Oracle announced the general availability of Java SE 10 (JDK 10) on Mar 20, 2018.

We can download Production-ready binaries of JDK 10 under the GPL from:

1. **Oracle** [ <http://www.oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html> ]
2. **OpenJDK [** <http://jdk.java.net/10/> ]

**What is JEP?**

According to wikipedia “The **JDK Enhancement Proposal** (or **JEP**) is a process drafted by [Oracle Corporation](https://en.wikipedia.org/wiki/Oracle_Corporation) for collecting proposals for enhancements to the [Java Development Kit](https://en.wikipedia.org/wiki/Java_Development_Kit) and [OpenJDK](https://en.wikipedia.org/wiki/OpenJDK).”

**What is JSR?**

According to Java Community Process([www.jcp.org](http://www.jcp.org)) “**Java Specification Requests (JSRs)** are the actual descriptions of proposed and final specifications for the Java platform. At any one time there are numerous JSRs moving through the review and approval process”.

**JDK 10 have several new features (referred as JEPs)**

1. **Local-Variable Type Inference**

Java is a type language. Every variable and value must have a type. The type compatibility must have checked on compile time as well as runtime.

In JDK 10 we can declare a local variable without specifying its type, just using by **var.**

See the example below**:**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Java10 {

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

var x= 30;

System.***out***.println("Infer as int "+x);

var y= "shadab";

System.***out***.println("Infer as String "+y);

// Before JDK 10

List<String> list1 = **new** ArrayList<>();

list1.add("JavaSE");

list1.add("JavaEE");

list1.add("JavaME");

System.***out***.println(list1);

// In JDK 10

var list2 = **new** ArrayList<>();

list2.add("JavaSE-7");

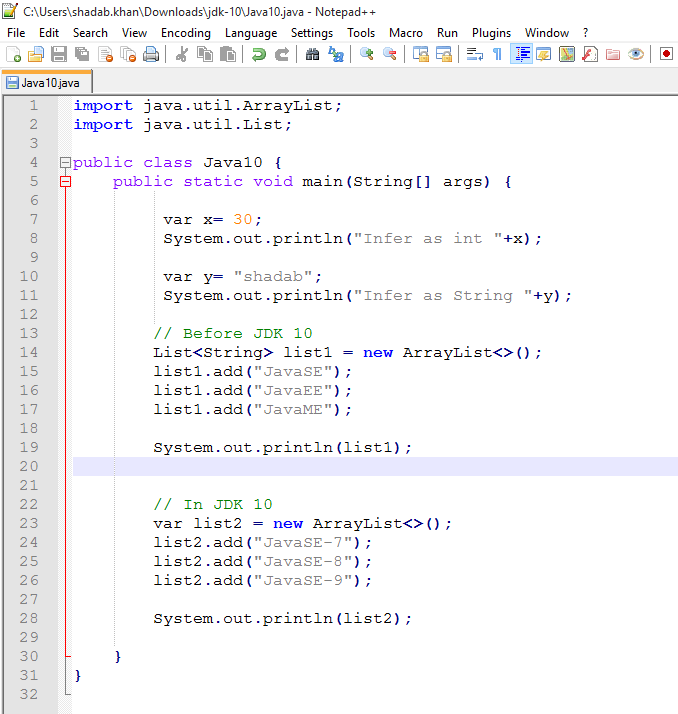
list2.add("JavaSE-8");

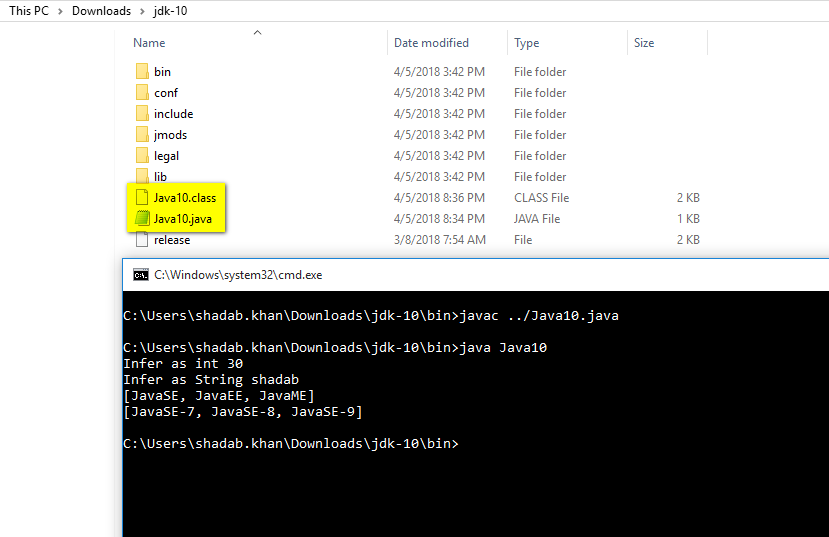
list2.add("JavaSE-9");

System.***out***.println(list2);

}

}





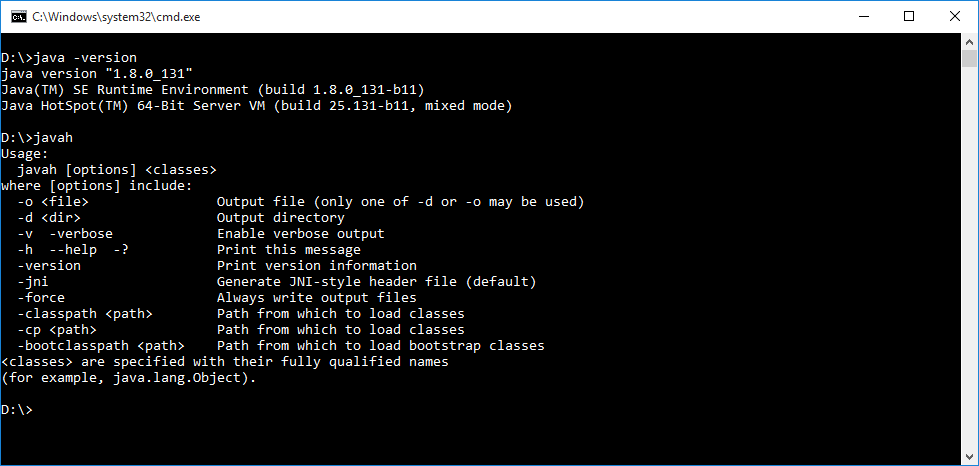
This is called type inferring means Java know the type by expression result assigned to the variable, so no need of boiler-plate code for local variable initialization and it increases code readability, seems good!

Some Important points about “**var**”:

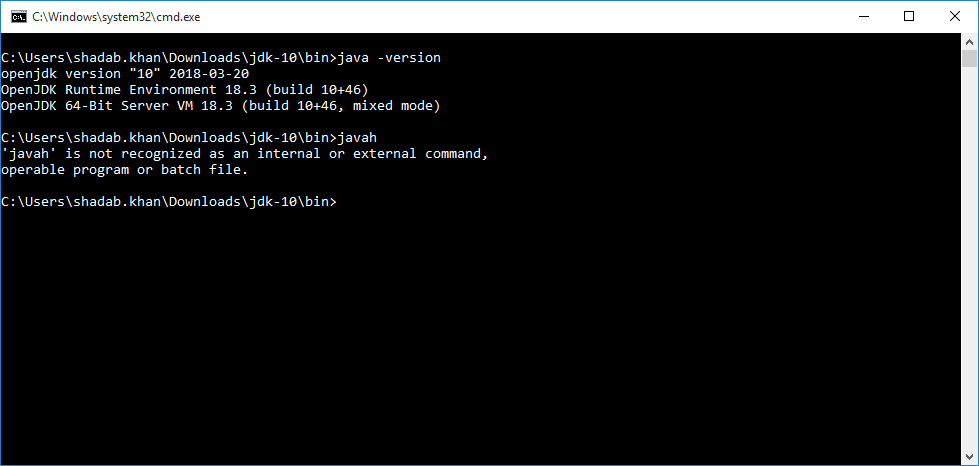
1. The identifier var is not a keyword or reserved word, it is *reserved type name*. So what is reserved type name means? It means any developer can use var as name of variable, method and/or package name but cannot use as class/interface name.
2. In case of variable only local variable like:
   1. Local variables declared in method body.
   2. Indexes in the enhanced for-loop, and locals declared in a traditional for-loop.
3. Where we cannot use var as local variable:
   1. Formal parameters of method.
   2. Formal parameters of constructor.
   3. Formal parameters of catch block.
   4. Return type of a method
4. **Consolidate the JDK Forest into a Single Repository**
5. JDK preserve their code in multiple numerous repositories.
6. In JDK 9 there are eight repos: root, corba, hotspot, jaxp, jaxws, jdk, langtools, and nashorn.
7. JDK forest combined all multiple repos into a single repository in order to simplify and streamline development.
8. Model of multiple repos has many downsides and does a poor job of supporting various desirable source-code management operations.
9. In particular, it is not possible to perform an atomic commit across repositories of inter-dependent changesets. For example, if the code for a single bug fix or RFE spans both the jdk and hotspot repos today, the change to both repositories cannot be done atomically.
10. **Remove the Native-Header Generation Tool (javah)**

Sorry! javah tool is no more in JDK 10, yes the javah tool is removed from the JDK10.

**On java Se 8(As I preinstalled it):**



**On java SE 10:**



1. **Garbage-Collector Interface**
2. This will improve the source code isolation of different garbage collectors by introducing a clean garbage collector (GC) interface.
3. Better modularity for HotSpot internal GC code.
4. Due to code modularity it is easier to exclude a GC from a JDK build.
5. The GC interface would be defined by the existing class CollectedHeap which every garbage collector needs to implement. The CollectedHeap class would drive most aspects of interaction between the garbage collector and the rest of HotSpot
6. **Parallel Full GC for G1**
7. This feature improves G1 worst-case latencies by making the full GC parallel.
8. The G1 garbage collector was made the default in JDK 9. The previous default, the parallel collector, has a parallel full GC. To minimize the impact for users experiencing full GCs, the G1 full GC should be made parallel as well.
9. **Application Class-Data Sharing**
10. Class-Data Sharing (CDS) first introduced in Jdk5 for improving JVM startup time.
11. In JDK 10, this feature is enhanced by allowing the built-in system class loader, the built-in platform class loader, and custom class loaders to load shared archived classes.
12. **Thread-Local Handshakes**
13. This is very low-level JVM change.
14. This feature makes JDK10 both cheap and possible to stop individual threads instead of all or none thread(s).
15. **Heap Allocation on Alternative Memory Devices.**
16. With the evolution of time, both software and hardware get evolves rapidly.
17. In JDK 10, now it possible to use non-volatile RAM with same interface along similar performance characteristics to conventional DRAM
18. With the availability of cheap NV-DIMM memory, future systems may be equipped with heterogeneous memory architectures. One example of such technology is Intel's 3D XPoint.
19. This features enables the HotSpot VM to allocate the Java object heap on an alternative memory device, such as an NV-DIMM, specified by the user.
20. **Experimental Java-Based JIT Compiler**
21. This JEP says that a larger part of JVM will rewritten in Java, Currently it in in C++.
22. This feature enables the Java-based JIT compiler named Graal (compiler research project).
23. It is used experimentally on the Linux/x64 platform.
24. It will not beat the performance of the existing JIT compilers.

**10. Root Certificates**

1. Some security components like TLS now work by default in OpenJDK builds due to default Root Certificate.
2. This is a beneficial add-on and is probably will become the part of the work Oracle is undertaking to ensure that OpenJDK binaries and Oracle JDK binaries are functionally the same.
3. Open-source the root certificates in Oracle's Java SE Root CA program in order to make OpenJDK builds more attractive to developers, and to reduce the differences between those builds and Oracle JDK builds.