## Ishaphj@gmail

gudushrma@gmail

## Machine Language

## Assembly Language

## Compiler

## Interpreter

## Structured

## Non Structured

## OOPS

## Java History

James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language

project in June 1991. The small team of sun engineers was called Green Team.

Originally designed for small, embedded systems in electronic appliances like set-top boxes. It was called Oak and was developed as a part of the Green project.

In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.

Originally developed by James Gosling at Sun Microsystems (which is now a

subsidiary of Oracle Corporation) and released in 1995.

## Java Version

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

JDK Alpha and Beta (1995)

JDK 1.0 (23rd Jan, 1996)

JDK 1.1 (19th Feb, 1997)

J2SE 1.2 (8th Dec, 1998)

J2SE 1.3 (8th May, 2000)

J2SE 1.4 (6th Feb, 2002)

J2SE 5.0 (30th Sep, 2004)

Java SE 6 (11th Dec, 2006)

Java SE 7 (28th July, 2011)

Java SE 8 (18th March, 2014)

## JDK Editions

Java Standard Edition (J2SE)

J2SE can be used to develop client-side standalone applications or applets.

Java Enterprise Edition (J2EE)

J2EE can be used to develop server-side applications such as Java servlets and Java

Server Pages.

Java Micro Edition (J2ME):

J2ME can be used to develop applications for mobile devices such as cell phones.

Which one do we use and why, what’s the use of the rest?

What is JAVA and its Features

Java technology is both a programming language and a platform.

The Java programming language is a high-level language that can be characterized by following buzzwords,

* Simple
* Object-Oriented
* Platform independent
* Portable
* High Performance
* Secured
* Robust
* Multithreaded
* Architecture Neutral

### Simple:

## According to Sun, Java language is simple because:

* Syntax is based on C++ (so easier for programmers to learn it after C++).
* Removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.
* No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

### Object Oriented:

## Object-Oriented Programming Language (OOPs) is the methodology which provide software development and maintenance by using object state, behavior, and properties.

**Java Feature: Platform Independence**

* Java Language is platform-independent due to its hardware and software environment. Java code can run on multiple platforms e.g. windows, Linux, sun Solaris, Mac/OS etc.
* Java code is compiled by the compiler and converted into byte code. This byte code is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).



**Java Feature: Portable and High Performance**

**Portable**: We may carry the java bytecode to any platform.

**High performance**: For all but the simplest or most infrequently used applications, performance is always a consideration for most applications, including graphics-intensive ones such as are commonly found on the world wide web, the performance of java is more than adequate.

**Secure**

Java apps are used in distributed environments too. Thus, lot of emphasis is on security.

The Java language is secure in the sense that it is very difficult to write incorrect code for viruses that can corrupt/steal your data, or harm hardware such as hard disks.

There are some main lines of defense:

* Interpreter level:
* No pointer arithmetic
* Garbage collection
* Array bounds checking
* No illegal data conversions
* Byte Code Verifier

**Java Feature: Robust**

* Reliable
* Early checking for potential problems.
* Dynamic checking to eliminate error-prone situations.
* Developer doesn’t have to worry about
* Bad pointers
* Memory allocation errors
* Memory leakage

**Java Feature: Multithreaded**

Thread-safe: Lib functions implemented such that it can be executed by multiple concurrent threads

Built-in support for threads

**Java Feature: Architecture Neutral**



**Structured vs Object Oriented Approach**



Concept of Class and Object

**Class**

A class can be defined as a template/ blue print from which individual objects are created. And that describes the behaviors/states that object of its type support.

Class is a design whereas object is a real entity based on the class.

**Object**

An object is a real world entity that can be a tangible, intangible or a conceptual entity.

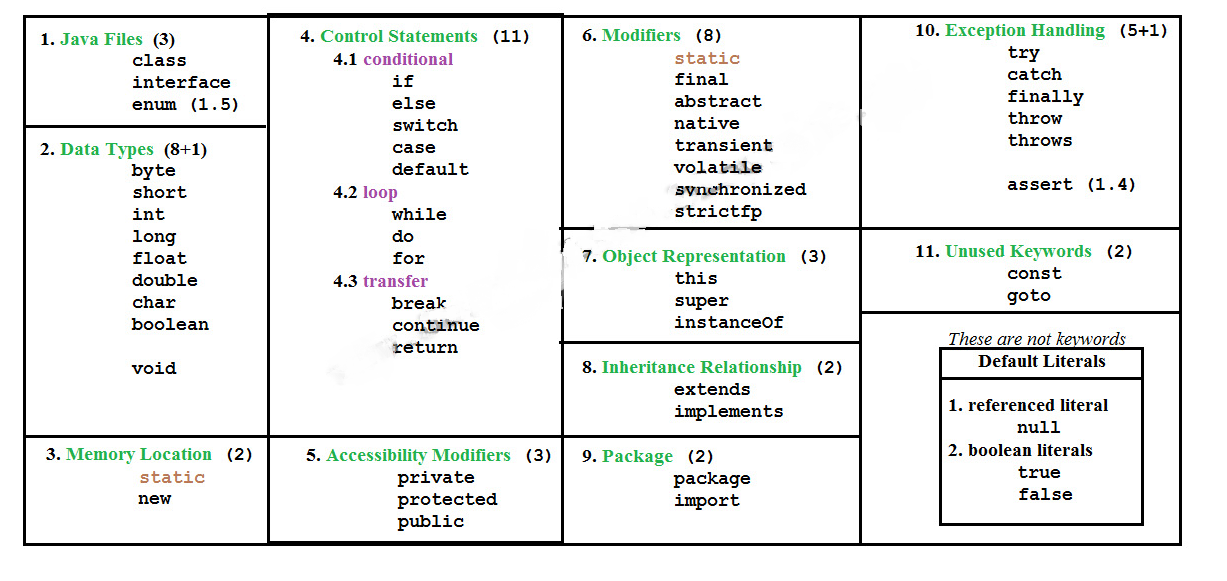
An instance of a class

An object can be considered as a "thing" that can perform a set of activities. The set of activities that the object performs defines the object's behavior. For example, the hand(object) can Grip something or a Student (object) can give the name or address.

Keywords

Every programming language uses some reserved words known as keywords.

They are an essential part of the language. In Java, keywords are defined in lower case and cannot be used as identifiers.



Data Types and Variables

Data Types determines the values it may contain, plus the operations that may be performed on it.

Variables provides us with named storage that our programs can manipulate

**Variable Declaration & Initialization**

**Syntax for declaration of variable:**

DataType variableName;

**Syntax for assignment of value to a variable:**

variableName=value;

**In a single statement declaration and assignment of variable:**

DataType variableName=value;

**In a single statement declaration and assignment of many variables:**

DataType variableName=value, variableName1=value1…;

### *The Scope and Lifetime of Variables*

All of the variables used have been declared at the start of the main( ) method.

However, 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.

In Java, the two major scopes are those defined by a class and those defined by a method.

However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method,

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method’s scope.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. 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.

To understand the effect of nested scopes, consider the following program:

**// 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);**

**}**

**}**

As the comments indicate, the variable x is declared at the start of main( )’s scope and is

accessible to all subsequent code within main( ). Within the if block, y is declared. Since a

block defines a scope, y is only visible to other code within its block. This is why outside of

its block, the line y = 100; is commented out. If you remove the leading comment symbol,

a compile-time error will occur, because y is not visible outside of its block. Within the if

block, x can be used because code within a block (that is, a nested scope) has access to

variables declared by an enclosing scope.

#### DATA TYPES

#### Java Data Types can be classified into two types:

Primitive

Non-Primitive

#### Primitive:



***float***

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is

faster on some processors and takes half as much space as double precision, but will become

imprecise when the values are either very large or very small. Variables of type float are

useful when you need a fractional component, but don’t require a large degree of precision.

For example, float can be useful when representing dollars and cents.

Here are some example float variable declarations:

float hightemp, lowtemp;

***double***

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double

precision is actually faster than single precision on some modern processors that have been

optimized for high-speed mathematical calculations. All transcendental math functions,

such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy

over many iterative calculations, or are manipulating large-valued numbers, double is the

best choice.

Here is a short program that uses double variables to compute the area of a circle:

**// Compute the area of a circle.**

**class Area {**

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

**double pi, r, a;**

**r = 10.8; // radius of circle**

**pi = 3.1416; // pi, approximately**

**a = pi \* r \* r; // compute area**

**System.out.println("Area of circle is " + a);**

**}**

**}**

***Characters***

Here is a program that demonstrates char variables:

**// Demonstrate char data type.**

**class CharDemo {**

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

**char ch1, ch2;**

**ch1 = 88; // code for X**

**ch2 = 'Y';**

**System.out.print("ch1 and ch2: ");**

**System.out.println(ch1 + " " + ch2);**

**}**

**}**

**This program displays the following output:**

**ch1 and ch2: X Y**

Notice that ch1 is assigned the value 88, which is the ASCII (and Unicode) value that

corresponds to the letter X. As mentioned, the ASCII character set occupies the first 127

values in the Unicode character set.

**// char variables behave like integers.**

**class CharDemo2 {**

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

**char ch1;**

**ch1 = 'X';**

**System.out.println("ch1 contains " + ch1);**

**ch1++; // increment ch1**

**System.out.println("ch1 is now " + ch1);**

**}**

**}**

**The output generated by this program is shown here:**

**ch1 contains X**

**ch1 is now Y**

In the program, ch1 is first given the value X. Next, ch1 is incremented. This results in ch1

containing Y, the next character in the ASCII (and Unicode) sequence.

NOTE In the formal specification for Java, char is referred to as an integral type, which means that it is

in the same general category as int, short, long, and byte. However, because its principal use is for

representing Unicode characters, char is commonly considered to be in a category of its own.

***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.

Here is a program that demonstrates the boolean type:

**// Demonstrate boolean values.**

**class BoolTest {**

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

**boolean b;**

**b = false;**

**System.out.println("b is " + b);**

**b = true;**

**System.out.println("b is " + b);**

**// a boolean value can control the if statement**

**if(b) System.out.println("This is executed.");**

**b = false;**

**Chapter 3 Data Types, Variables, and Arrays 41**

**Part I**

**if(b) System.out.println("This is not executed.");**

**// outcome of a relational operator is a boolean value**

**System.out.println("10 > 9 is " + (10 > 9));**

**}**

}

The output generated by this program is shown here:

b is false

b is true

This is executed.

10 > 9 is true

**Non-Primitive:**

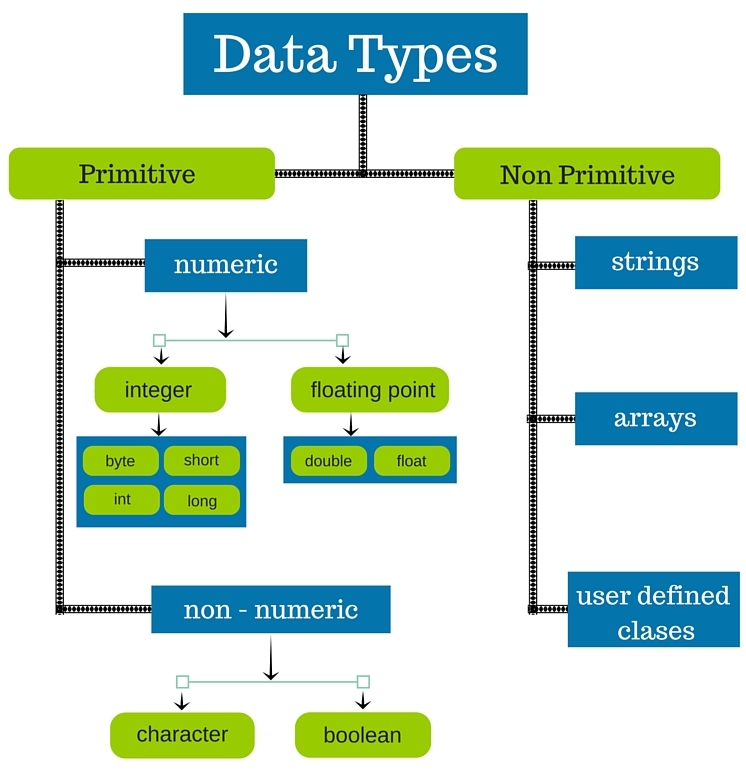
**String**

Strings, which are widely used in Java programming, are a sequence of characters.

It is an inbuilt class in java.

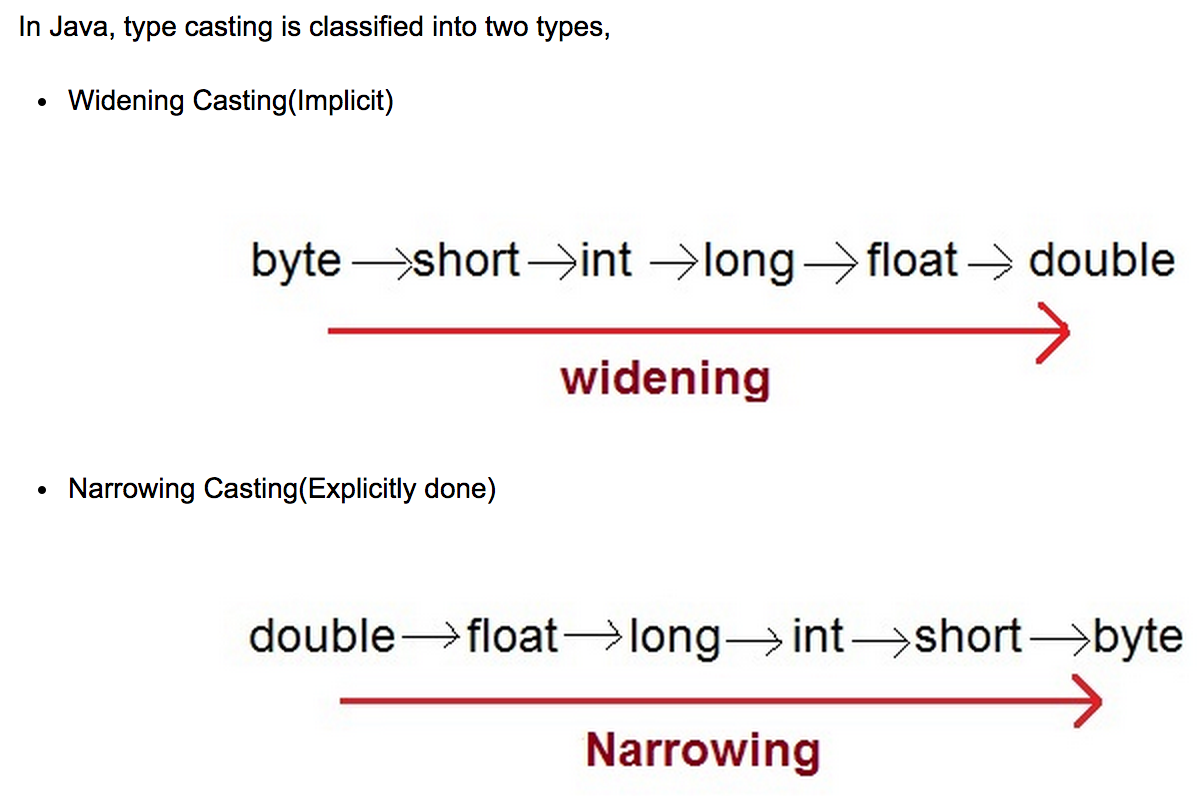
The most direct way to create a string is to write:

String greeting = "Hello world!";



TypeCasting

Assigning a value of one type to a variable of another type is known as Type Casting.



Widening Casting/Implicit/ Casting/Automatic Type casting - It takes place when the two types are compatible & the target type is larger than the source type.

Narrowing Casting/Explicit type casting - When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

**// Demonstrate casts.**

**class Conversion {**

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

**byte b;**

**int i = 257;**

**double d = 323.142;**

**System.out.println("\nConversion of int to byte.");**

**b = (byte) i;**

**System.out.println("i and b " + i + " " + b);**

**System.out.println("\nConversion of double to int.");**

**i = (int) d;**

**System.out.println("d and i " + d + " " + i);**

**System.out.println("\nConversion of double to byte.");**

**b = (byte) d;**

**System.out.println("d and b " + d + " " + b);**

**}**

**}**

**Automatic Type Promotion in Expressions**

In addition to assignments, there is another place where certain type conversions may

occur: in expressions. To see why, consider the following. In an expression, the precision

required of an intermediate value will sometimes exceed the range of either operand.

**For example, examine the following expression:**

**byte a = 40;**

**byte b = 50;**

**byte c = 100;**

**int d = a \* b / c;**

The result of the intermediate term a \* b easily exceeds the range of either of its byte

operands. To handle this kind of problem, Java automatically promotes each byte, short,

or char operand to int when evaluating an expression. This means that the subexpression

a\*b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate

expression, 50 \* 40, is legal even though a and b are both specified as type byte.

As useful as the automatic promotions are, they can cause confusing compile-time

errors.

For example, this seemingly correct code causes a problem:

**byte b = 50;**

**b = b \* 2; // Error! Cannot assign an int to a byte!**

The code is attempting to store 50 \* 2, a perfectly valid byte value, back into a byte

variable. However, because the operands were automatically promoted to int when the

expression was evaluated, the result has also been promoted to int. Thus, the result of the

expression is now of type int, which cannot be assigned to a byte without the use of a cast.

This is true even if, as in this particular case, the value being assigned would still fit in the

target type.

In cases where you understand the consequences of overflow, you should use an explicit

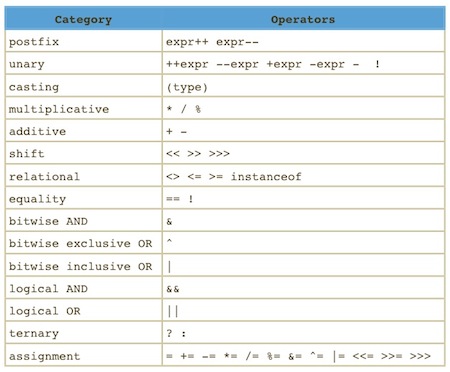
cast, such as

**byte b = 50;**

**b = (byte)(b \* 2);**

**which yields the correct value of 100.**

Operators in Java



Introduction to JDK,JVM and JRE

JDK is Java Developer Kit which you need to compile Java source code

JVM is Java Virtual Machine which runs Java bytecode.

JRE is Java Runtime Environment which you need to run a java program. It contains a JVM, among other things.

* 