface. **(Because, strictfp can only be used with concrete functions)**

**Example 1:**

**strictfp interface Test**

**{**

**double sum();**

**double mul();**

**}**

It is allowed

**Example 2:  
  
 strictfp interface Test**

**{**

**double sum();**

**strictfp double mul(); // compile-time error here**

**}**

It is not.

**Example 3:**

//Java program to illustrate strictfp modifier

**public class Test**

**{**

**// calculating sum using strictfp modifier**

**public strictfp double sum()**

**{**

**double num1 = 10e+10;**

**double num2 = 6e+08;**

**return (num1+num2);**

**}**

**public static strictfp void main(String[] args)**

**{**

**Test t = new Test();**

**System.out.println(t.sum());**

**}**

**}**

**User Interfaces:**When you create an interface, you are defining what a class can do without saying how the class will do it.

Any class type that implements the interface must write code for all methods.

**Difference between interface and abstract class:**

Now, abstract class-inheritance-is a relationship, Whereas, modular kitchen is implemented by normal kitchen

However, except the theoretical part, technically consider interface as a 100% abstract class

**Some Points For Interfaces:**

* All interface methods are implicitly public and abstract. In other words, you do not need to type public or abstract modifiers in the method declaration, but the method will always be public and abstract.
* All variables defined in an interface must be public, static and final. **(and these modifiers are not implicit. You have to mention it).** In other words, **interfaces can only have constants, not instance variables.**
* However, unlike, the variables, which should be **public, static and final,** interfaces method should not be static.
* Because, interface methods are abstract, they cannot be marked as final.
* **An interface can extend one or more other interfaces.**
* An interface cannot extend anything but another interface.
* **An interface cannot implements another interface or class.**
* An interface must be declared with the keyword interface

Like the following:  
 **public interface bouncable**

**{**

**void bounce();**

**void setbouncefactor(int bf);**

**}**

* Because, interface methods are abstract (implicitly), we cannot use **final, strictfp or native modifiers with them.**Final methods cannot be abstract.

strictfp modifer must be applied on concrete functions

native keyword is used to declare a function which is defined elsewhere. It cannot have a combination with abstract.

* Interface types can be used polymorphically.
* **Further Note:** the following is a legal interface declaration:

**public abstract interface Rollable()**

**{**

**//the variables (which must be static, public and final)**

**//the methods (which are implicitly public and abstract)**

**}**

However, typing in the **abstract modifier** is considered redundant.

Static