**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.