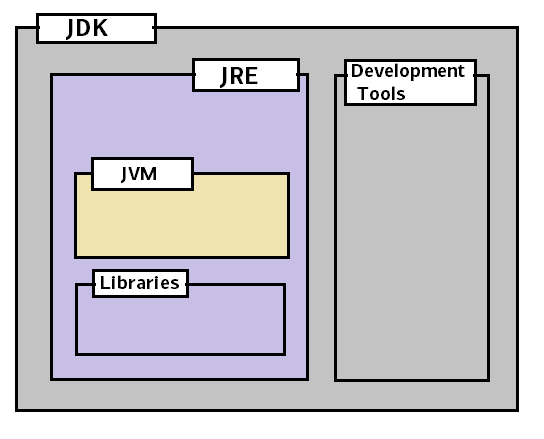
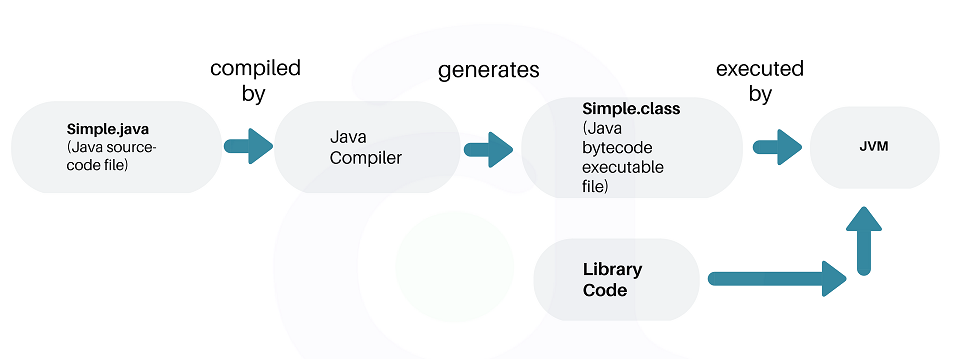
# Introduction

What is Java?

* Java is concurrent, object-oriented, and is intended to let application developers “write once, run anywhere” (WORA).
* WORA is achieved by compiling a Java program into an intermediate language called bytecode. The format of bytecode is platform independent. A virtual machine, called the Java Virtual Machine (JVM), is used to run the bytecode on each platform.



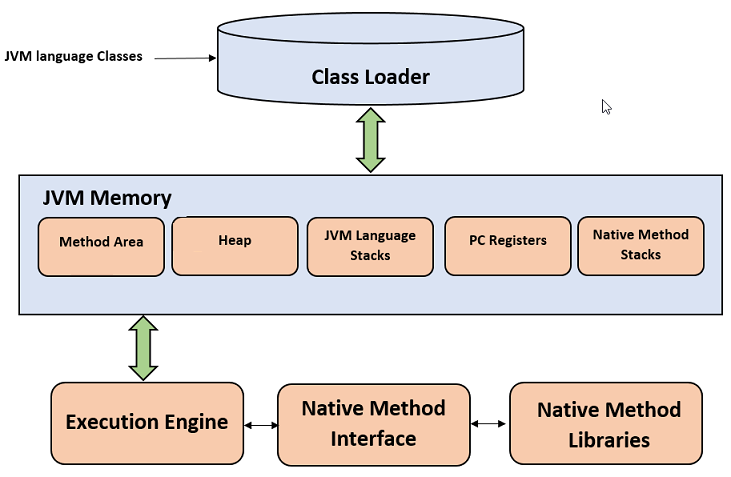
* JVM translates bytecode into native machine code which machines can execute.



## JVM

* Java Virtual machine (JVM) is the virtual machine that runs the Java bytecodes.
* In the real world, JVM is a specification that provides a runtime environment in which Java bytecode can be executed. Different vendors provide different implementations of this specification.
* JVM delivers the optimal performance for Java applications using many advanced techniques, incorporating a state-of-the-art memory model, garbage collector, and adaptive optimizer.
* JVM comes in two different flavors – client and server. Developers can choose which system they want by specifying -client or -server.
* The JVM is called virtual because it provides a machine interface that does not depend on the underlying operating system and machine hardware architecture.

### JVM Architecture



#### Class Loader

The class loader is a subsystem used for loading class files. It performs three primary functions, i.e. class loading, linking, and initialization.

1. **Loading**

* To load classes, JVM has 3 kind of class loaders. **Bootstrap**, **extension** and **application** class loader.
* When loading a class file, JVM finds out a dependency for some arbitrary class XYZ.class

1. First bootstrap class loader tries to find the class. It scans the rt.jar file in JRE lib folder
2. If class is not found, then extension class loader searches the class file in inside jre\lib\ext folder.
3. Again, if class is not found then application classloader searches all the Jar files and classes in CLASSPATH environment variable of system.
4. If class is found by any loader, then class is loaded by class loader; else **ClassNotFoundException** is thrown.
5. **Linking**

* After class is loaded by the classloader, linking is performed. A bytecode verifier will verify whether the generated bytecode is proper or not. If verification fails we will get a verification error.
* It also performs the memory allocation to static variables and methods found in the class.

1. **Initialization**

* This is the final phase of class loading, here all static variables will be assigned with the original values, and the static blocks will be executed.

#### JVM Memory Areas

**Method Area** stores class structures like metadata, the constant runtime pool, and the code for methods.

**Heap** stores all objects that are created during application execution.

Stacks store local variables, and intermediate results. All such variables are local to the thread by which they are created. Each thread has its own JVM stack, created simultaneously as the thread is created. So all such local variable are called thread-local variables.

**PC register** store the physical memory address of the statements which is currently executing. In Java, each thread has its separate PC register.

Java supports and uses **native code** as well. Many low level code is written in languages like C and C++. Native method stacks hold the instruction of native code.

#### JVM Execution Engine

* All code assigned to JVM is executed by an execution engine. The execution engine reads the byte code and executes one by one.
* It uses two inbuilt **interpreter** and **JIT compiler** to convert the bytecode to machine code and execute it.
* With JVM, both interpreter and compiler produce native code. The difference is in how they generate the native code, **how optimized it** is as well how costly the optimization is.

##### Interpreter

* A JVM interpreter pretty much converts each byte-code instruction to corresponding native instruction by looking up a predefined JVM-instruction to machine instruction mapping.
* It directly executes the bytecode and does not perform any optimization.

##### just-in-time (JIT) Compiler

* To improve performance, JIT compilers interact with the JVM at runtime and compile appropriate bytecode sequences into native machine code.
* Typically, the JIT compiler takes a block of code (not one statement at a time as interpreter), optimizes the code, and then translates it to optimized machine code.
* The JIT compiler is enabled by default.

## JRE

* The Java Runtime Environment (JRE) is a software package which bundles the libraries (jars) and the Java Virtual Machine, and other components to run applications written in the Java.
* JRE bundles the following components –

1. DLL files used by the Java HotSpot **Client Virtual Machine.**
2. DLL files used by the Java HotSpot **Server Virtual Machine.**
3. **Code libraries**, **property settings**, and **resource files** used by the Java runtime environment. e.g. rt.jar and charsets.jar.
4. Java **extension files** such as localedata.jar.
5. Contains files used for security management. These include the **security policy** (java.policy) and **security properties** (java.security) files.
6. Jar files containing support classes for **applets**.
7. Contains **TrueType font files** for use by the platform.

## JDK

**JDK is a superset of JRE**. JDK contains everything that JRE has along with development tools for developing, debugging, and monitoring Java applications.

## Difference between JDK, JRE and JVM

JRE = JVM + libraries to run Java application.

JDK = JRE + tools to develop Java Application.

# History of Java

Java was originally developed by **James Gosling** at Sun Microsystems (which has since been acquired by Oracle Corporation) and released in **1995** as a core component of Sun Microsystems’ Java platform.

# Features of Java

* **Object Oriented** – In Java, everything is represented as objects. An object is kind of wrapper that encapsulated data and its associated behavior.
* **Platform Independent** – The programs written in Java are converted to bytecode first, by Java compiler. This bytecode can be run in any machine having Java runtime environment (JRE). It makes the Java applications platform-independent.
* **Secure** – Java applications run in Java runtime environment (JRE) with almost no interaction with system OS. It makes Java more secure than other languages.
* **Multithreaded** – Java supports writing applications which can do multiple tasks in separate threads. All tasks progress using the [time slicing](https://en.wikipedia.org/wiki/Time_slice) technique of OS threads.
* **High-performance** – Java is an interpreted language, so it may never be as fast as a compiled language like C or C++. But Java enables high performance with the use of **just-in-time compiler**.
* **OS Architecture-neutral** – Java compiler generates an OS architecture-neutral class files or bytecode.

# Automatic Garbage Collection

* Java uses an automatic [garbage collector](https://howtodoinjava.com/java/garbage-collection/revisiting-memory-management-and-garbage-collection-mechanisms-in-java/) to manage memory in the object lifecycle.
* Java runtime is responsible for recovering the memory once objects are no longer in use. Once no references to an object remain, the unreachable memory becomes eligible to be freed automatically by the garbage collector.
* Something similar to a memory leak may still occur if a programmer’s code holds a reference to an object that is no longer needed
* Garbage collection may happen at any time. Ideally, it will occur when a program is idle. It is guaranteed to be triggered if there is insufficient free memory on the heap to allocate a new object; this can cause a program to stall momentarily. Explicit memory management is not possible in Java.

# Naming Conventions

1. Naming Packages

* Package names must be a group of words starting with all lowercase domain names (e.g. com, org, net, etc).
* **package** com.howtodoinjava.webapp.controller;

1. Naming Classes

* class names generally should be nouns, in title-case with the first letter of each separate word capitalized
* **public** **class** ArrayList {}

**public** **class** Employee {}

1. Naming Interfaces

* interfaces names, generally, should be adjectives. Interfaces should be in the title case with the first letter of each separate word capitalized

1. Naming Methods

* Methods always should be verbs.
* Words should be in camel case notation.
* **public** Report getReportById(Long id) {}

1. Naming Variables

* All instance, static and method parameter variable names should be in camel case notation. They should be short and enough to describe their purpose.
* Temporary variables can be a single character e.g. the counter in the loops.

1. Constant Naming Conventions

* Java constants should be all UPPERCASE where words are separated by underscore character (“\_”).
* Make sure to use the **final** modifier with constant variables.
* **public** **final** String SECURITY\_TOKEN = "...";

1. Naming Generic Types

* Generic type parameter names should be uppercase single letters.
* The letter 'T' for type is typically recommended. In JDK classes, E is used for collection elements, S is used for service loaders, and K and V are used for map keys and values.
* **public** **interface** Map <K,V> {}

**public** **interface** List<E> **extends** Collection<E> {}

1. Naming Enums

* enumeration names should be all uppercase letters.
* **enum** Direction {NORTH, EAST, SOUTH, WEST}

1. Naming Annotations

* Annotation names follow title case notation. They can be adjectives, verbs, or nouns based on the requirements.

# Classes & Objects

* objects are containers like data structure which have **state and behavior**. Objects represent the actors in the system or the application.
* An object is an instance of a class.
* The classes are the template that describes the state and behavior of its objects.

# Data Types

* The data type of the variable determines the range of the values that the memory location can hold. Therefore, the amount of memory allocated for a variable depends on its data type.
* Java supports two kinds of data types:

1. Primitive data type
2. Non-primitive or reference data type.

## Primitive Data Types

* A primitive data type directly holds a value in memory
* Primitive data types are not objects, as well as no references to the objects.
* The values stored in primitives are called **literals**.

A **literal** is the source code representation of a fixed value; literals are represented directly in your code without requiring computation.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | Description | Default Value | Memory Size |
| boolean | A binary value of either true or false | false | 1 bit |
| char | Any Unicode character | \u0000 (0) | 16-bit Unicode character |
| byte | Values from -128 to 127 | 0 | 8-bit signed value |
| short | Values from -32768 to 32767 | 0 | 16-bit signed value |
| int | Values from -231 to 231-1 | 0 | 32-bit signed value |
| long | Values from -263 to 263-1 | 0 | 64-bit signed value |
| float | [IEEE 754 floating point](https://en.wikipedia.org/wiki/IEEE_754) | 0.0 | 32-bit floating-point value |
| double | IEEE 754 floating point | 0.0 | 64-bit floating-point value |

* Float class defines three constants that represent positive infinity, negative infinity, and NaN of the float data type.

Float.POSITIVE\_INFINITY - Positive infinity of type **float**.

Float.NEGATIVE\_INFINITY - Negative infinity of type **float**.

Float.NaN - Not a Number of type **float**.

Float.MAX\_VALUE - The largest positive value that can be represented in a **float** variable.

Float.MIN\_VALUE - The smallest positive value greater than zero that can be represented in a **float** variable.

## Non-primitive Data Types

1. A non-primitive or reference data type holds the reference to an object in memory. Using the reference stored in the variable, you can access the fields and methods of the referenced object.

## Wrapper Classes

* A [wrapper class](https://howtodoinjava.com/java/basics/java-wrapper-classes/) is a class whose object **wraps or contains primitive data types**. In other words, we can wrap a primitive value into a wrapper class object.

|  |  |
| --- | --- |
| Primitive Type | Wrapper Class |
| double | Double |
| float | Float |
| long | Long |
| int | Integer |
| short | Short |
| byte | Byte |
| char | Character |
| boolean | Boolean |

### When to use Wrapper Classes

Java wrapper classes are used in scenarios –

* When two methods want to refer to the same instance of an primitive type, then pass wrapper class as **method arguments**.
* Java **Generics works only with object types** and does not support primitive types.
* Java **Collections deal only with objects**; to store a primitive type in one of these classes, you need to wrap the primitive type in a class.
* When you want to refer null from data type, you need object. **Primitives cannot have null** as value.

Best practices

1. Use primitives for variables which are **local in scope**. e.g., inside methods, counter for loops and intermediate results.
2. When data is transferred among method or classes, it’s better to use objects because only their references will be copied, no memory overhead will be added.
3. While sending data over network, use objects and make them [Serializable](https://howtodoinjava.com/java/serialization/java-serialization/). Wrapper classes are automatically Serializable.

# Variables

## Types of Variables

1. Instance variables
   * Non-static fields are also known as instance variables because their values are unique to each instance of a class. They are also called state variables.
2. Static variables
   * Also known as class variables. It is any field declared with the [static](https://howtodoinjava.com/java/basics/java-static-keyword/) modifier. It means that there is **exactly one copy of this variable in existence**, regardless of how many times the class has been instantiated.
   * A variable declared as “public static” can be treated as global variable in java.
3. Local variables

* These are used inside methods as temporary variables exist during the method execution.

1. Method arguments

* An argument is a variable that is passed to a method when the method is called.

## Variable Naming Conventions

# References

* <https://howtodoinjava.com/java/basics/java-tutorial/>
* <https://howtodoinjava.com/best-practices/solid-principles/>