Java Certification
A Few Guidelines

In the IT domain, an individual's knowledge is today measured by the certifications he or she holds. Employers, and sometimes clients, insist on these. If you are fascinated with the Java-2 Enterprise Edition (J2EE) technology, here are some general tips that will help you get the Sun Certified Java Programmer (SCJP) certification—a first step in demonstrating your mastery in Sun technologies.

DR SANJAY GUPTA

The author works as an architect in Interops Solutions, Wipro Technologies. He has published research papers in various international journals. He can be reached at sanjay.gupta@wipro.com

A systematic approach and right guidance is compulsory to achieve the Sun Certified Java Programmer (SCJP) certification. Before diving into the technical pool, I will take you into confidence by disclosing that I am a Sun Certified Java Programmer (SCJP) and also a Sun Certified Trainer. All the technical information about the Sun Certified Programmer for Java 2, Platform 1.2 and 1.4 is available at Sun's site http://suned.sun.com/US/certification/java/java_progj2se.html.

If you see the exam objectives on the given Sun site, the syllabus is given in the form of sections, for example, Section 1 talks about 'Declarations and Access Control'. The objective of my article, however, is not to teach you the Java language, but to take you through the elementary and finer aspects of Java from the SCJP exam point of view and to help you in acquiring the coveted Sun Certification. Before going directly to the subject, let us begin with a basic understanding of the Java Language Fundamentals and how it is different from the C++ language.

I assume that all of you must have worked with at least one basic programming language like C, C++ or the like. C as a programming language, I feel, is the most powerful language as it is very close to the machine language. You have the power to control any portion of the memory by using the strong concept of 'Pointers'. With time, as the paradigm changed from Structural programming to Object Oriented Programming, C++ hit the market hard and became an instant hit. As software engineers, we started to feel that this is the language, which has the capability to do everything that we want to do. But as the requirements and technologies changed, a new concept, 'platform independence', emerged, and there was a need for a unique language, which has it. James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan then conceived a language with the name 'Java'.

If you have worked with C and C++, you

might have felt some difficulty to get a control on pointers. Pointers are strong and can increase the program efficiency manifold, if used properly. Wrong use of pointers could leave you with a crashed system. Using pointers, you can assign memory, so you are also responsible for de-allocating the memory once it is not in use. If you forget to de-allocate the memory, it can cause 'memory leaks' or 'memory corruption'. As a Java developer, I am not at all bothered with the tough concepts of pointers and as such I am not responsible for allocating the memory, and de-allocating it. Java comes with a very strong component known as GC (Garbage Collector), which de-allocates the objects in use in memory.

Java is platform-independent. We can write the code, compile it and run it on any platform (OS) till you have a Java Interpreter corresponding to that platform. Java is an interpreted language—first we compile it to the 'byte codes' and then these byte codes are interpreted to the machine level language. I am not wrong if I say that byte codes are platform-independent. Let us summarise what I have discussed till now about Java. It is object oriented, platform-independent, and blessed with GC, which takes care of de-allocating the unused objects from the memory. There are no pointers; here we create the objects by using the keyword new. GC does de-allocation of the objects automatically.

When we say that Java is an object-oriented language, it means that it supports the basic features of object orientation, namely, Abstraction, Encapsulation, Polymorphism and Inheritance.

'Abstraction' means hiding the information from the end user in which he/she is not interested. Take the example of remote control for your TV set. To operate the remote, it is not required for a user to know the internal circuitry. We as end users are only interested in changing channels or the volume level. In the programming language, we achieve abstraction by declaring the variables with a 'private' access modifier. A good object-oriented design says, keep your important variables (data) private and access them through 'public' methods (known functions in C).

'Encapsulation', as the name suggests, is the art of keeping data members and the methods together. A Class definition itself is the best example of this.

'Polymorphism', as the name suggests, is a property by which the same name can be used for different purposes. Java supports polymorphism by the concepts of method overloading and method over-riding.

The last property, 'Inheritance', is a very important property and is also known as reusability. It is not necessary for you to design and code the same class or functionality again and again from scratch. If you feel that you have a well-tested class, which can be used later on in another class, then you can inherit your new class from the base class by using the keyword 'extend'. Java does not support 'multiple inheritance' but you can have multiple inheritance by using interfaces. A class can extend from only one class but it can implement many interfaces, that is,

```
public class child extend Base implements
Interface1, Interface2, Interface3...
```

We use a keyword *'implements'* to implement any interface in our class definition. We will discuss interfaces in detail later on.

If you are writing a class where you want to confine this within a package and want to import some other statements (packages like *java.io*, *java.awt* etc), you may also want to implement some interfaces in this program. If this is your requirement, then it is important for you to know the right order of these declarations. The correct order of the statements is:

- Package declaration
- Import statements like *'import java.io;'*
- Class/Interface declarations, either public or non-public, in any order.

that is,

```
// This is a comment
package mypackage;
import java.io.*;
class MyClass { }
```

It is important for you to know that there can be one and only one package declaration for a single Java file, if it is required. You cannot have declarations for two packages in a single file, which means:

```
package mypackage1;
package mypackage2;
import java.awt.*;
class MyClass { }
```

The above class declaration defines two packages and will give you a compiler error. Also, if I make a class declaration like:

```
import java.io.*;
package pkg;
class MyClass { }
```

it will give a compile time error, as the import statement is coming before the package declaration.

There can be one and only one public class defined in a Java file, and the file name and the public class name should match with each other. Since every program begins the execution with the main method, it is important for a programmer to understand precisely how to define it.

If you have worked with Java, you will be aware that the first program of any programming language displays *'Hello All'*. Let us write the syntax of the *'Hello All'* program and discuss it in detail.

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

Have you ever thought why this *'Hello'* program is the first program in any book? Is it a cosmetic requirement or is there a deep meaning attached to this? According to me, if you understand any *'Hello'* program you will know that it gives you a deep insight into that language.

Let me start explaining this code. The first line, *public class Hello,* tells that we have created a class definition with the name *Hello* and it is declared as *public*. So it can be accessed from anywhere—within the same class, same package (directory) or from another package. The next line *'public static void main (String [] args)'* tells about the definition of the *main ()* function. You know that in any language the execution of the program starts with the *main ()* function. Java is a strict language. You will appreciate this by seeing that it does not allow any code to be free. Even the definition of *main ()* method is confined within the class definition. The *main ()* method is declared as *public*, as it can be called from anywhere. It is *static* as it is the class level member and it can be called even before creating any object of this class. It is declared *void*, as the *main ()* does not return anything. Its most important use is to execute the program. If you see the parameter list of the *main (),* it is an array of type *String. String* is a well-defined class in Java—if you confine any text within a double code it is treated by Java as string, that is, 'My First String'. You can pass the arguments to a Java program from the command line, typing them immediately after the class name. Suppose in the same *'Hello'* program, I add one more line saying *'System.out. println (args [1]);'* and execute this program with the command *java Hello How are you*. What will the output be? Just think! Readers, who have a background in C, may say that it will print *'How'*, as the program name is considered as the zero argument in C. But in Java the first passing parameter is the zeroth element, so in this case the output will be *'are'*.

Assume that the following program has been compiled.

```
public class MyClass {
 public static void main (String args [ ]){
int n=1;
System.out.println ("My name is " + args [n]);
}
   }
```

Select the correct command line to execute the program and produce the following output line: *My name is Sanjay*.

A) *MyClass Ajay Sanjay Vijay Rahul*
B) *java MyClass Ajay Vijay Sanjay Rahul*
C) *java MyClass Ajay Sanjay Rahul*
D) *java MyClass.class Ajay Sanjay Rahul*

The option C is right; as here *Ajay* is the argument number zero. *Sanjay* is argument number one and *Rahul* is argument number two. In the above code, we are going to print the argument number one.

The different correct syntax for main method are:

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

The next line *'System.out.println (" Hello All");'* is a statement to print any data on the console. *System.out* is a PrintStream. You will learn more about PrintStream once I discuss Input/Output in Java. One important point to remember is that there may be "only one public class in a single code".

Java supports 52 keywords. These keywords are otherwise known as reserved words, because they cannot be used for any variable, method or class name. You should remember all these keywords, as there will be at least one question on this, say for example, what are the valid keywords in Java?

Java keywords

abstract	continue	float	long	short
true	boolean	default	for	native
false	static	try	break	goto
new	super	void	byte	double
catch	null	switch	char	const
volatile	case	else	implements	do
package	synchronized	while	if	extends
import	private	this	class	int
instanceof	protected	throw	final	strictfp
public	throws	finally	interface	return
transient	assert			

One important point to remember is that all the keywords start with small case letters. I can simply confuse you by asking whether *instanceOf* is a valid keyword or not? And the majority will say it is wrong as 'O' is capital in *instanceOf*. The keywords *goto* and *const* are known as reserved keywords, as they are not in use but still are part of Java keywords.

I will start my technical discussion with data types in Java. If you compare the data types in Java with any other programming language like C or C++, it is the same. The main difference here is that all the data types are allocated with a fixed amount of memory, no matter what OS we are working on. The amount of memory allocated to any data type is fixed across all the OSs. We have eight data types in Java and they are:

```
byte (1byte), short (2 bytes), int (4 bytes), long (8 bytes), float (4 bytes) double (8 bytes), char (2 bytes) and boolean (1 byte)
```

Here, one important thing to remember is that intrinsic type casting is not done in Java, if you are going from a higher side to the lower side. This means that if you want to assign a *byte* or *short* variable to the *integer* or *long*, it will work fine but if you want to assign an *integer* variable to a *short* type, it will give you a compiler error. For example,

```
int a =10;
byte b=a;
System.out.println (b); // compile time error
```

Another point to remember here is that any decimal number is treated as a *double* number. So if you want some variable as *float*, post fix it by f or F. If you write *float f= 2.3;* it will give you a compile time error, as in this case 2.3 is treated as *double*. But *double* is big (8 *bytes*), and you will put a big quantity (8 *bytes*) in a small container, which has a capacity of 4 *bytes*. So to run this code without errors, write *float f = 2.3f;* and it will work well.

Default values of data types

Data Type	Initial Value
byte	0
int	0
float	0.0f
char	'\u0000'
object reference	null
short	0
long	0L
double	0.0d
boolean	false

Data Type Range

Data Type	Range
boolean	true or false
byte	-128 to 127
char	0 to (2^{16} -1)
short	-2^{15} to (2^{15}-1)
int	-2^{31} to (2^{31}-1)
long	-2^{63} to (2^{63}-1)

These points seem very simple but are important from the examination viewpoint. This section will give you quite good marks if you do not commit any mistake. If you want to convert *char* to *byte* type, first convert *char* to *int* and then you can type cast it to *byte* or *short*. You cannot convert *char* type to *short* or *byte* directly. A *boolean* value is either *true* or *false* and it cannot be type cast to any other data type.

To name any variable in Java, there are certain rules. It is important for you to know these to distinguish between legal and illegal identifiers, and they are:

- Except for the keywords you can give any name to an identifier.
- Identifiers must begin with a character, a dollar sign ($), or an underscore (_).
- Subsequent characters can be any characters, digits, $, or _.

So which of the following will be a valid identifier for Java variables?

A) my_variable
B) _yourvariable
C) $money
D) _3data
E) %path

A, B, C and D are valid identifiers but E will give compile time error as it is starting with the % sign. The variable name can start with only one special character, and that is the $ sign.

By now I am sure you've got a basic feel of Java. Now, try finding the answers to the following:

- What if you remove [] in the definition of *main ()* parameter list, I mean *public static void main (String args);* will it give compile time error or run time error?
- What will happen if I make this line as *public static main void (String args);* will it work, or give any error?