Overview of Java with Basic Syntax

This covers basic java concepts and syntax + some portions of "Advanced Java Programming" –Unit: 1 (BSc. CSIT, TU)

What is Java?

- Java is a **programming language and a platform**.
- Java is a high level, robust, secured and object-oriented **programming language**.
- Any hardware or software environment in which a program runs, is known as a **platform**. Since Java has its own runtime environment (JRE) and API, it is called platform.

Java's Lineage

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades.

By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Indeed, for a brief moment it seemed as if programmers had finally found the perfect language. Because C++ blended the high efficiency and stylistic elements of C with the object-oriented paradigm, it was a language that could be used to create a wide range of programs. However, just as in the past, forces were brewing that would, once again, drive computer language evolution forward. Within a few years, the World Wide Web and the Internet would reach critical mass. This event would precipitate another revolution in programming.

The Creation of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995.

- ✓ The primary motivation for creation of Java was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.
- ✓ As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time consuming to create.

- ✓ An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform independent language that could be used to produce code that would run on a variety of CPUs under differing environments. **This effort ultimately led to the creation of Java.**
- ✓ About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics. However, with the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.
- ✓ Currently Java is owned by Oracle Corporation

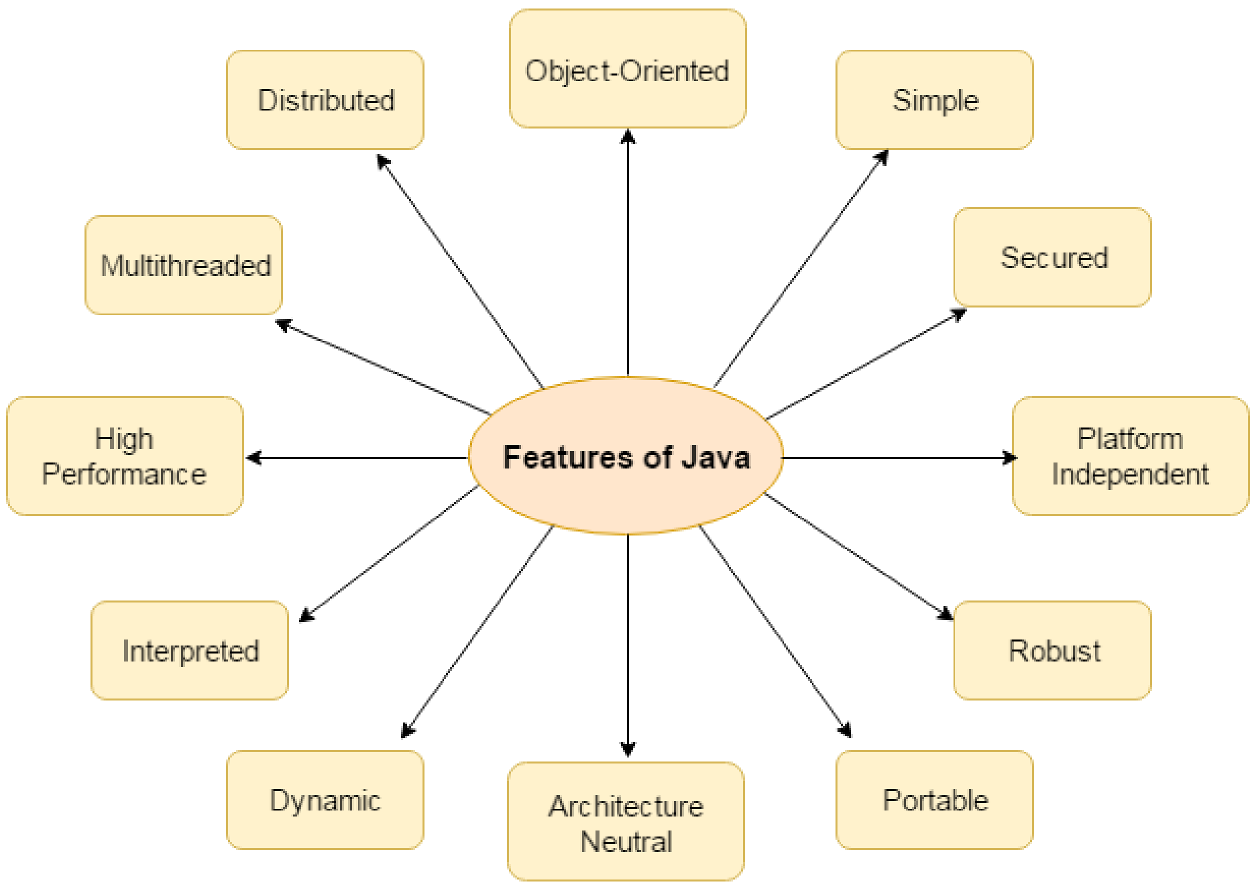
The Bytecode

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java runtime system, which is called the Java Virtual Machine (JVM).

When we write a program in Java, firstly, the compiler compiles that program and a bytecode is generated for that piece of code. When we wish to run this .class file on any other platform, we can do so. After the first compilation, the bytecode generated is now run by the Java Virtual Machine and not the processor in consideration. This essentially means that we only need to have basic java installation on any platforms that we want to run our code on. Resources required to run the bytecode are made available by theJava Virtual Machine, which calls the processor to allocate the required resources.

Bytecode is essentially the machine level language which runs on the Java Virtual Machine. Whenever a class is loaded, it gets a stream of bytecode per method of the class. Whenever that method is called during the execution of a program, the bytecode for that method gets invoked. Javac not only compiles the program but also generates the bytecode for the program. Thus, we have realized that the bytecode implementation makes Java a platform-independent language. This helps to add portability to Java which is lacking in languages like C or C++. Portability ensures that Java can be implemented on a wide array of platforms like desktops, mobile devices, severs and many more. Supporting this, Sun Microsystems captioned JAVA as "write once, read anywhere" or "WORA" in resonance to the bytecode interpretation.

Features of Java (The Java Buzzwords)



Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high performance non objects.

Java is an object oriented language. It supports various object oriented features like class, object, encapsulation, polymorphism, inheritance, abstraction, etc.

Robust

Robust simply means strong. The ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time.

- → It provides many features that make the program execute reliably in variety of environments.
- → Java is a strictly typed language. It checks code both at compile time and runtime.
- → Java takes care of all memory management problems with garbage-collection.
- → Java, with the help of exception handling captures all types of serious errors and eliminates any risk of crashing the system.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.

A thread is like a sub program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. At the time of Java's creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

- ✓ Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand.
- ✓ Java supports dynamic compilation and automatic memory management (garbage collection).
- ✓ Java is capable of linking in new class libraries, methods, and objects.
- ✓ It can also link native methods (the functions written in other languages such as C and C++).

Compiled and Interpreted

Usually a computer language is either compiled or Interpreted. Java combines both this approach and makes it a two-stage system.

Compiled : Java enables creation of a cross platform programs by compiling into an intermediate representation called Java Bytecode.

Interpreted : Bytecode is then interpreted, which generates machine code that can be directly executed by the machine that provides a Java Virtual machine.

High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

- ✓ Java performance is high because of the use of bytecode.
- ✓ The bytecode was used, so that it would be easily translated into native machine code.

Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

Secure

- ✓ Java provides a "firewall" between a networked application and your computer.
- ✓ When a Java Compatible Web browser is used, downloading can be done safely without fear of viral infection or malicious intent.
- ✓ Java achieves this protection by confining a Java program to the java execution environment and not allowing it to access other parts of the computer.

Java Architecture

- In Java, there is the process of compilation and interpretation.
- Whatever code is written in Java, is converted into byte codes by the Java Compiler.
- These byte codes, in turn are converted into machine code by the Java Virtual Machine.
- The Machine code is executed directly by the machine in which the Java program runs.
- Hence, there is the process of compilation and interpretation.

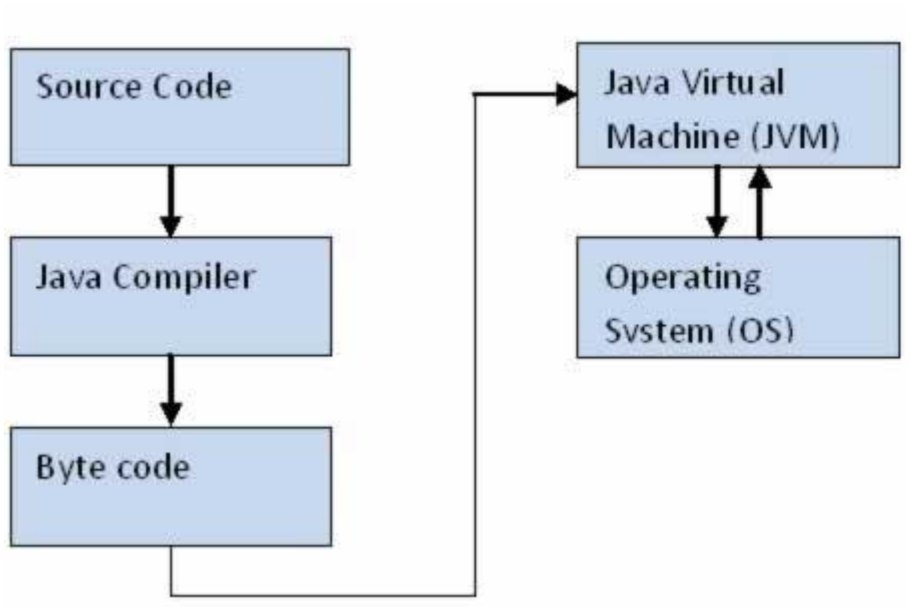


Figure 1: Java Architecture

- In addition to converting the compiled code into machine language code, the JVM provides an environment for running Java Programs.

How is Java Platform Independent?

- One of the main advantages of Java programs is that it is Platform independent.
- That is, the same java program can be run on different platforms without having to make any changes in the source code.
- You just have to make Java program in any one platform and the same program can be executed in any platform. Have you wondered how?

- The Java compiler compiles the source code and converts it into bytecode. This bytecode is stored in class files.
- Each platform has its own unique JVM.
- The process of interpreting the bytecode by the JVM and converting it into machine code is same for all JVMs regardless of the platforms.
- This makes Java Platform independent. Let us understand this with a diagram.
- The same bytecode is interpreted by two different JVMs of Windows and Linux and converted into machine code.

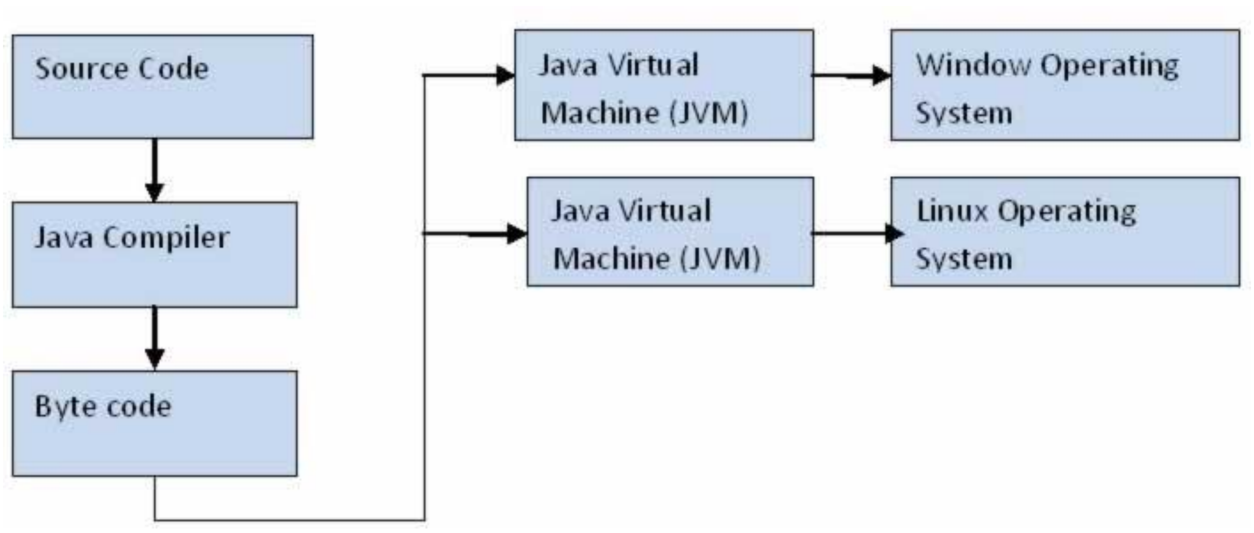


Figure 2: Java is platform independent

What is JRE?

- JRE stands for Java Runtime Environment and this is where the JVM resides along with all the class libraries and other supporting components.
- As explained earlier, the source code is converted into bytecode and this bytecode is stored in class files. When the program is run, the class file is loaded , verified and the JVM interprets the bytecode into machine code which will be executed by the machine in which the program runs.
- With respect to the above explanation, the JRE performs the following tasks.
- The JRE loads the class file(done by the class loader)
- The JRE verifies the bytecode.(done by the bytecode verifier)
- The JRE interprets the bytecode.(done by the JVM)

This is explained in the diagram below. A detailed Java Architecture can drawn as:

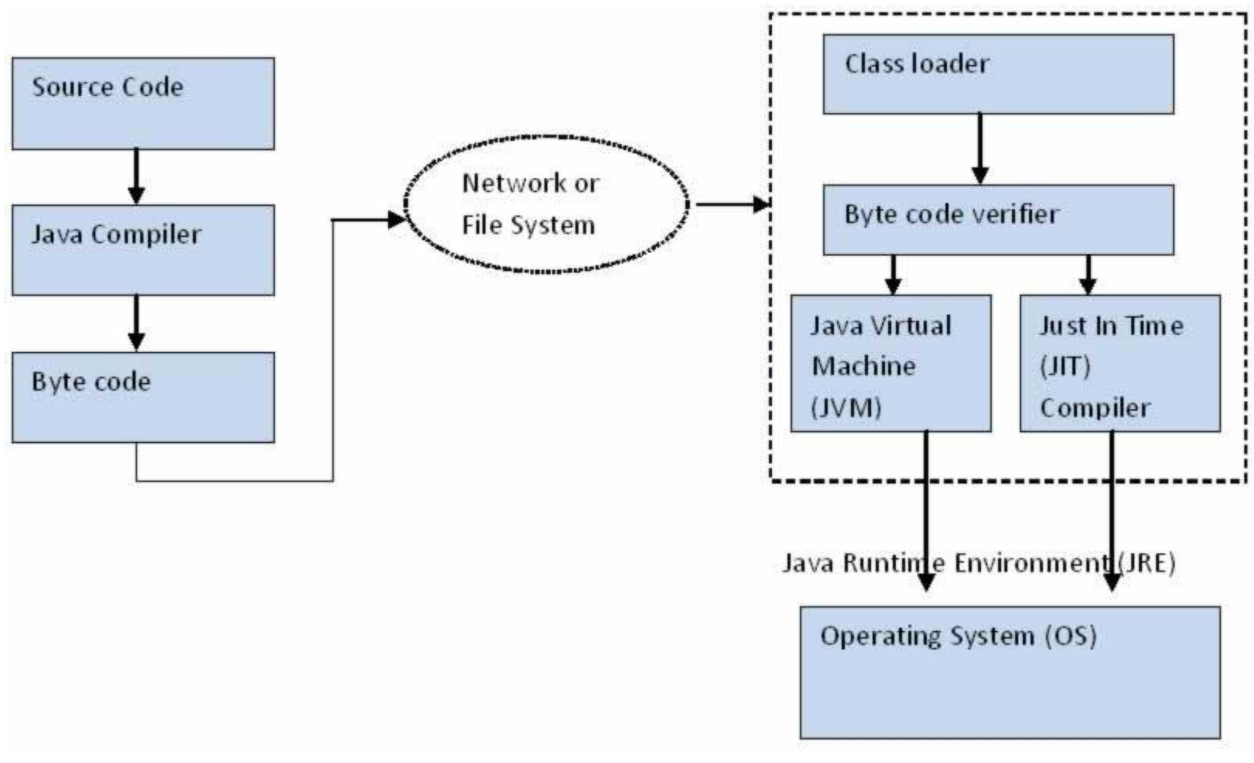


Figure 3: Java Architecture in detail

What are the components of JRE?

Class Loader

- All the class files required to run a Java program is loaded by the class loader.
- The Java program is made safe and secure by the Class loader as it separates the namespace of the classes obtained from the network and the namespace of the classes obtained locally.
- Once the byte code is loaded, the next step is to verify the bytecode which is done by the bytecode verifier.

Byte code Verifier.

- The bytecode verifier verifies the byte code to check if there are any security problems.
- The following things are checked by the Bytecode verifier
  - ✓ .Does the code follow JVM specifications.
  - ✓ Is there any unauthorized access to memory made by the code?
  - ✓ Does the code cause any stack overflows?
  - ✓ Are there any invalid or illegal data conversions (egs converting float into object type)?

- Once this verification is successfully completed, the JVM converts this byte code into machine code which will be executed directly by the machine in which the Java Program runs.

Whats the role of Just in Time Compiler(JIT)?

- For those of you who are wondering what does JIT compiler do, well the JIT compiler helps in the faster execution of the Java Program. How does that happen?
- As we discussed earlier the bytecode is interpreted by the JVM and converted into machine code for the program to execute. This interpretation is a slow process.
- To overcome this problem, the JRE includes a component called Just in Time(JIT) compiler. JIT makes the interpretation and execution faster.

Why the name “Just in Time”?

- If there is a JIT compiler library in the JRE, the JIT compiler compiles the bytecode into native machine code , when a particular bytecode is executed for the first time.
- This native machine code can be directly executed by the machine.
- Once the native machine code is recompiled by the JIT compiler, the execution time will be much lesser. This compilation happens when the bytecode is about to be executed and hence the name “Just in Time”.
- When the bytecode is compiled into a particular machine code, it is also cached by the JIT compiler and so, it can be reused for any future needs.
- So the performance improvement can be seen when the same code is executed again and again as the JIT uses the cached machine code.

JDK, JRE, JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance.

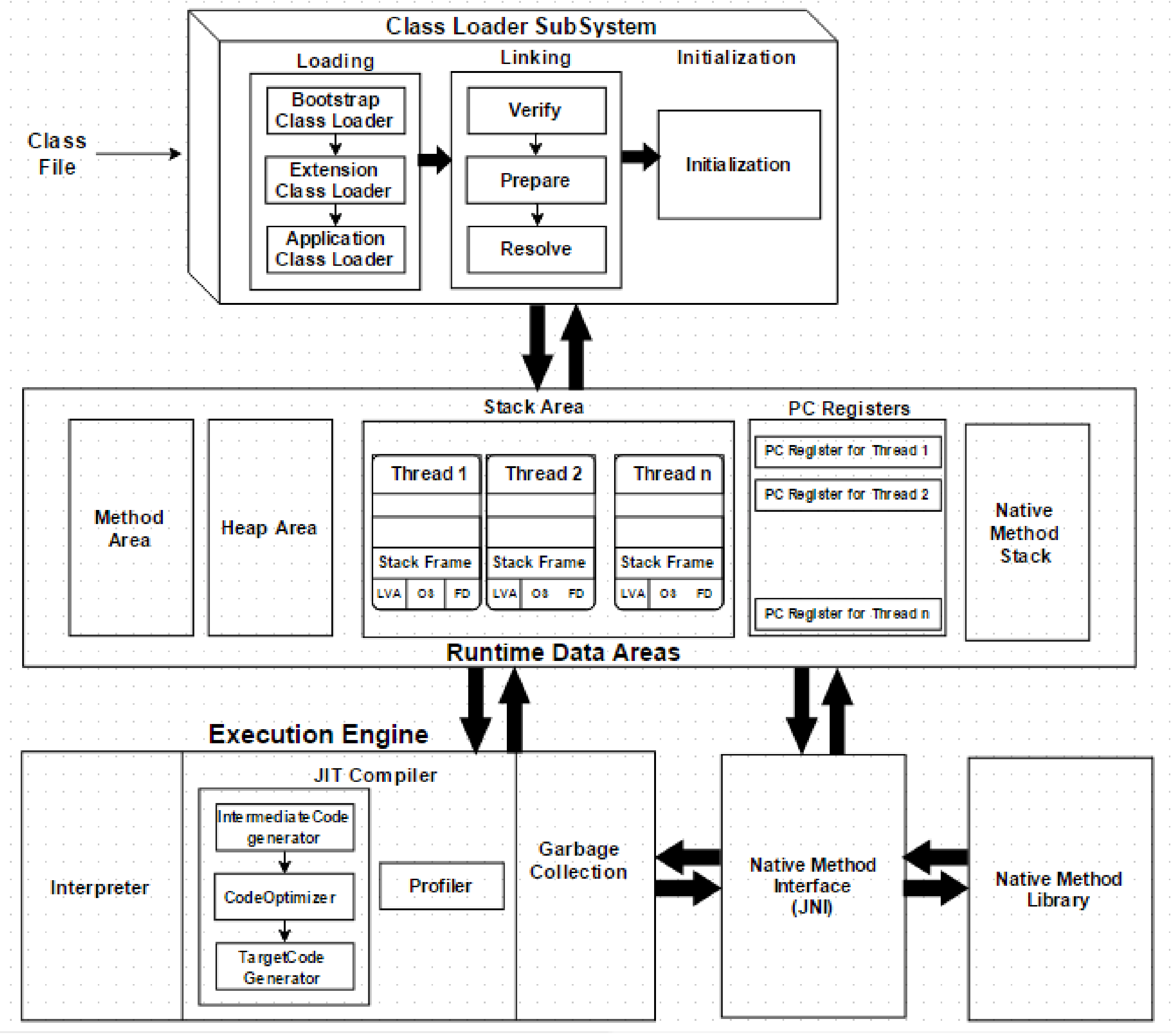


Figure 4: JVM Architecture

The JVM performs following main tasks:

- ✓ Loads code
- ✓ Verifies code
- ✓ Executes code
- ✓ Provides runtime environment

JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.

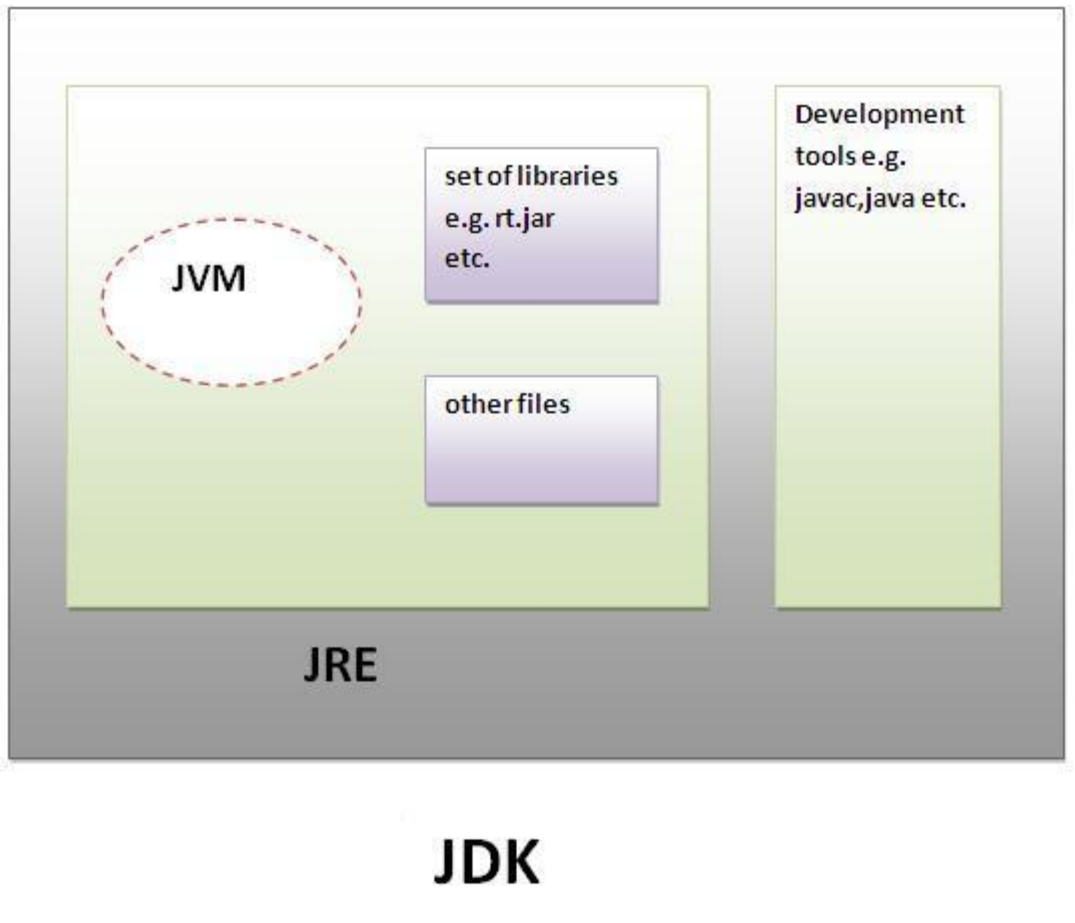

JVM
set of libraries e.g. rt.jar etc.
other files
Development tools e.g. javac,java etc.
JRE
JDK

Figure 5: JDK, JRE and JVM

Java program compilation

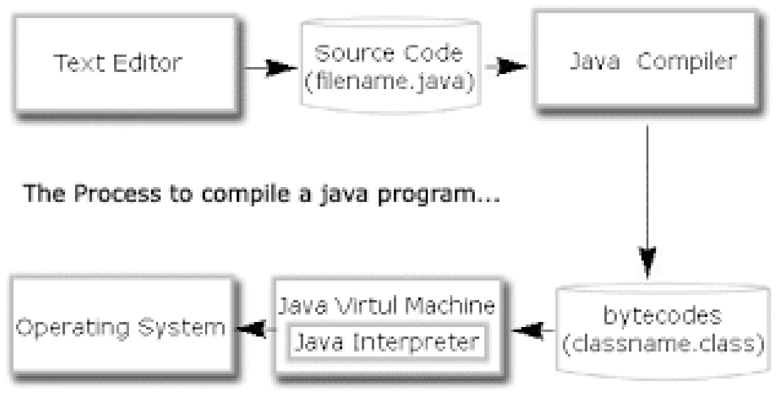

Text Editor
Source Code (filename.java)
Java Compiler
The Process to compile a java program...
bytecodes (classname.class)
Java Virtul Machine
Java Interpreter
Operating System

Figure 6: Process to compile a java program

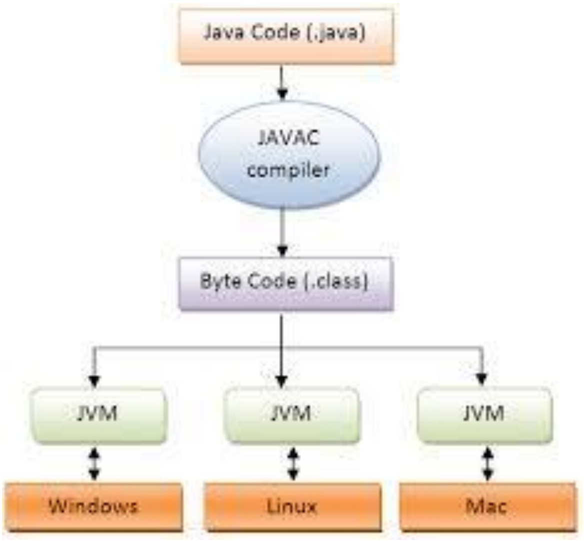


Figure 7: Another diagram to show Java is platform independent (Architecture Neutral) language

What are Java SE, Java EE and Java ME?

- **Java SE:** It stands Java Standard Edition which means all concepts related to core java such as I/O, Collections, Threading, Concurrency, etc.
- **Java EE**: It stands Java Enterprise Edition which means all concepts related to advance java such as servlets, web-services, jsps and all things related to web based software applications. This is mainly used by enterprises for business solutions based on web or network.
- **Java ME**: It means Java Micro Edition which means all concepts related to handheld or appliances based applications. This has been used for mobile applications, home appliances applications, etc.

Java Path

- The path is required to be set for using tools such as javac, java etc.
- If you are saving the java source file inside the jdk/bin directory, path is not required to be set because all the tools will be available in the current directory.
- But If you are having your java file outside the jdk/bin folder, it is necessary to set path of JDK.
- There are 2 ways to set java path:
    - temporary
    - permanent
- To set the temporary path of JDK in Windows, you need to follow following steps:
  - Open command prompt
  - Copy the path of jdk/bin directory
  - Write in command prompt: set path=copied_path

- For setting the permanent path of JDK in Windows, you need to follow these steps:
    - Go to MyComputer (This PC) properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

What is CLASSPATH?

- CLASSPATH is actually an environment variable in Java, and tells Java applications and the Java Virtual Machine (JVM) where to find the libraries of classes. These include any that you have developed on your own.
- An environment variable is a global system variable, accessible by the computer's operating system (e.g., Windows). Other variables include COMPUTERNAME, USERNAME (computer's name and user name).
- In Java, CLASSPATH holds the list of Java class file directories, and the JAR file, which is Java's delivered class library file.

Lexical Issues

Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords. These are the atomic elements of a java program.

Whitespace

→ Java is a free-form language. This means that you do not need to follow any special indentation rules

→ In Java, whitespace is a space, tab, or newline

Identifiers

- → Identifiers are used to name things, such as classes, variables, and methods.
- → An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.)
- → They must not begin with a number, lest they be confused with a numeric literal.
- → Java is case-sensitive, so **NAME** is a different identifier than **Name**.
- → Reserved words cannot be used as identifiers
- → There is no length limit for identifiers but it is not recommended to use more than 15 character long.
- → Examples of valid identifiers are : **EmpName, salary, t3, $test, it_is_ok**
- → Invalid identifiers: **student-Name, 2increase, Not/Ok, class, final**

Literals

- → A constant value which can be assigned to a variable is called a literal
- → A constant value in Java is created by using a literal representation of it. For example, here are some literals:

100	98.6	'X'	"This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third

Is a character constant, and the last is a string. (integer literal, floating point literal, character literal and string literal)

- → A literal can be used anywhere a value of its type is allowed.

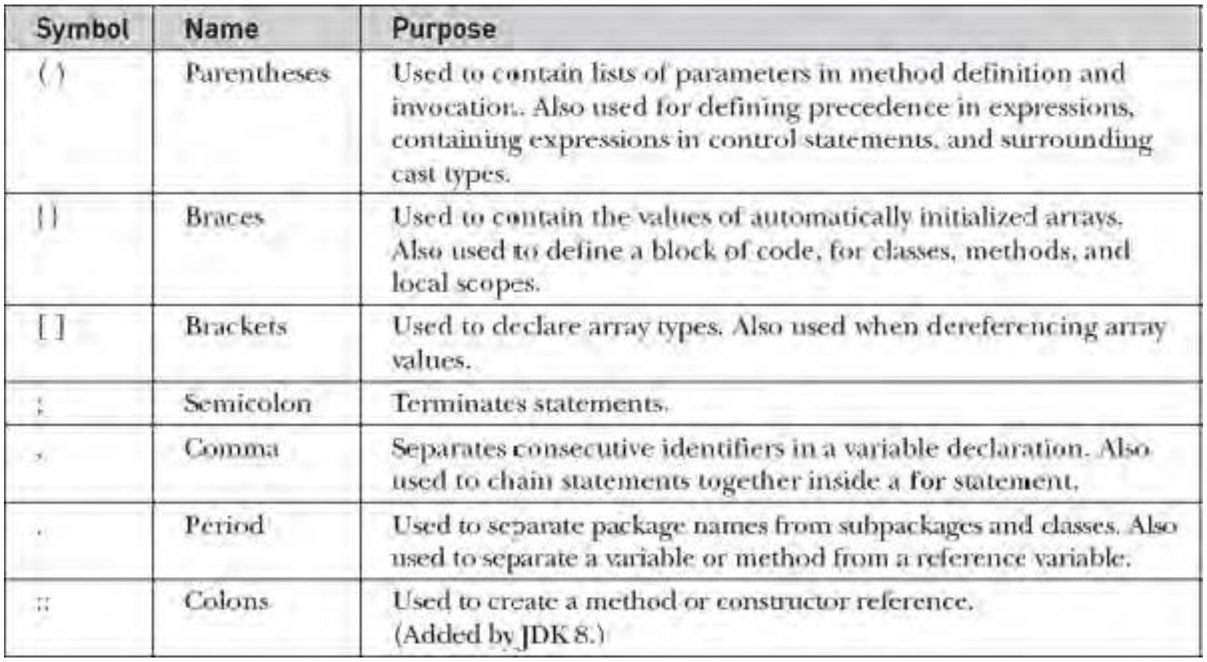
Comments

There are three types of comments defined by Java.

- → Single line comment.
```
// it is single line comment
```
- → Multi line comment
```
/* it is a comment
this is in next line */
```
- → Documentation comment: This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a `/** and ends with a */.`

Separators

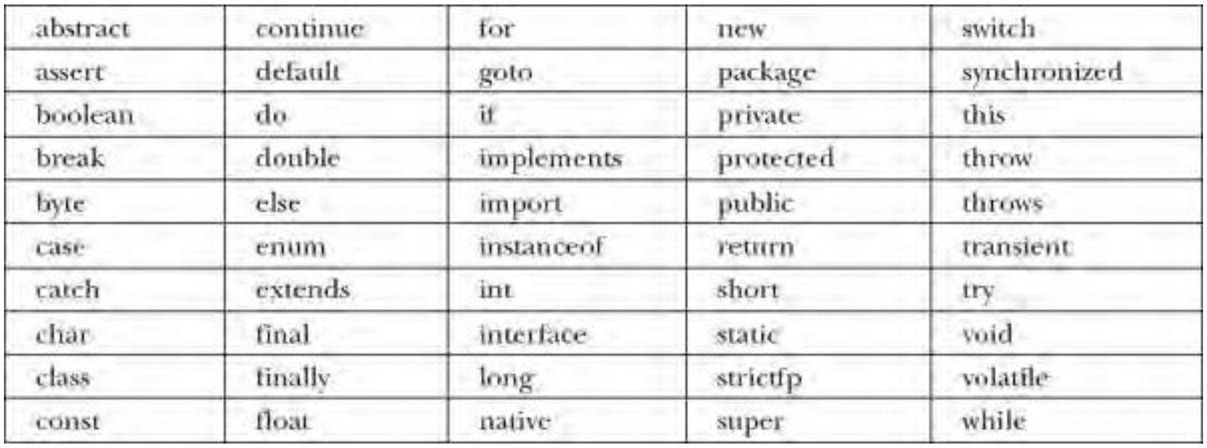
In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon which is used to terminate statements. The separators are shown in the following table:



Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference. (Added by JDK 8.)

The Java Keywords

There are 50 keywords currently defined in the Java language. These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. These keywords cannot be used as identifiers. Thus, they cannot be used as names for a variable, class, or method.



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

→ The keywords **const** and **goto** are reserved but not used.
→ In addition to the keywords, Java reserves the following: **true, false, and null**. These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

Java Modifiers

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers:

- **Access Modifiers:** default, public , protected, private
- **Non-access Modifiers:** final, abstract, strictfp

Java Access modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity -** Java is case sensitive, which means identifier *Hello* and *hello* would have different meaning in Java.
- **Class Names -** For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case. Example: class *MyFirstJavaClass*
- **Method Names -** All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case. Example: public void *myMethodName()*
- **Program File Name -** Name of the program file should exactly match the class name. When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile). Example: Assume *'MyFirstJavaProgram'* is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*
- **public static void main(String args[ ]) -** Java program processing starts from the main() method which is a mandatory part of every Java program.

Java Naming conventions

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc.

But, it is not forced to follow. So, it is known as convention not rule.

All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

CamelCase in java naming conventions

Java follows camelcase syntax for naming the class, interface, method and variable.

If name is combined with two words, second word will start with uppercase letter always e.g. **actionPerformed(), firstName, ActionEvent, ActionListener** etc.

Data Types in Java

A variable is a container which holds the value while the java program is executed. A variable is assigned with a data type.

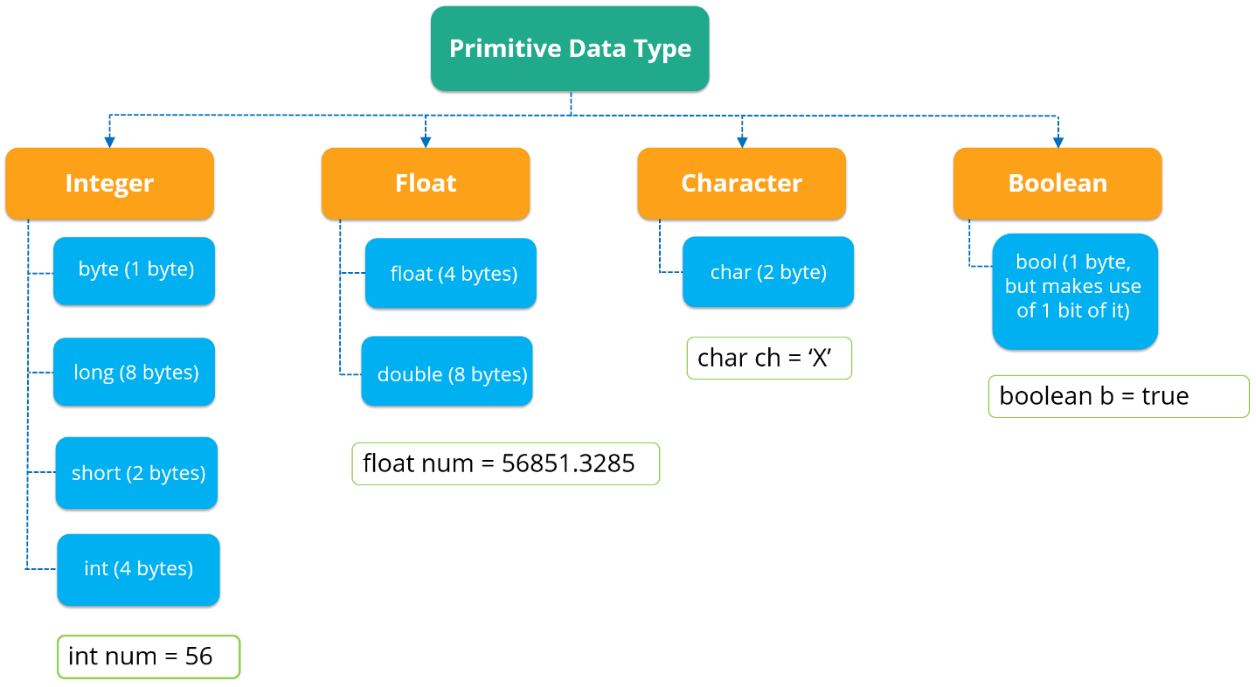


Figure 8: Data types in Java

As you can see in the above figure, data types are of 4 major types.

- The first data type is an Integer which stores a numerical value.
- Now, if a numerical value contains decimal part, it will be referred as float.
- Next, if you wish to store a character, then the third data type i.e char is used. In char, you can store any alphabetical character as well as a special character.
- The last data type is Boolean which stores only 'true' or 'false' value.

Variable is a name of memory location. There are three types of variables in java: **local, instance and static.**

There are two types of data types in java: **primitive** and **non-primitive**

➢ Variable is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed.

```
int data=20;     //Here data is variable
```

Java Is a Strongly Typed Language

- It is important to state at the outset that Java is a strongly typed language.
- Indeed, part of Java's safety and robustness comes from this fact.
- First, every variable has a type, every expression has a type, and every type is strictly defined.
- Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

Every variable has a type, every expression has a type and all types are strictly define more over every assignment should be checked by the compiler by the type compatibility hence java language is considered as strongly typed programming language.

Java is pure object oriented programming or not?

→ Java is not considered as pure object oriented programming language because several oops features (like multiple inheritance, operator overloading) are not supported by java moreover we are depending on primitive data types which are nonobjects.

The Primitive Types

→ Java defines eight primitive types of data: **byte, short, int, long, char, float, double,** and **boolean**.
→ The primitive types are also commonly referred to as simple types.
→ These can be put in four groups:

1. **Integers :** This group includes byte, short, int, and long, which are for whole-valued signed numbers.
2. **Floating-point numbers:** This group includes float and double, which represent numbers with fractional precision.
3. **Characters**: This group includes char, which represents symbols in a character set, like letters and numbers.
4. **Boolean:** This group includes boolean, which is a special type for representing true/false values.

→ Except Boolean and char all remaining data types are considered as signed data types

because we can represent both "+ve" and"-ve" numbers.

Integral Data Types

Java defines four integer types: byte, short, int, and long. All of these are signed, positive

and negative values.

byte

→ The smallest integer type is byte.
→ This is a signed 8-bit type that has a range from –128 to 127.
→ Byte variables are declared by use of the **byte** keyword.
→ Variables of type byte are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.

short

→ short is a signed 16-bit type. It has a range from –32,768 to 32,767.
→ It is probably the least used Java type.
→ Keyword **short** is used for declaration.

int

→ The most commonly used integer type is int.
→ It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
→ In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays

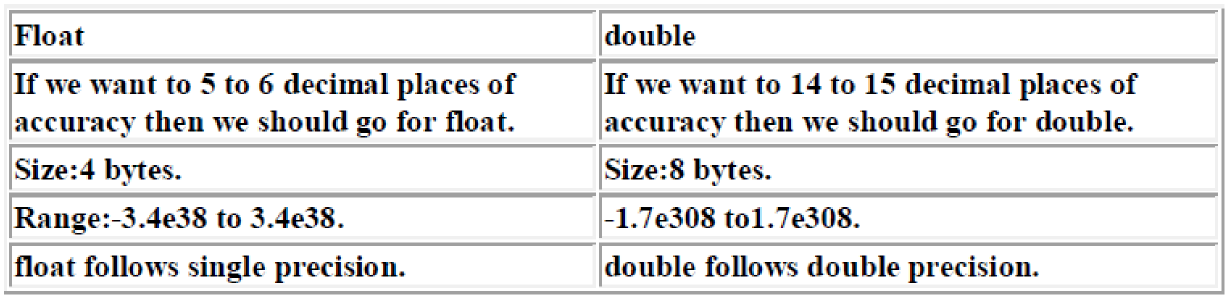
long

→ long is a signed 64-bit type and has a range from -2^{63} to 2^{63}-1 bits.
→ It is useful for those occasions where an int type is not large enough to hold the desired value.

Floating-Point Types

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type.

→ There are two kinds of floating-point types, float and double
→ The type **float** specifies a single-precision value that uses 32 bits of storage
→ The type **double** specifies a double-precision value that uses 64 bits of storage



Float	double
If we want to 5 to 6 decimal places of accuracy then we should go for float.	If we want to 14 to 15 decimal places of accuracy then we should go for double.
Size:4 bytes.	Size:8 bytes.
Range:-3.4e38 to 3.4e38.	-1.7e308 to1.7e308.
float follows single precision.	double follows double precision.

Characters

→ In Java, the data type used to store characters is **char**.
→ C & C++ are ASCII code based the no. of ASCII code characters. So Size is of **char** is 8 bits (Since ASCII codes can be represented using less than 256 (i.e. 2^8))
→ This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.
→ Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars.
→ The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.
→ Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters.
→ Although char is designed to hold Unicode characters, it can also be used as an integer type on which you can perform arithmetic operations. For example, you can add two characters together, or increment the value of a character variable.

Booleans

→ Java has a primitive type, called **boolean**, for logical values.
→ It can have only one of two possible values, **true** or **false**.
→ This is the type returned by all relational operators, as in the case of a < b.
→ **boolean** is also the type required by the conditional expressions that govern the control statements such as if and for.

Variables

- The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.
- Java Variables or variables in any other programming language are containers, which hold some value. Because Java is a strongly typed language, so every variable must be declared and initialized or assigned before it is used. A variable, in the simplest way, is declared by placing a valid type followed by the variable name

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```
type identifier [= value][, identifier [= value] …];
```

Here, **type** is one of Java's atomic types, or the name of a class or interface. The **identifier** is the name of the variable. We can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list.

Examples:

```
int a, b, c;            // declares three ints, a, b, and c.
int d = 3, e, f = 5;    // declares three more ints, initializing
                        // d and f.
byte z = 22;            // initializes z.
double pi = 3.14159;    // declares an approximation of pi.
char x = 'x';           // the variable x has the value 'x'.
```

Initialization is the process of providing value to a variable at declaration time. A variable is initialized once in its life time. Any attempt of setting a variable's value after its declaration is called assignment.

Dynamic Initialization

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

- Initializing a variable at run time is called dynamic initialization.

```
/* DynamicInitializationDemo.java */
public class DynamicInitializationDemo
{
    public static void main(String[] args)
    {
         double a = 3.0, b= 4.0;
        //c will be initialized when Math.sqrt
        //will be executed at run time
        double c = Math.sqrt (a*a+b*b);
        System.out.println("The value of c  is : " + c);
    }
}
```

Here, three local variables—**a, b,** and **c**—are declared. The first two, **a** and **b**, are initialized by constants. However, c is initialized dynamically. The program uses another of Java's built-in methods, **sqrt( ),** which is a member of the **Math** class, to compute the square root of its argument.

The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Indeed, the scope rules provide the foundation for encapsulation. Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that **objects declared in the outer scope will be visible to code within the inner scope**. However, the reverse is not true. **Objects declared within the inner scope will not be visible outside it**.

```
// Demonstrate block scope.
class Scope {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope
      int y = 20; // known only to this block

      // x and y both known here.
      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it. For example, this fragment is invalid because count cannot be used prior to its declaration

```
// This fragment is wrong!
count = 100; // oops! cannot use count before it is declared!
int count;
```

Notes:

- Variables are created when their scope is entered, and destroyed when their scope is left.
- The scope of a variable defines the section of the code in which the variable is visible. As a general rule, variables that are defined within a block are not accessible outside that block. The lifetime of a variable refers to how long the variable exists before it is destroyed

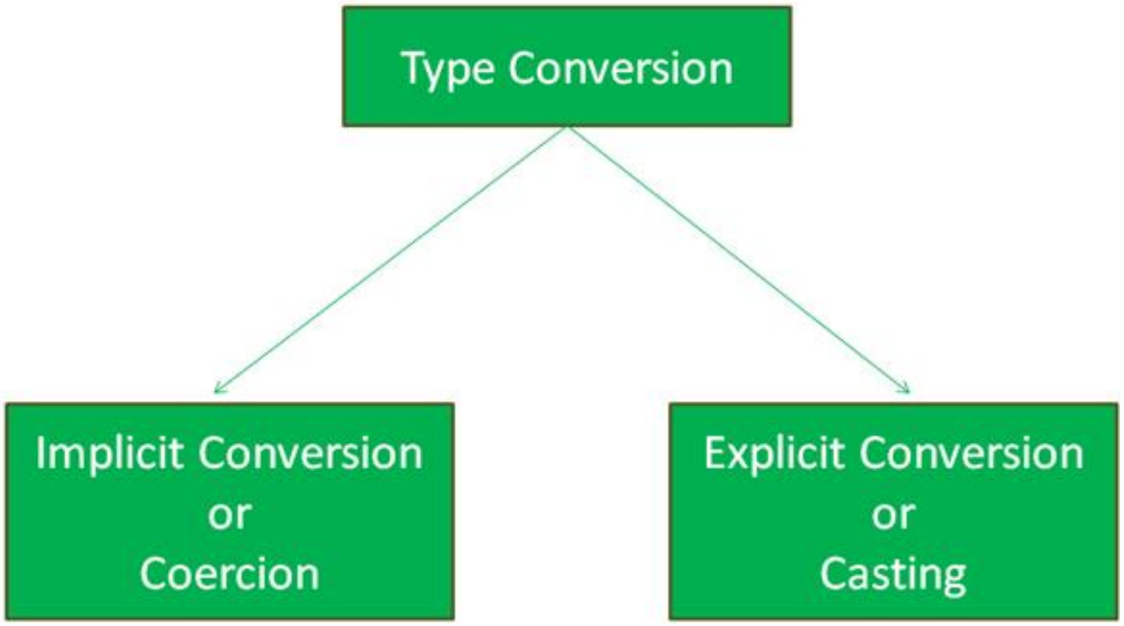
Java Keywords

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

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

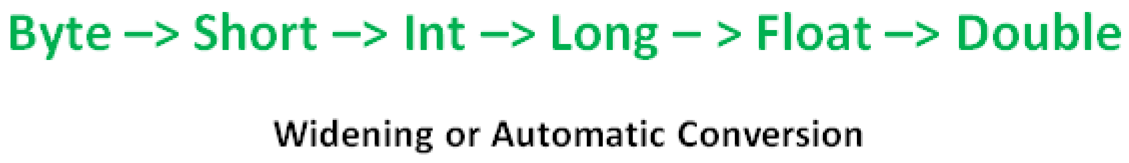
Type Conversion and Casting

- When we assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly.
- It is fairly common to assign a value of one type to a variable of another type.
- If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an **int** value to a **long** variable.
- However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte.** Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, we must use a **cast**, which performs an explicit conversion between incompatible types


Type Conversion
Implicit Conversion or Coercion
Explicit Conversion or Casting

Java’s Automatic Conversions

- Also known as widening/ implicit conversion / coercion
- When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
    - The two types are compatible.
    - The destination type is larger than the source type
- When these two conditions are met, a widening conversion takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.
- For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.
- However, there are no automatic conversions from the numeric types to **char** or **boolean.** Also, **char** and **boolean** are not compatible with each other.
- As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte, short, long,** or **char.**


Byte –> Short –> Int –> Long – > Float –> Double

Widening or Automatic Conversion

```
//Example: Automatic type conversion
            class Test
            {
            public static void main(String[] args)
            {
            int i = 100;

            //automatic type conversion
            long l = i;

            //automatic type conversion
            float f = l;
            System.out.println("Int value "+i);
            System.out.println("Long value "+l);
            System.out.println("Float value "+f);
            }
            }
```

Output:

```
Int value 100

Long value 100

Float value 100.0
```

Casting Incompatible Types

- Also known as narrowing/ Type casting /Explicit Conversion
- There may be situations where we want to convert a value having a type of size less than the destination type size.(For example, what if we want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**.)
- In such cases Automatic Conversion will not help us. We have do it on our own explicitly. That is why this type of conversion is known as explicit conversion or casting as the programmer does this manually.
- This is useful for incompatible data types where automatic conversion cannot be done.
- To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:
  `(target-type) value`
  Here, **`target-type`** specifies the desired type to convert the specified value to.
- An example for type casting is shown below:
  ```
        int a = 10;
        byte b = (byte) a;
  ```
- In the above example, we are forcing an integer value to be converted into a byte type. For type casting to be carried out both the source and destination types must be compatible with each other. For example, we can’t convert an **integer** to **boolean** even if we force it.

- In the above example, size of source type int is 32 bits and size of destination type byte is 8 bits. Since we are converting a source type having larger size into a destination type having less size, such conversion is known as **narrowing conversion.**
- A type cast can have unexpected behavior. For example, if a **double** is converted into an **int**, the fraction component will be lost.(it is called **truncation**)

Double –> Float –> Long –> Int –> Short –> Byte

Narrowing or Explicit Conversion

- While assigning value to byte type the fractional part is lost and is reduced to modulo 256 (range of byte). Example:

```
//Java program to illustrate Conversion of int and double to byte
class Test
{
    public static void main(String args[])
    {
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("Conversion of int to byte.");

        //i%256
        b = (byte) i;
        System.out.println("i = " + i + " b = " + b);
        System.out.println("\nConversion of double to byte.");

        //d%256
        b = (byte) d;
        System.out.println("d = " + d + " b= " + b);
    }
}
```

OUTPUT:

```
Conversion of int to byte.

i = 257 b = 1

Conversion of double to byte.

d = 323.142 b = 67
```

Automatic Type promotion Rules

- In addition to assignments, there is another place where certain type conversions may occur: in expressions
- Java defines several type promotion rules that apply to expressions
    - All **char, short** and **byte** values are automatically promoted to **int** type.
    - If at least one operand in an expression is a **long** type, then the entire expression will be promoted to **long**.
    - If at least one operand in an expression is a **float type**, then the entire expression will be promoted to **float**.
    - If at least one operand in an expression is a **double** type, then the entire expression will be promoted to **double**.

```
//Java program to illustrate Type promotion in Expressions
class Test
{
    public static void main(String args[])
    {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;

        // The Expression
        double result = (f * b) + (i / c) - (d * s);

        //Result after all the promotions are done
        System.out.println("result = " + result);
    }
}
```

Output:

```
result = 626.7784146484375
```

In the first subexpression, **f * b**, **b** is promoted to a **float** and the result of the subexpression is **float**. Next, in the subexpression **i/c**, **c** is promoted to **int**, and the result is of type **int**. Then, in **d * s**, the value of **s** is promoted to **double**, and the type of the subexpression is **doubl**e. Finally, these three intermediate values**, float, int,** and **double**, are considered. The outcome of **float** plus an **int** is a **float**. Then the resultant **float** minus the last **double** is promoted to double, which is the type for the final result of the expression.

Comments in Java

Java supports single-line and multi-line comments very similar to C and C++. All characters available inside any comment are ignored by Java compiler.

```
public class MyFirstJavaProgram{

/* This is my first java program.

* This will print 'Hello World' as the
output

* This is an example of multi-line
comments.

*/

public static void main(String []args){

// This is an example of single line
comment

/* This is also an example of single line
comment. */

System.out.println("Hello World");

}

}
```

Using Blank Lines

A line containing only white space, possibly with a comment, is known as a blank line, and Java totally ignores it.

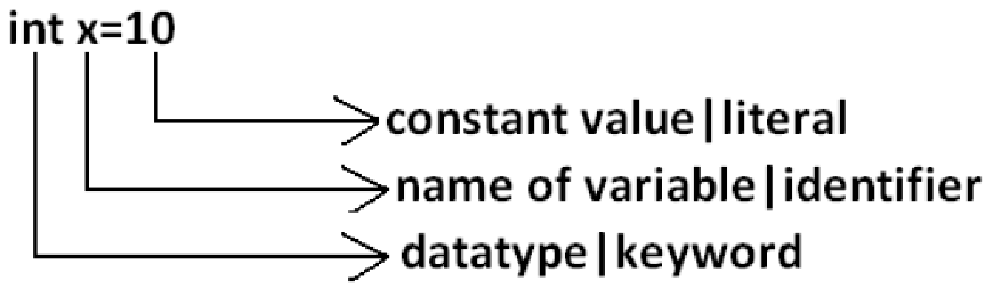
Java Operators

Operator Type	Category	Precedence
Unary	postfix	expr++ expr--
	prefix	++ expr --expr +expr -expr ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&

	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Literals

- → Any constant value which can be assigned to the variable is called literal.
- → Example:



- → **Types:** Integer Literals, Floating-Point Literals, Character Literals, String Literals

Integral Literals:

For the integral data types (byte, short, int and long) we can specify literal value in the following ways.

- → ***Decimal literals*****:** Allowed digits are 0 to 9.

Example: int x=10;

→ ***Binary literals :*** Allowed digits are 0 and 1. Literal value should be prefixed with **0b** or **0B**. For example, this specifies the decimal value 10 using a binary literal

Example: int x = 0b1010;

→ ***Octal literals*:** Allowed digits are 0 to 7. Literal value should be prefixed with zero.

Example: int x=010;

→ ***Hexa Decimal literals:***

- ✓ The allowed digits are 0 to 9, **A** to **F** (**a** to **f**)
- ✓ For the extra digits we can use both upper case and lower case characters.
- ✓ This is one of very few areas where java is not case sensitive.
- ✓ Literal value should be prefixed with 0x(or) 0X.

Example: int x=0x10;

Which of the following are valid declarations?

```
int x=0777; //(valid)

int x=0786; //(invalid)

int x=0xFACE; (valid)

int x=0xbeef; (valid)

int x=0xBeer; //(invalid)

int x=0xabb2cd;(valid)
```

→ By default every integral literal is int type but we can specify explicitly as long type by suffixing with small "l" (or) capital "L".

→ We can embed one or more underscores in an integer literal. Doing so makes it easier to read large integer literals. When the literal is compiled, the underscores are discarded. For example, given `int x = 123_456_789;` the value given to x will be 123,456,789. The underscores will be ignored. Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits.

Floating-Point Literals

→ Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation.

→ Standard notation consists of a whole number component followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers.

- → Scientific notation uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an E or e followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05, and 2e+100.
- → Floating point literal is by default double type but we can specify explicitly as float type by suffixing with f or F.
- → We can specify explicitly floating point literal as double type by suffixing with d or D.
- → Underscores are allowed in between digits as in integer literals.

```
double x = 3_123_456.4_5_7 //valid
```

Boolean Literals

Boolean literals are simple. There are only two logical values that a boolean value can have, **true** and **false**. The values of true and false do not convert into any numerical representation. The true literal in Java does not equal 1, nor does the false literal equal 0.

```
Example:

boolean b=true;(valid)

boolean b=0;//(invalid)

boolean b=True;//(invalid)

boolean b="true";//(invalid)
```

Character Literals

- A char literal can be represented as single character within single quotes.

Example:

```
char ch='a';//(valid)

char ch=a;//(invalid)

char ch="a";//(invalid)

char ch='ab';//(invalid)
```

- We can specify a char literal as integral literal which represents Unicode of that character. We can specify that integral literal either in decimal or octal or hexadecimal form but allowed values range is 0 to 65535.

Example: `char ch=97; //(valid)`

- We can represent a char literal by Unicode representation which is nothing but '\uxxxx' (4 digit hexa-decimal number)

Example : `char ch='\ubeef'`**Escape Sequences**

A character preceded by a backslash (\) is an escape sequence and has a special meaning to the compiler.

The newline character (\n) has been used frequently in this tutorial in System.out.println() statements to advance to the next line after the string is printed.

Following table shows the Java escape sequences −

Escape Sequence	Description
\t	Inserts a tab in the text at this point.
\b	Inserts a backspace in the text at this point.
\n	Inserts a newline in the text at this point.
\r	Inserts a carriage return in the text at this point.
\f	Inserts a form feed in the text at this point.

\'	Inserts a single quote character in the text at this point.
\"	Inserts a double quote character in the text at this point.
\\	Inserts a backslash character in the text at this point.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

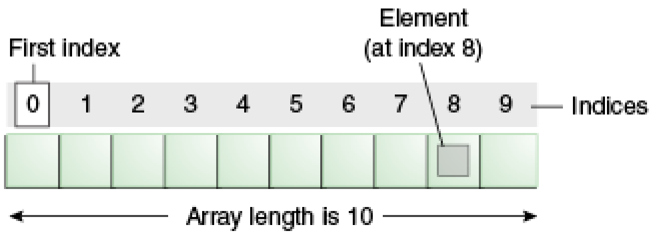
String Literals

String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes.

Example : `String s = " Hello world" ;`

Java Arrays

- An array is a group of like-typed variables that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.



One-Dimensional Arrays

A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a onedimensional array declaration is:

```
type var-name[ ];
```

Here, type declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element type for the array determines what type of data the array will hold.

For example, the following declares an array named month_days with the type “array of int” :

```
          int month_days [];
```

Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists. To link **month_days** with an actual, physical array of integers, we must allocate one using new and assign it to month_days. **new** is a special operator that allocates memory.

Instantiating an Array in Java

When an array is declared, only a reference of array is created. To actually create or give memory to array, we create an array like this:

The general form of new as it applies to one-dimensional arrays appears as follows:

```
var-name = new type [size];
```

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and **var-name** is the name of array variable that is linked to the array. That is, to use new to allocate an array, we must specify the type and number of elements to allocate.

- The elements in the array allocated by **new** will automatically be initialized to zero (for numeric types), false (for boolean), or null (for reference type)

Array Initialization

Assigning values into array locations is known as **array initialization**. Array initialization can be static or dynamic. **Static initialization** involves specifying values at the time of declaring an array as shown below:

```
     int a[ ] = {1,2,3,4,5,6,7,8,9,10};
```

An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. **There is no need to use new.**

```
//Example: Array initialization

public class TestArray {

    public static void main(String [] args){

    int arr[]={2,3,5,7,11};//array initialization


    //to print 4th element of array
System.out.println("The fourth element of array arr is :"+  arr[3]);
      //will print 7
    }
}
```

Dynamic initialization involves assigning values into an array at runtime. Following code segment demonstrates dynamic initialization of arrays:

```
      int  a[] = new int[10];
      Scanner  input = new Scanner(System.in);
      for(int i = 0; i < 10; i++)
      {
            System.out.println("Enter element number "+(i+1)+" : ");
            a[i] = input.nextInt();
      }
```

Syntax to Declare an Array in java

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

Instantiation of an Array in java

arrayRefVar=new datatype[size];

We can declare, instantiate and initialize the java array together by:

int a[]={33,3,4,5};//declaration, instantiation and initialization

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**:

```
int twoD[][] = new int [4][5];
```

This allocates a 4 by 5 array and assigns it to twoD. Internally, this matrix is implemented as an array of arrays of **int**. Conceptually, this array will look like the one shown in Figure

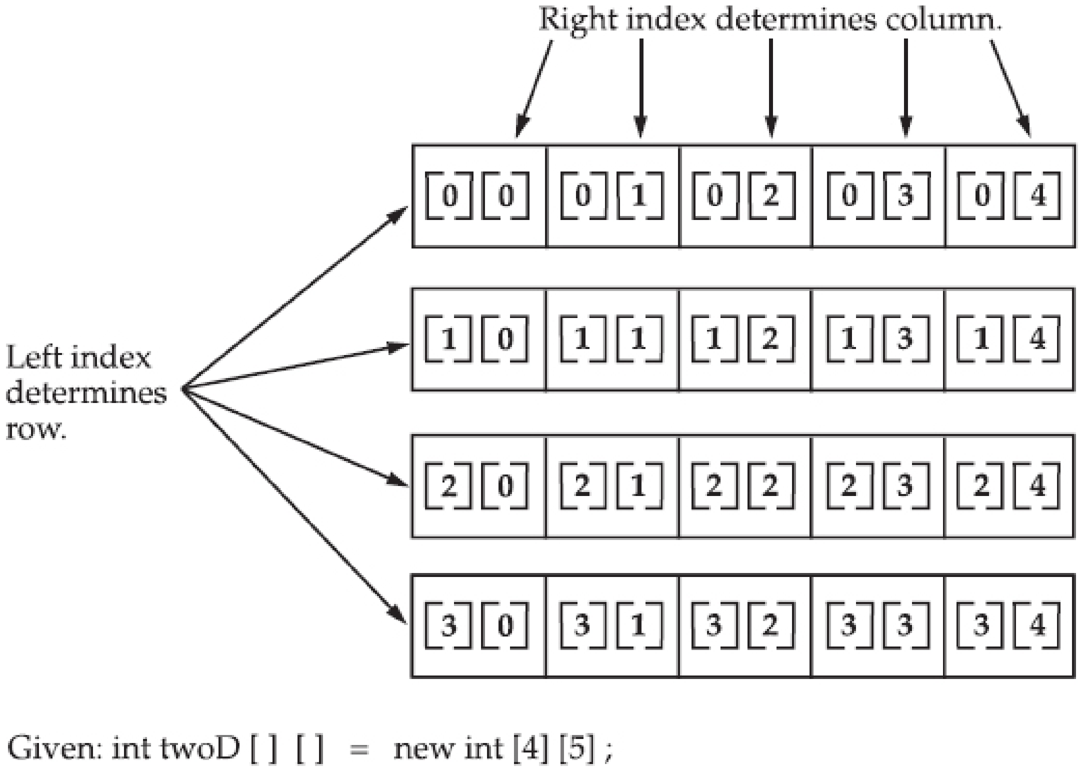


Figure 9:A conceptual view of a 4 by 5, two-dimensional array

When we allocate memory for a multidimensional array, we need only specify the memory for the first (leftmost) dimension. We can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of twoD when it is declared. It allocates the second dimension manually.

```
int twoD[][] = new int [4][];

twoD[0] = new int[5];

twoD[1] = new int[5];

twoD[2] = new int[5];

twoD[3] = new int[5];
```

Jagged Array

- Jagged means to have an uneven edge or surface. In java, a jagged array means to have a multi -dimensional array with uneven size of rows in it.

```
//Example: 2D Array
public class TwoDArray {
    public static void main(String[] Args){
    int[][] arr = new int[2][3];
    for(int i = 0; i < arr.length; i++) {
      for (int j = 0; j < arr[i].length; j++) {
           arr[i][j] = j;
           System.out.print(arr[i][j] + " ");
      }
      System.out.println("");
    }
  }
}
```

```
//Example: 3D Array
//Basically, 3d array is an array of 2d arrays.
public class ThreeDArray {
    public static void main(String[] args){
        int threeD[][][]= new int[3][4][5];
        int i,j,k;
        for(i=0;i<3;i++){
            for(j=0;j<4;j++){
                for(k=0;k<5;k++){
                    threeD[i][j][k]=i*j*k;
                }
            }
        }
        //to print array elements
        for(i=0;i<3;i++){
            for(j=0;j<4;j++){
                for(k=0;k<5;k++){
                   System.out.print(threeD[i][j][k]+ " ");
                   System.out.println();
                }
               System.out.println();
            }
        }
    }
}
```

<u>Advantage of Java Array</u>

- ✓ Code Optimization: It makes the code optimized, we can retrieve or sort the data easily.
- ✓ Random access: We can get any data located at any index position.

Java Enums

Enums were introduced in Java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

With the use of enums it is possible to reduce the number of bugs in your code.

For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium, and large. This would make sure that it would not allow anyone to order any size other than small, medium, or large.

Example:

```
class FreshJuice {
enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
FreshJuiceSize size;
}
public class FreshJuiceTest {
public static void main(String args[]){
FreshJuice juice = new FreshJuice();
juice.size = FreshJuice.FreshJuiceSize.MEDIUM ;
System.out.println("Size: " + juice.size);
}
}
```

The above example will produce the following result:

```
Size: MEDIUM
```

Note: Enums can be declared as their own or inside a class. Methods, variables, constructors can be defined inside enums as well.

Control statements

Control statements are the statements that define the flow of your program. There are 3 types of control statements in Java: Selection, iteration and jump statements.

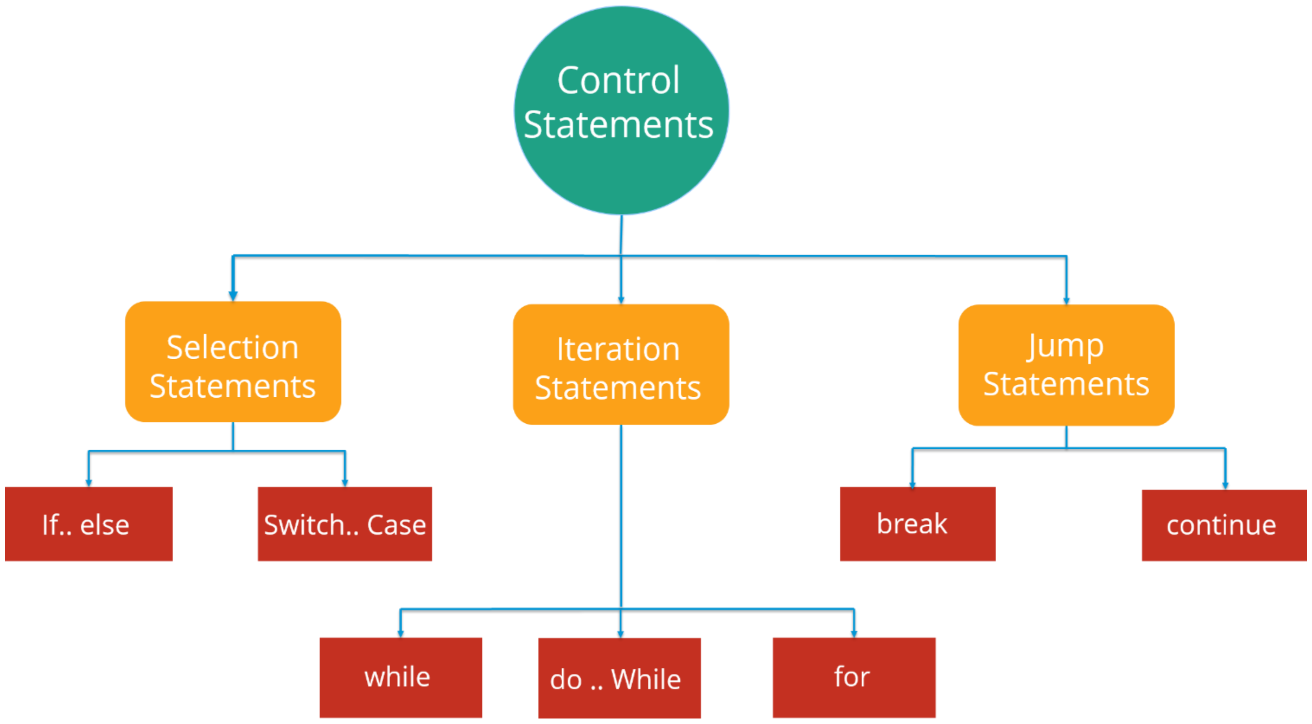


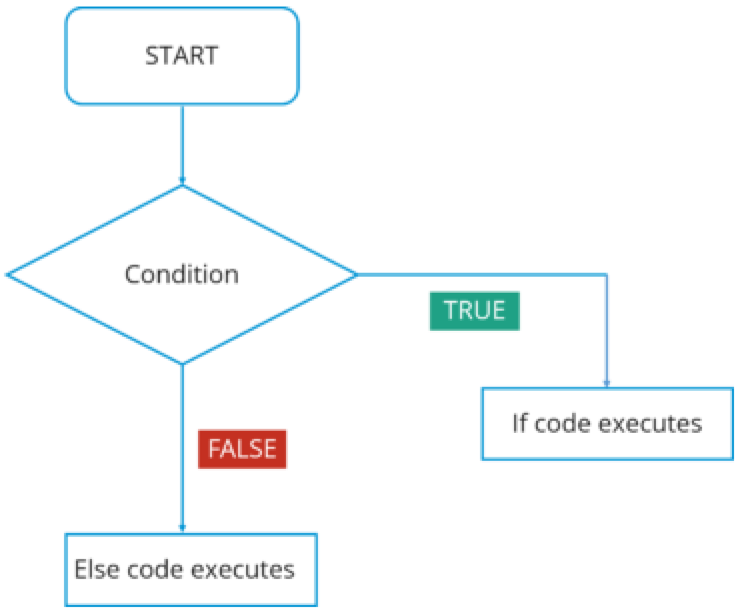
Figure 10: Control statements in Java

Selection Statements: Selection statements allow you to control the flow of the program during run time on the basis of the outcome of an expression or state of a variable. For example: you want to eat pizza, but then where can you get that pizza in best price. You can select between various popular options like Domino's, Pizza Hut or any other outlet. So here you are following a selection process from the various options available.

Now these statements can be further classified into the following:

- **If-else Statements**
- **Switch Statements**

Refer to the following flowchart to get a better understanding of if-else statements:



In this flowchart, the code will respond in the following way:

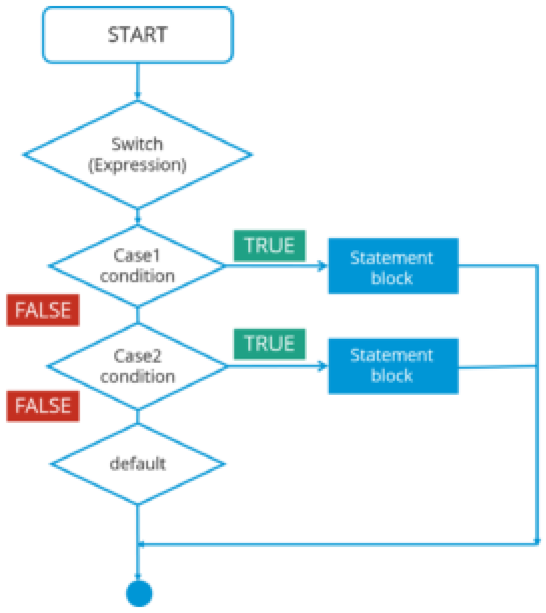
1. First of all, it will enter the loop where it checks the condition.
2. If the condition is true, the set of statements in ‘if’ part will be executed.
3. If the condition is false, the set of statements in the ‘else’ part will be executed.

```
public class Compare {

     int a=10,
     int b=5;

if(a>b)
      {  // if condition
     System.out.println(" A is greater than B");
      }
else
      {     // else condition
      System.out.println(" B is greater");
      }
}
```

In the above code, I have created a class Compare where I have compared two numbers ‘a’ and ‘b’. First of all, it will go in ‘if’ condition where it checks whether the value of ‘a’ is greater than ‘b’ or not. If the condition is true, it will print “A is greater than B” else it will execute “B is greater”.

Moving on, we have **Switch case statement**. The switch statement defines multiple paths for execution of a set of statements. It is a better alternative than using a large set of if-else statements as it is a multi-way branch statement.

Refer to the following flowchart to get a better understanding of switch statements:



In this Switch case flowchart, the code will respond in the following steps:

1. First of all it will enter the switch case which has an expression.
2. Next it will go to Case 1 condition, checks the value passed to the condition. If it is true, Statement block will execute. After that, it will break from that switch case.
3. In case it is false, then it will switch to the next case. If Case 2 condition is true, it will execute the statement and break from that case, else it will again jump to the next case.
4. Now let’s say you have not specified any case or there is some wrong input from the user, then it will go to the default case where it will print your default statement.

```
public class SwitchExample {

     int week=7;
     String weeknumber;

switch(week){     // switch case
case1:
          weeknumber="Monday";
       break;

case2:
          weeknumber="tuesday";
       break;

case3:
          weeknumber="wednesday";
       break;

default:        // default case
          weeknumber="invalid week";
       break;
     }
  System.out.println(weeknumber);
     }
}
```

In the above code, I have created a class SwitchExample which has 3 cases that print days of a week. It also has a default case which is executed whenever a user doesn't specify a case.

Concluding both of the selection statements, we understood that if we are comparing two statements, we are using if-else, but let's say if you are checking a specific value against a particular statement, then we are going for the Switch statement.

Iteration Statements: In Java, these statements are commonly called as loops, as they are used to iterate through small pieces of code. Iteration statements provide the following types of loop to handle looping requirements.

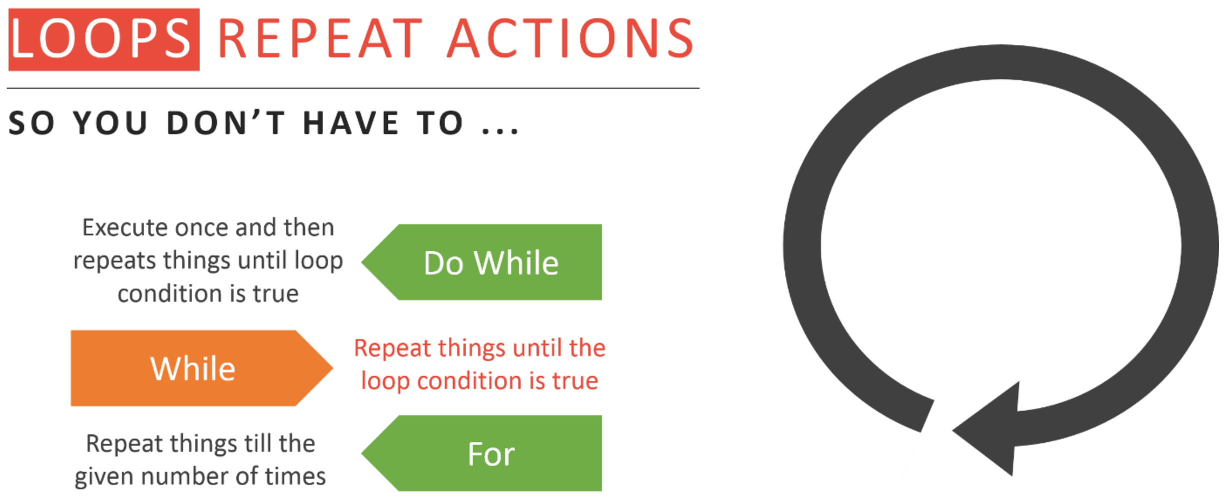
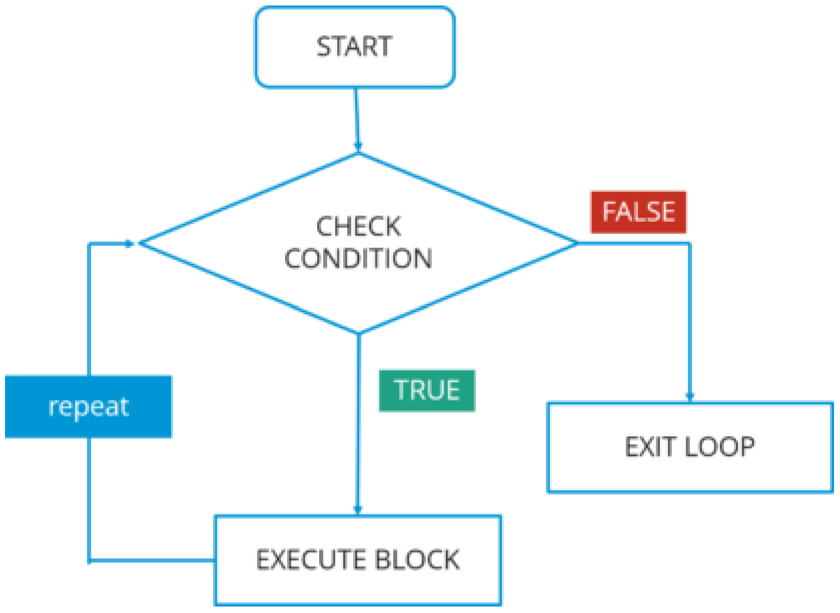

LOOPS REPEAT ACTIONS
SO YOU DON'T HAVE TO ...
Execute once and then repeats things until loop condition is true
Do While
While
Repeat things until the loop condition is true
Repeat things till the given number of times
For

Figure 11: Loops in Java

While statement: Repeat a group of statements while a given condition is true. It tests the condition before executing the loop body. Let's understand this better with a flow chart:



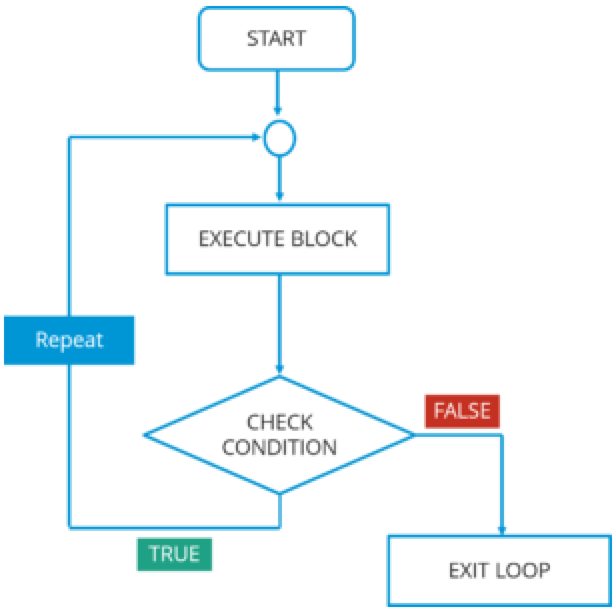
In this flowchart, the code will respond in the following steps:

1. First of all, it will enter the loop where it checks the condition.
2. If it's true, it will execute the set of code and repeat the process.
3. If it's False, it will directly exit the loop.

```
public class WhileExample {

      public static void main(String args[]) {
         int a=5;
   while(a<10)   //while condition
         {
         System.out.println("value of a" +a);
         a++;
   System.out.println("\n");
         }
    }
}
```

In the above code, it first checks the condition whether the value of a is less than 10 or not. Here, the value of a is 5 which in turn satisfy the condition, and thus perform functions

Do-while statement: It is like a while statement, but it tests the condition at the end of the loop body. Also, it will executes the program at least once. Let's understand this better with a flow chart:



In this do-while flowchart, the code will respond in the following steps:

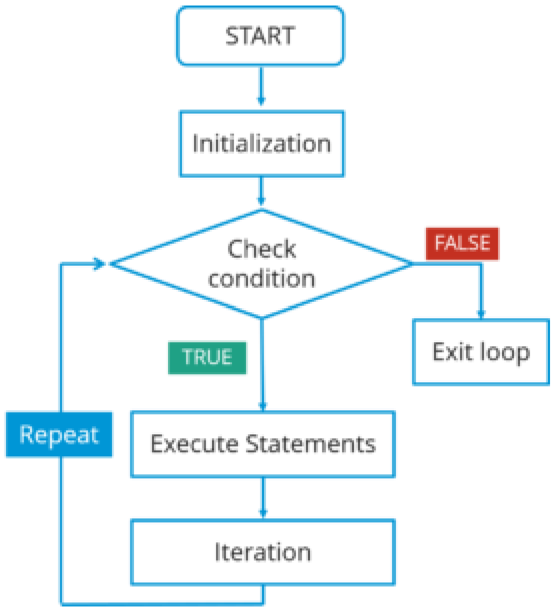
1. First of all, it will execute a set of statements that is mentioned in your 'do' block.
2. After that, it will come to 'while' part where it checks the condition.

3. If the condition is true, it will go back and execute the statements.
4. If the condition is false, it will directly exit the loop.

```
public class DoWhileExample {

      public static void main(string args[]){
          int count=1;
do {                         // do statement
     System.out.println("count is:"+count);
     count++;
   }
 while (count<10)      // while condition
       }
  }
```

In the above code, it will first execute the 'do' statements and then jump to the while part. In this program, the output would be : 1 2 3 4 5 6 7 8 9.

For statement: For statement execute a sequence of statements multiple time where you can manage the loop variable. You basically have 3 operations here: initialization, condition and iteration. Let's understand this better with a flow chart:



In this flowchart, the code will respond in the following steps:

1. First of all, it will enter the loop where it checks the condition.
2. Next, if the condition is true, the statements will be executed.
3. If the condition is false, it directly exits the loop.

```
public class ForExample {

      public static void main(String args[]) {
          for(int i=0; i<=10; i++)  // for condition
          {
          System.out.println(i);
          }
     }
}
```

In the above code, it will directly print the numbers from 1 to 10.

The For-Each Version of the for Loop

- Beginning with JDK 5, a second form of **for** was defined that implements a **"for-each"** style loop.
- A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. Unlike some languages, such as C#, that implement a for-each loop by using the keyword foreach, Java adds the for-each capability by enhancing the for statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken.
- The **for-each** style of for is also referred to as the **enhanced for loop**.
- The general form of the for-each version of the for is shown here:

  `for(type itr-var : collection) statement-block`

  Here, **type** specifies the type and **itr-var** specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by **collection**. There are various types of collections that can be used with the for, but the only type used in this chapter is the **array**.

- With each iteration of the loop, the next element in the collection is retrieved and stored in **itr-var**. The loop repeats until all elements in the collection have been obtained.
- Because the iteration variable receives values from the collection, type must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, type must be compatible with the element type of the array.
- Although the for-each for loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement

The following program uses a traditional for loop to compute the sum of the values in an array:

```
public class Sum {
    public static void main(String[] args){
        int numbers[] = {1,2,3,4,5,6,7,8,9,10};
        int sum=0;
        for(int i =0;i<numbers.length;i++){
            sum+=numbers[i];
        }
System.out.println("The sum of elements of the array is= "+sum);


    }
  }
```

The following program uses for each style loop to compute the sum of elements of an array:

```
public class ForEachDemo {
    public static void main(String[] args){
        int numbers[] = {1,2,3,4,5,6,7,8,9,10};
        int sum=0;
        //using for-each style loop
        for(int x: numbers){
            sum+=x;
        }
System.out.println("The sum of elements of the array is= "+sum);


    }
}
```

In summary:

- → It starts with the keyword for like a normal for-loop.
- → Instead of declaring and initializing a loop counter variable, we declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
- → In the loop body, we can use the loop variable we created rather than using an indexed array element.
- → It's commonly used to iterate over an array or a Collections class (eg, ArrayList)

Limitations of for-each loop

- → For-each loops are not appropriate when we want to modify the array:
- → For-each loops do not keep track of index. So we cannot obtain array index using For-Each loop
- → For-each only iterates forward over the array in single steps

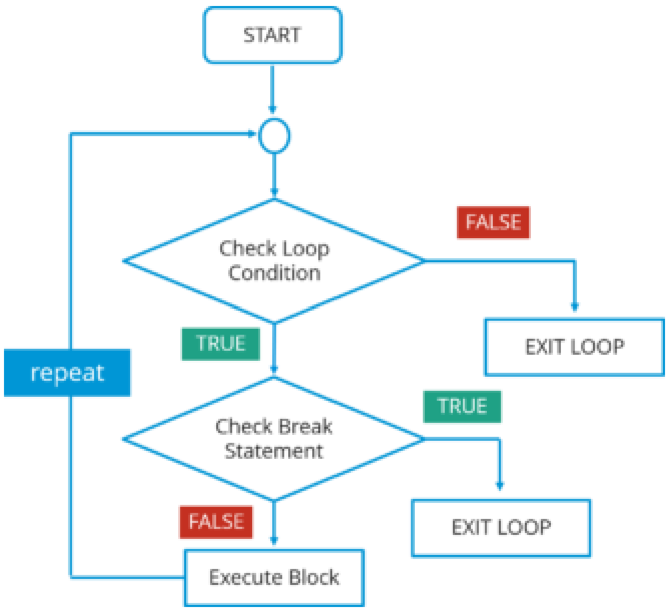
Iterating Over Multidimensional Arrays using enhanced for loop

- The enhanced version of the for also works on multidimensional arrays.
- When using the for-each for to iterate over an array of N dimensions, the objects obtained will be arrays of N–1 dimensions. For example, in the case of a two dimensional array, the iteration variable must be a reference to a one-dimensional array
  Example:

```
public class TwoD {
    public static void main(String[] args){
        int arr[][]= new int[3][5];
        int sum=0;
        //adding some values to arr
        for(int i = 0; i<3;i++){
            for(int j =0;j<5;j++){
                arr[i][j]=(i+1)*(j+1);
            }
        }
        //using for each loop to display and sum the values
        for(int x[]:arr){
            for(int y: x){
                System.out.println("The element is:"+y);
                sum = sum+y;
            }
           }
        System.out.println("Sum is = "+sum);
    }
}
```

Jump statement: Jump statement are used to transfer the control to another part of your program. These are further classified into – ***break and continue***.

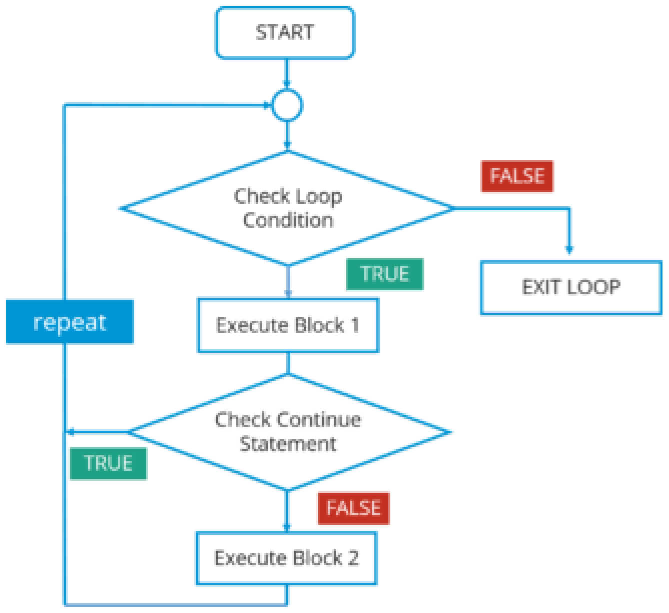
Break statement: Whenever a break statement is used, the loop is terminated and the program control is resumed to the next statement following the loop. Let's understand this better with a flow chart:



In this flowchart, the code will respond in the following steps:
1. First of all, it will enter the loop where it checks the condition.
2. If the loop condition is false, it directly exits the loop.
3. If the condition is true, it will then check the break condition.
4. If break condition is true, it exists from the loop.
5. If the break condition is false, then it will execute the statements that are remaining in the loop and then repeat the same steps.

The syntax for this statement is just the 'break' keyword followed by a semicolon

Continue statement: Continue statement is another type of control statements. The continue keyword causes the loop to immediately jump to the next iteration of the loop. Let's understand this better with a flow chart:



In this flowchart, the code will respond in the following steps:

1. First of all, it will enter the loop where it checks the condition.
2. If the loop condition is false, it directly exits the loop.
3. If the loop condition is true, it will execute block 1 statements.
4. After that it will check for ‘continue’ statement. If it is present, then the statements after that will not be executed in the same iteration of the loop.
5. If ‘continue’ statement is not present, then all the statements after that will be executed.

The syntax is just the ‘continue’ keyword followed by a semicolon.

Java – Objects & Classes

The **class** is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an **object**. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class

What Is an Object?

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

- If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.
- If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.
- If you compare the software object with a real-world object, they have very similar characteristics.
- Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.
- So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

An object has three characteristics:

- ♣ **State**: represents the data (value) of an object.
- ♣ **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- ♣ **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Class Fundamentals

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.

- Classes and objects are the fundamental components of OOP's
- A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.
- A class is a template for an object, and an object is an instance of a class. Because an object is an instance of a class, we will often see the two words object and instance used interchangeably.

Class Vs Object

A class is a template for objects. A class defines object properties including a valid range of values, and a default value. A class also describes object behavior. An object is a member or an "instance" of a class. Class is logical entity while object is physical.

The General Form of a Class

→ When we define a class, we declare its exact form and nature. We do this by specifying the data that it contains and the code that operates on that data

→ . While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

→ A class is declared by use of the **class** keyword.

A simplified general form of a class definition is shown here:

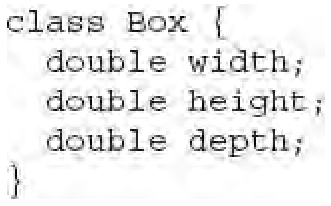
```
class classname {
     type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;

    type methodname1(parameter-list) {
      // body of method
    }
    type methodname2(parameter-list) {
      // body of method
    }
    // ...
    type methodnameN(parameter-list) {
        // body of method
    }
}
```

- → The data, or variables, defined within a class are called **instance variables**.
- → The code is contained within **methods**.
- → Collectively, the methods and variables defined within a class are called **members** of the class.
- → In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.
- → Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.
- → Thus, the data for one object is separate and unique from the data for another.
- → All methods have the same general form as **main( ),** which we have been using thus far.

- → However, most methods will not be specified as static or public. Notice that the general form of a class does not specify a main( ) method. Java classes do not need to have a main( ) method. You only specify one if that class is the starting point for your program. Further, some kinds of Java applications, such as applets, don't require a main( ) method at all.

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called Box that defines three instance variables: **width**, **height**, and **depth**.



```
class Box {
  double width;
  double height;
  double depth;
}
```

- → A class defines a new type of data. In this case, the new data type is called **Box**.
- → We can use this name to declare objects of type Box.
- → It is important to remember that **a class declaration only creates a template; it does not create an actual object.**
- → Thus, the preceding code does not cause any objects of type Box to come into existence.

Declaring Objects

- → As just explained, when we create a class, we are creating a new data type. We can use this type to declare objects of that type.
- → However, obtaining objects of a class is a two-step process.
  - o First, we must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
  - o Second, we must acquire an actual, physical copy of the object and assign it to that variable. We can do this using the **new** operator.

→ The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable. **Thus, in Java, all class objects must be dynamically allocated.**

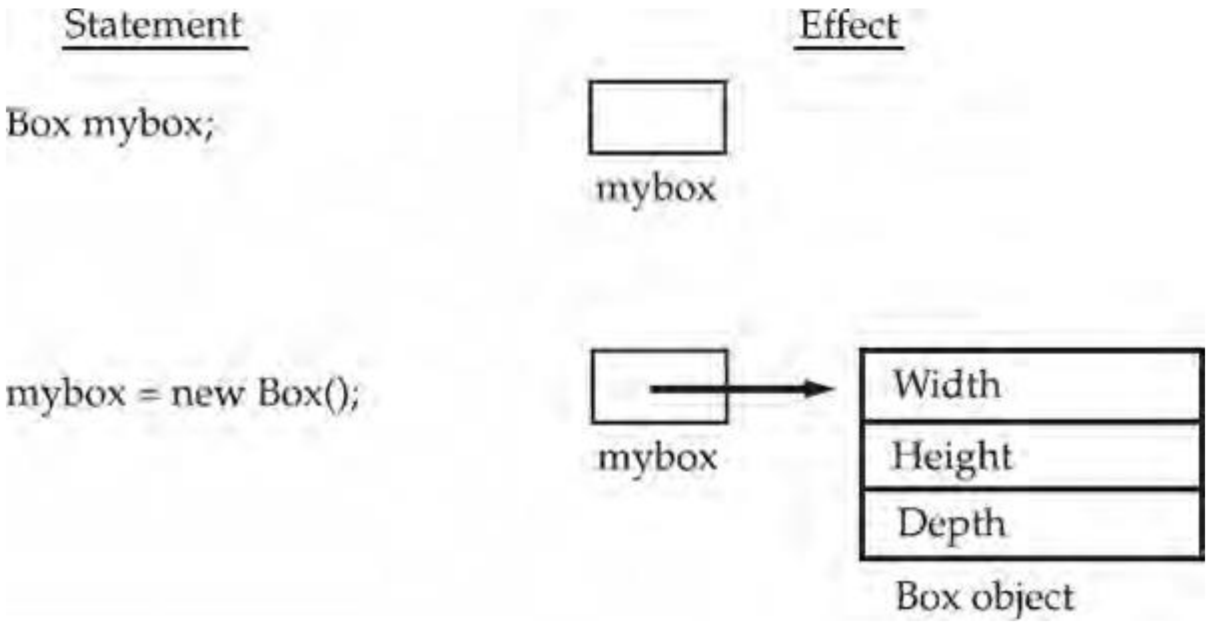
Creating an object of the class "Box" :

```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box.** At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds, in essence, the memory address of the actual **Box** object. The effect of these two lines of code is depicted in following figure:



The new operator

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

```
        class-var = new classname ( );
```

Here, **class-var** is a variable of the class type being created. The **classname** is the name of the class that is being instantiated. The class name followed by parentheses specifies the **constructor** for the class. A constructor defines what occurs when an object of a class is created.

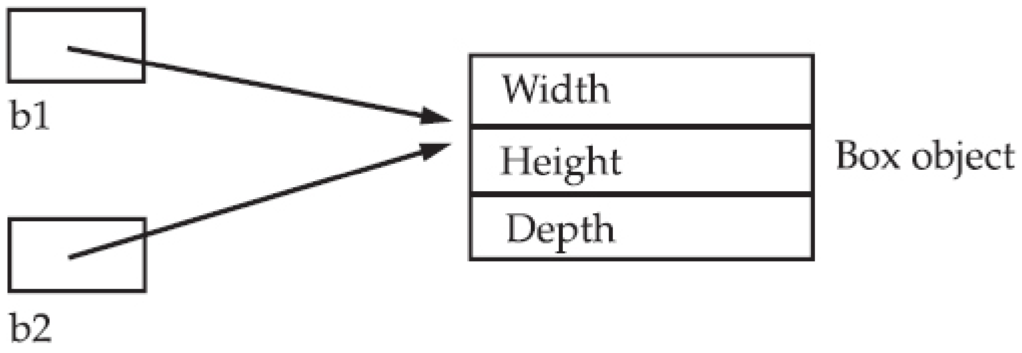
Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
     Box b1 = new Box();
     Box b2 = b1;
```

You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

This situation is depicted here:



Although b1 and b2 both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2.

For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, b1 has been set to null, but b2 still points to the original object.

Methods in Java

Classes usually consist of two things: instance variables and methods.

The general form of a method is :

```
type name(parameter-list) {
    // body of method
}
```

→ Here, **type** specifies the type of data returned by the method. This can be any valid type, including class types that you create.

→ If the method does not return a value, its return type must be **void**. The name of the method is specified by **name**. This can be any legal identifier other than those already used by other items within the current scope.

→ The **parameter-list** is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

→ Methods that have a return type other than **void** return a value to the calling routine using the following form of the return statement:

return value;

Here, **value** is the value returned.

- While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.

It is important to keep the two terms **parameter** and **argument** straight.

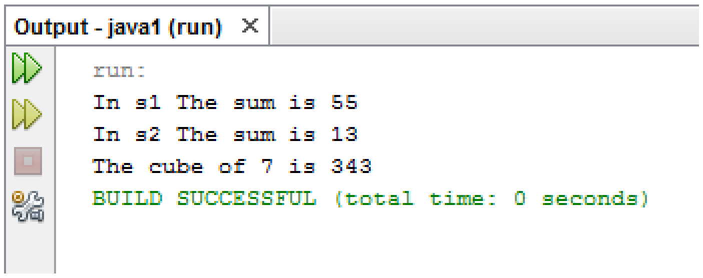
→ A **paramete**r is a variable defined by a method that receives a value when the method is called.

→ An **argument** is a value that is passed to a method when it is invoked

Example :

```
class Solver {
    int n1;  int n2;
    //method to find sum of two numbers
    //return type is 'int', name is 'getSum'
    int getSum(){
        //returning the value
        return n1+n2;
    }

    //method to find cube of a given number
    // here n is called parameter of method
    int findCube(int n ){
        int c = n*n*n;
        return c;
    }
    public static void main(String[] args) {
        //Creating objects of class Solver
        Solver s1 = new Solver();
        Solver s2 = new Solver();
        //assigning values for n1 and n2 in s1 object
        s1.n1=10;
        s1.n2=45;
        //assigning values for n1 and n2 in s2 object
        s2.n1=5;
        s2.n2=8;
        //calling getSum() by s1
        int result = s1.getSum();
```

```
            System.out.println("In s1 The sum is "+ result);
            //calling getSum() by s2
            System.out.println("In s2 The sum is "+ s2.getSum());
            //calling findCube() by s1
            int cube =s1.findCube(7);//here 7 is argument
            System.out.println("The cube of 7 is "+ cube);
        }


}
```



Output - java1 (run)

```
run:
In s1 The sum is 55
In s2 The sum is 13
The cube of 7 is 343
BUILD SUCCESSFUL (total time: 0 seconds)
```

Note: ***The concepts of the method invocation, parameters, and return values are fundamental to Java programming.***

Object initialization (assigning values to instance variables)

- Objects can be initialized using **`objectname.instanceVariableName`**
  `[In above example,` `mybox2.width=3;]`
- Objects can also me initialized using methods.
  we can use **setDim()** method where we can define this method as
  ```
  void setDim( doble a, double b, double c){
  width = a;
  height =b;
  depth =c;
  }
  ```

  Then we can call this method as
  `mybox2.setDim(4.5,6.7,8.9);`

- Object can be initialized automatically using constructors.

  It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like **setDim( ),** it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. *This automatic initialization is performed through the use of a constructor.*

Constructors

- A constructor in Java is a block of code similar to a method that's called when an instance of an object is created.
- A constructor **initializes an object immediately upon creation**.
- It has the **same name as the class** in which it resides and is syntactically similar to a method.
- Once defined, the **constructor is automatically called when the object is created, before the new operator completes.**
- Constructors look a little strange because **they have no return type, not even void.** This is because the implicit return type of a class' constructor is the class type itself.
- It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately
- In above example when **`Box mybox = new Box();`** statement is reached, **new Box( )** calls the **Box( )** constructor.
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.

Types of constructor:

1. **Default constructor**
2. **Parameterized Constructor**

Default Constructor

- If we don't define a constructor in a class, then compiler creates default constructor for the class.
- The default constructor automatically initializes all instance variables to their default values, which are **zero**, **null**, and **false**, for **numeric** types, **reference** types, and **boolean**, respectively.
- The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones.

- Once you define your own constructor, the default constructor is no longer used.
- Default constructor does not take any parameters so it is **non parameterized** constructor and sometimes also known as **no-argument** constructor or **zero-argument** constructor.

```
//example of default constructor which displays the default values
class Student{
int id;
String name;
//method to display the value of id and name
void display(){
System.out.println(id+" "+name);
}
public static void main(String args[]){
//creating objects
Student3 s1=new Student();
Student3 s2=new Student();
//displaying values of the object
s1.display();
s2.display();
}
}
```

Output:

```
0 null
0 null
```

Parameterized Constructors

- A constructor that accepts one or more parameters is known as Parameterized constructor.
- Parameters are added to a constructor in the same way that they are added to a method, we should just declare them inside the parentheses after the constructor's name.
- The parameterized constructor is used to provide different values to the distinct objects. However, you can provide the same values also

```
//Java Program to demonstrate the use of parameterized constructor
class Student{
   int id;
   String name;
   //creating a parameterized constructor
```

```
    Student (int i, String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){
          System.out.println(id+" "+name);
          }

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student(101,"Ram");
    Student4 s2 = new Student(102,"Gopal");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Differences between constructor and method in Java

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as class name.

Garbage collection

- Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator.
- Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection.
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.

- Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.
- Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

In summary:

→ In java, garbage means unreferenced objects.
→ Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
→ To do so, we were using **free()** function in C language and **delete()** in C++. **But, in java it is performed automatically**. So, java provides better memory management.
→ Advantage of Garbage Collection:
  - It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
  - It is automatically done by the garbage collector (a part of JVM) so we don't need to make extra efforts.

The finalize( ) Method

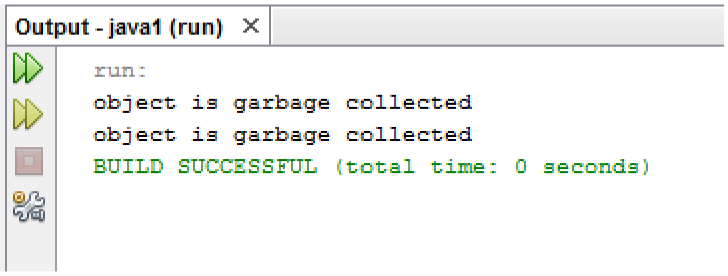
❖ Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
❖ To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
❖ To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.
  ❖ The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the finalize( ) method on the object.
  ❖ The finalize( ) method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

Here, the keyword **protected** is a specifier that limits access to **finalize( ).**

❖ It is important to understand that **finalize( )** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

```
//Example
public class TestGarbage{
 protected void finalize(){
System.out.println("object is garbage collected");
}
 public static void main(String args[]){
  TestGarbage s1=new TestGarbage();
  TestGarbage s2=new TestGarbage();
  s1=null;  //unreferencing
  s2=null;  // the objects
  System.gc();  // The gc() method is used to invoke the garbage collector to
                    //perform cleanup processing.
 }
}
```



```
run:
object is garbage collected
object is garbage collected
BUILD SUCCESSFUL (total time: 0 seconds)
```

The this Keyword

- Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword.
- **this** can be used inside any method to refer to the current object. That is, ***this** is always a reference to the object on which the method was invoked*.
- You can use this anywhere a reference to an object of the current class' type is permitted. To better understand what this refers to, consider the following version of **Box( )**

```
// A redundant use of this.
Box(double w, double h, double d) {
  this.width = w;
  this.height = h;
  this.depth = d;
}
```

This version of **Box( )** operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside **Box( ),** this will always refer to the invoking object. While it is redundant in this case, this is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding

- As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.
- However, when a local variable has the same name as an instance variable, the local variable hides the instance variable. This is why **width, height**, and **depth** were not used as the names of the parameters to the **Box( )** constructor inside the **Box** class. If they had been, then **width**, for example, would have referred to the formal parameter, hiding the instance variable **width**.
- While it is usually easier to simply use different names, there is another way around this situation. Because this lets you refer directly to the object, you can use it to resolve any

namespace collisions that might occur between instance variables and local variables. For example, here is another version of **Box( ),** which uses **width**, **height**, and **depth** for parameter names and then uses this to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt.

//Example : using **this** to overcome the instance variable hiding problem

```
class Student{
int rollno;
String name;
double  fee;
Student(int rollno,String name,double fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){
System.out.println(rollno+" "+name+" "+fee);
}

public static void main(String args[]){
Student s1=new Student(111,"Ankit",5000);
Student s2=new Student(112,"Sumit",6000);
s1.display();
s2.display();
}
}
```

Check!!!

Check what happens when you do not use **this** in above program!!

Java static keyword

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static.

The static can be:

- ✓ Variable (also known as a class variable)
- ✓ Method (also known as a class method)
- ✓ Block
- ✓ Nested class

- ♦ When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- ♦ You can declare both methods and variables to be static. The most common example of a static member is main( ). **main( )** is declared as static because it must be called before any objects exist.
- ♦ Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

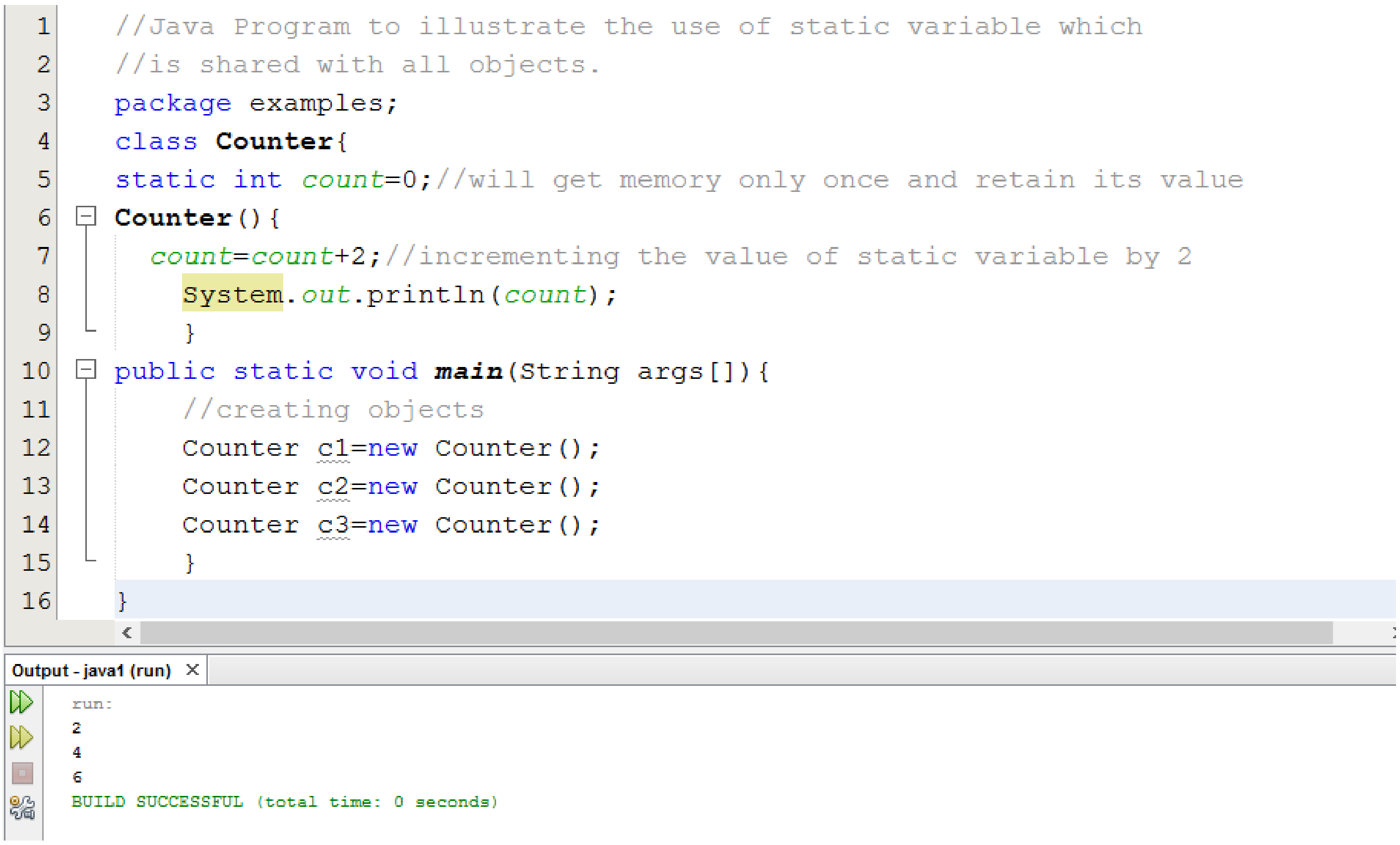
Static variable

- → The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- → The static variable gets memory only once in the class area at the time of class loading.

If you need to do computation in order to initialize your static variables, you can declare a **static block** that gets executed exactly once, when the class is first loaded.

Advantage of static variable

- ❖ It makes your program memory efficient (i.e., it saves memory).



```
//Java Program to illustrate the use of static variable which
//is shared with all objects.
package examples;
class Counter{
static int count=0;//will get memory only once and retain its value
Counter(){
 count=count+2;//incrementing the value of static variable by 2
   System.out.println(count);
   }
public static void main(String args[]){
   //creating objects
   Counter c1=new Counter();
   Counter c2=new Counter();
   Counter c3=new Counter();
   }
}
```

Output - java1 (run)

```
run:
2
4
6
BUILD SUCCESSFUL (total time: 0 seconds)
```

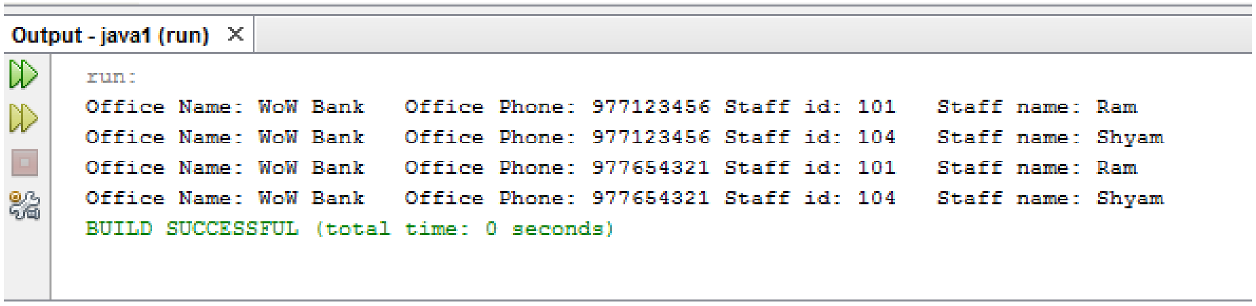
Static method

If you apply **static** keyword with any method, it is known as static method.

- → A static method belongs to the class rather than the object of a class.
- → A static method can be invoked without the need for creating an instance of a class.
- → A static method can access static data member and can change the value of it.
- → A static method can only directly call other static method (s).
- → A static method cannot refer to **this** or **super** in any way.

//Java Program to demonstrate the use of a static method.

```
package examples;
class Bank{
     int staff_id;
     String staff_name;
     static String office_name = "WoW Bank";
     static long office_phone = 977123456;
     //static method to change the value of static variable
     static void changePhone(long phn){
     office_phone = phn;
     }
     //constructor to initialize the variables
     Bank(int id, String n){
        staff_id = id;
        staff_name = n;
     }
     //method to display values
     void display(){
      System.out.println("Office Name: "+ office_name +"\t"
         +"Office Phone: "+ office_phone+"\t"
         +"Staff id: "+ staff_id +"\t"
         +"Staff name: "+ staff_name);
    }
}
//Test class to create and display the values of object
public class TestStaticMethod{
    public static void main(String args[]){
    //creating objects
    Bank stf1 = new Bank(101,"Ram");
    Bank stf2 = new Bank(104,"Shyam");
    //calling display method
    stf1.display();
    stf2.display();
    Bank.changePhone(977654321);//calling change method
    //again calling display method
    stf1.display();
    stf2.display();
    }
}
```

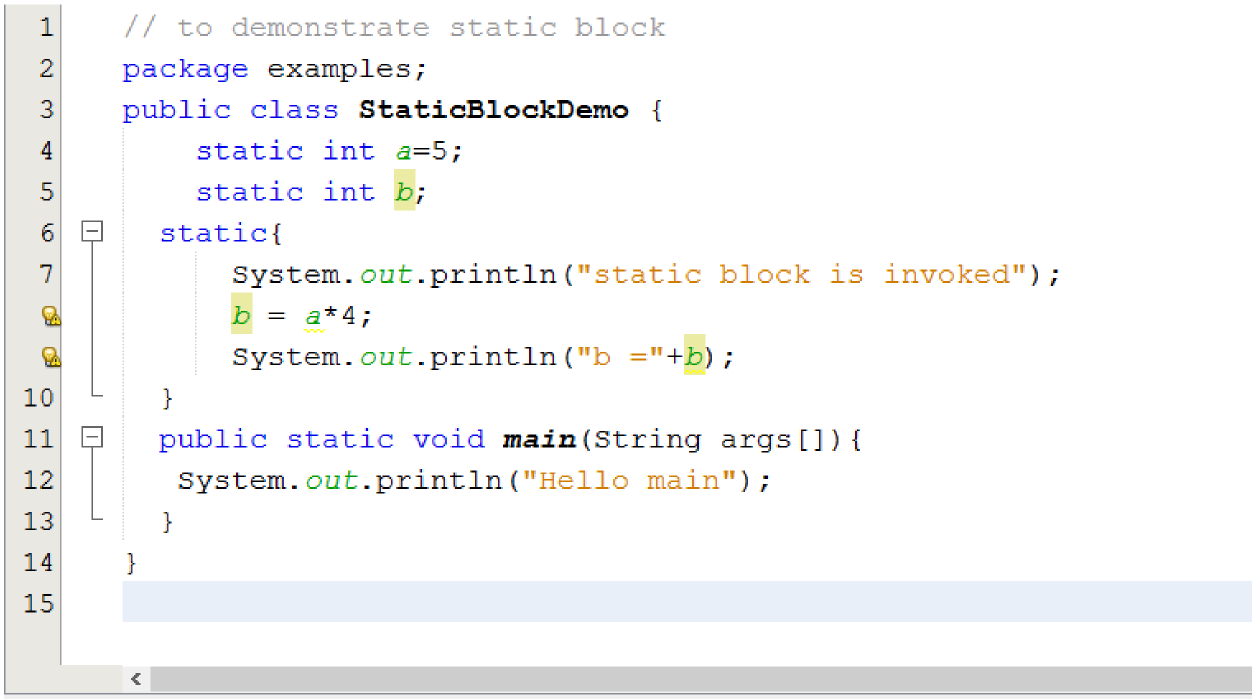


```
run:
Office Name: WoW Bank   Office Phone: 977123456 Staff id: 101   Staff name: Ram
Office Name: WoW Bank   Office Phone: 977123456 Staff id: 104   Staff name: Shyam
Office Name: WoW Bank   Office Phone: 977654321 Staff id: 101   Staff name: Ram
Office Name: WoW Bank   Office Phone: 977654321 Staff id: 104   Staff name: Shyam
BUILD SUCCESSFUL (total time: 0 seconds)
```

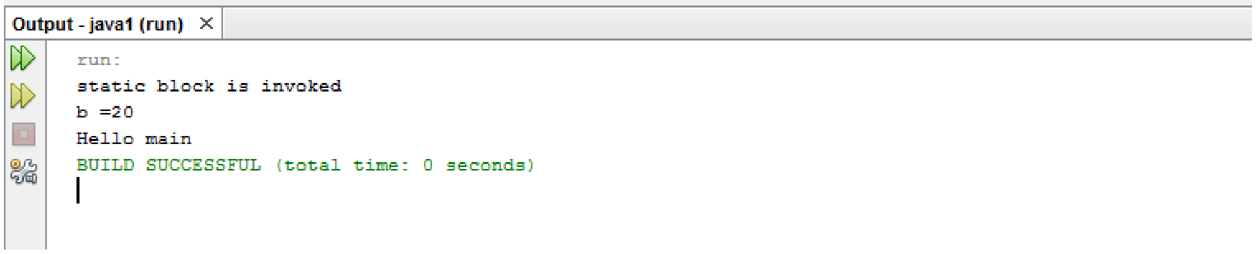
Static Block

→ Is used to initialize the static data member.

→ It is executed before the main method at the time of classloading.



```
// to demonstrate static block
package examples;
public class StaticBlockDemo {
    static int a=5;
    static int b;
  static{
      System.out.println("static block is invoked");
      b = a*4;
      System.out.println("b ="+b);
  }
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}

```



```
run:
static block is invoked
b =20
Hello main
BUILD SUCCESSFUL (total time: 0 seconds)
```

Accessing static members of a class

- Outside of the class in which they are defined, static methods and variables can be used independently of any object.
- To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a static method from outside its class, you can do so using the following general form: **`Classname.method( )`**

  Here, **Classname** is the name of the class in which the static method is declared.

- As you can see, this format is similar to that used to call non-static methods through object-reference variables.
- A static variable can be accessed in the same way—by use of the dot operator on the name of the class.
  `Classname.staticVariableName`
- This is how Java implements a controlled version of global methods and global variables.

Why is the Java main method static?

- ➢ It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call **main()** method that will lead the problem of extra memory allocation.

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }

  static {
    System.out.println("Static block initialized.");
    b = a * 4;
  }

  public static void main(String args[]) {
    meth(42);
  }
}
```

```
//Program of static variable

class Student8{
   int rollno;
   String name;
   static String college ="ITS";

   Student8(int r,String n){
   rollno = r;
   name = n;
   }
 void display (){System.out.println(rollno+" "+name+" "+college);}

 public static void main(String args[]){
 Student8 s1 = new Student8(111,"Karan");
 Student8 s2 = new Student8(222,"Aryan");

 s1.display();
 s2.display();
 }
```

```
Output:
111 Karan ITS
222 Aryan ITS
```

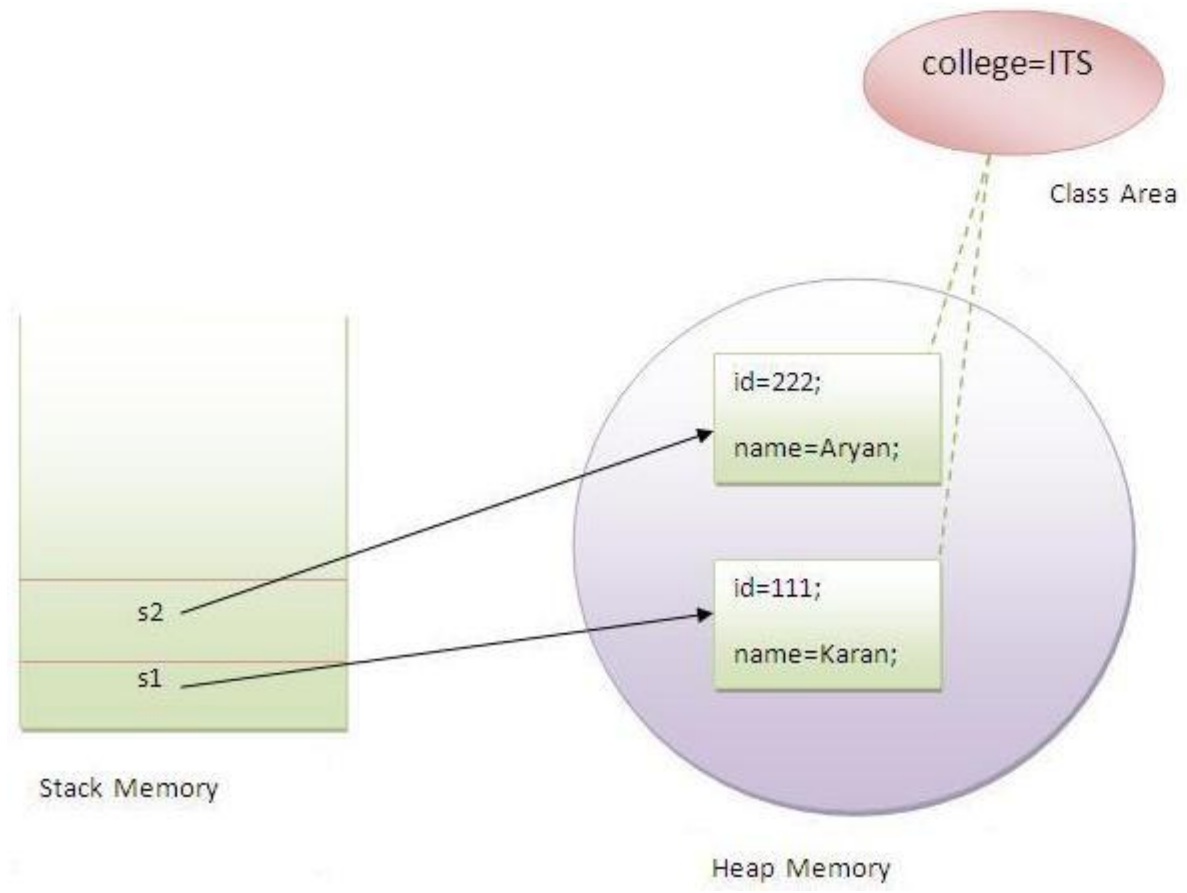


Figure 12:Static variable in java and memory

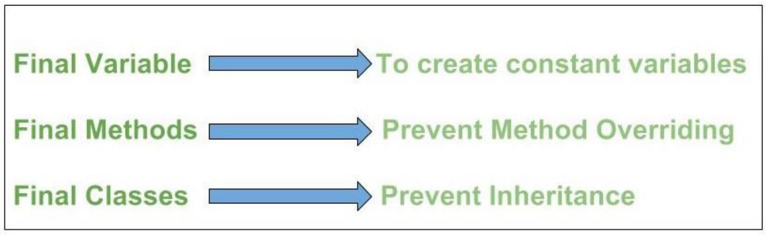
Introducing final [final keyword]

- **final** keyword is a **non-access modifier**
- The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:
    - variable (field)
    - method
    - class

A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a **constant**. This means that you must initialize a final field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common.

In addition to fields, both method parameters and local variables can be declared final. Declaring a parameter final prevents it from being changed within the method. Declaring a local variable final prevents it from being assigned a value more than once. The keyword final can also be applied to methods and class.

- If we make any **variable as final**, we cannot change the value of final variable (It **will be constant**) [**final** variable once assigned a value can never be changed.]
- If we make any **method as final**, we **cannot override** it.
- If we make any class as final, we cannot extend it (i.e. we cannot create subclass)



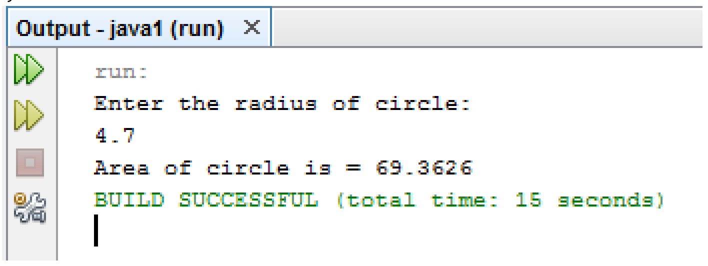
- **NOTE: final variables are declared in upper case (convention)**

The following program will produce compile time error

```
 class Bike{

final int SPEED_LIMIT=50;//final variable
void run(){
SPEED_LIMIT =110;  // We cannot reassign the value to final variable . ERROR occurs !
}
public static void main(String args[]){
Bike mybike=new  Bike9();
obj.run();
}
}//end of class
```

//Program to show the use of final variable

```
package examples;
import java.util.Scanner;
public class MyCircle {
    final double PI = 3.14;
    double r;
    void getRad(){
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the radius of circle:");
        r = sc.nextDouble();
    }
    void circleArea(){
        double a = PI*r*r;
        System.out.println("Area of circle is = "+a);
    }
    public static void main(String[] args) {
        MyCircle c1 = new MyCircle();
        c1.getRad();
        c1.circleArea();
    }
}
```



```
run:
Enter the radius of circle:
4.7
Area of circle is = 69.3626
BUILD SUCCESSFUL (total time: 15 seconds)
```

Java Package

- ✓ A java package is a group of similar types of classes, interfaces and sub-packages.
- ✓ Package in java can be categorized in two form, built-in package and user-defined package.
- ✓ There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package.This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

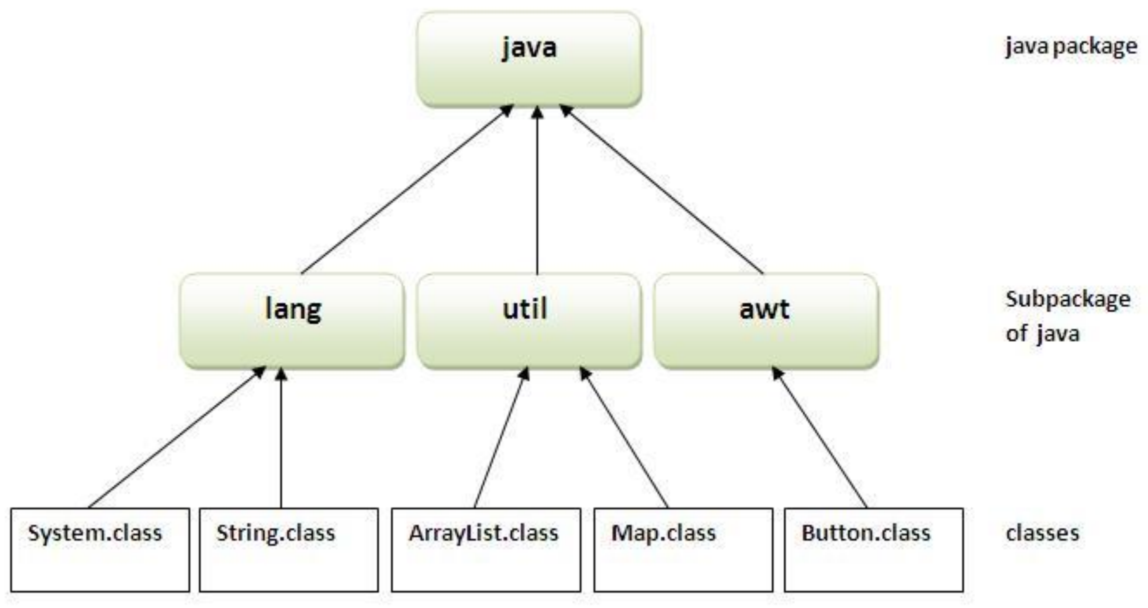


Figure 13: Java Packages

- The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  }
}
```

- ❖ How to access package from another package?
  - ✓ There are three ways to access the package from outside the package.

1. **import package.***

   If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

   The **import keyword** is used to make the classes and interface of another package accessible to the current package.

2. **import package.classname**

   If you import package.classname then only declared class of this package will be accessible.

3. **fully qualified name**

   If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

   It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Subpackage in java

Package inside the package is called the **subpackage**. It should be created to categorize the package further.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.