Reymart C San Buenaventura

BS-IS 2

1. Brief history of java

**Java history** is interesting to know. The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

**James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

Originally designed for small, embedded systems in electronic appliances like set-top boxes.

Firstly, it was called **"Greentalk"** by James Gosling and file extension was .gt.

After that, it was called **Oak** and was developed as a part of the Green project.

In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies. Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995. In 1995, Time magazine called **Java one of the Ten Best Products of 1995**. JDK 1.0 released in(January 23, 1996).

Java Version History

There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)
11. How is java platform independent?

|  |  |
| --- | --- |
| Cept | Typically, the compiled code is the exact set of instructions the CPU requires to "execute" the program. In Java, the compiled code is an exact set of instructions for a "virtual CPU" which is required to work the same on every physical machine.  So, in a sense, the designers of the Java language decided that the language and the compiled code was going to be platform independent, but since the code eventually has to run on a physical platform, they opted to put all the platform dependent code in the JVM.  This requirement for a JVM is in contrast to your Turbo C example. With Turbo C, the compiler will produce platform dependent code, and there is no need for a JVM work-alike because the compiled Turbo C program can be executed by the CPU directly.  With Java, the CPU executes the JVM, which is platform dependent. This running JVM then executes the Java bytecode which is platform independent, provided that you have a JVM availble for it to execute upon. You might say that writing Java code, you don't program for the code to be executed on the physical machine, you write the code to be executed on the Java Virtual Machine.  The only way that all this Java bytecode works on all Java virtual machines is that a rather strict standard has been written for how Java virtual machines work. This means that no matter what physical platform you are using, the part where the Java bytecode interfaces with the JVM is guaranteed to work only one way. Since all the JVMs work exactly the same, the same code works exactly the same everywhere without recompiling. If you can't pass the tests to make sure it's the same, you're not allowed to call your virtual machine a "Java virtual machine".  Of course, there are ways that you can break the portability of a Java program. You could write a program that looks for files only found on one operating system (cmd.exe for example). You could use JNI, which effectively allows you to put compiled C or C++ code into a class. You could use conventions that only work for a certain operating system (like assuming ":" separates directories). But you are guaranteed to never have to recompile your program for a different machine unless you're doing something really special (like JNI). |

1. Difference between JDK, JRE, JVM.

Jvm, Jre, Jdk these are all the backbone of java language. Each components work separately . Jdk and Jre physically exists but Jvm is an abstract machine that means it has not physically exists.

**JVM**

JVM (Java Virtual Machine) is a software. It is a specification that provides runtime environment in which java bytecode can be executed. It is not physically exists.

JVMs are not same for all hardware and software, for example for window os JVM is different and for Linux VJM is different. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

**JRE**

The Java Runtime Environment (JRE) is part of the Java Development Kit (JDK). It contains set of libraries and tools for developing java application. The Java Runtime Environment provides the minimum requirements for executing a Java application. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

**JDK**

**JDK**  stands Java Development Kit it contain all necessary components which used in programming like class, methods, swing, AWT, package, java (interpretor), javac (compiler), appletviewer(applet application viewer) etc. So final conclusion is it content every file which useful in developing an application weather it standalone or web based.

1. Why is the main() method declared static?

Now come to the main point *"Why the main method is static in Java"*, there are quite a few reasons around but here are few reasons which make sense to me:

1. Since the main method is static Java virtual Machine can call it without creating any instance of a class which contains the main method.

2. Since C and C++ also have similar main method which serves as entry point for program execution, following that convention will only help Java.

3. If main method were not declared static than JVM has to create instance of main Class and since constructor can be overloaded and can have arguments there would not be any certain and consistent way for JVM to find main method in Java.

4. Anything which is declared in [class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html) comes under reference type and requires object to be created before using them but static method and static data are loaded into separate memory inside JVM called context which is created when a class is loaded. If main method is static than it will be loaded in JVM context and are available to execution.

5. When the JVM makes are call to the main method there is not object existing for the class being called therefore it has to have static method to allow invocation from class

1. Is java purely object oriented? Explain.

**Java is not a pure Object oriented language, but so called a "Hybrid" language.**  
For any language to be pure object oriented it must follow these 6 points strictly...  
**1) It must have full support for Encapsulation and Abstraction**  
**2) It must support Inheritance**  
**3) It must support Polymorphism**  
**4) All predefined types must be Objects**  
**5) All user defined types must be Objects**  
**6) Lastly, all operations performed on objects must be only through methods exposed at the objects.**  
  
Now, java supports 1, 2, 3 & 5 but fails to support **4 & 6.**   
In  java we have some predefined types as non-objects (primitive types).  Although we have wrapper classes for the same but a Pure OOL can't have  anything other than Objects strictly.   
And regarding point 6, In java we can have communicate with objects without calling their methods for e.g. using arithmetic operators.  
String s1 = "ABC" + "A”;  
These 2 points stops java from being a pure OOL.  
  
Why java doesn't design primitive data   types as object way ?  
Back in the 90's there were Performance reasons and at the same time  Java stays backward compatible. So they cannot take them out.  
  
Also James Gosling answered this. He said that they'd liked to have totally  abstracted away from primitives - to leave only standard objects, but  then ran out of time and decided to ship with what they had.