CHAPTER 2

JAVA TOKENS

2.1 Java Unicode Character Set

The smallest units of Java language are the characters used to write Java tokens. These characters are defined by the *Unicode* character set, an emerging standard that tries to create characters.

The Unicode is a 16-bit character coding system and currently support more than 34,000 defined characters derived from 24 languages from America, Europe, Middle East, Africa and Asia (including India). However, most of us use only the basic ASCII characters, which include letters, digits and punctuation marks, used in normal English. ASCII character set is a subset of Unicode character set.

Why java uses Unicode System?

Before Unicode, there were many language standards:

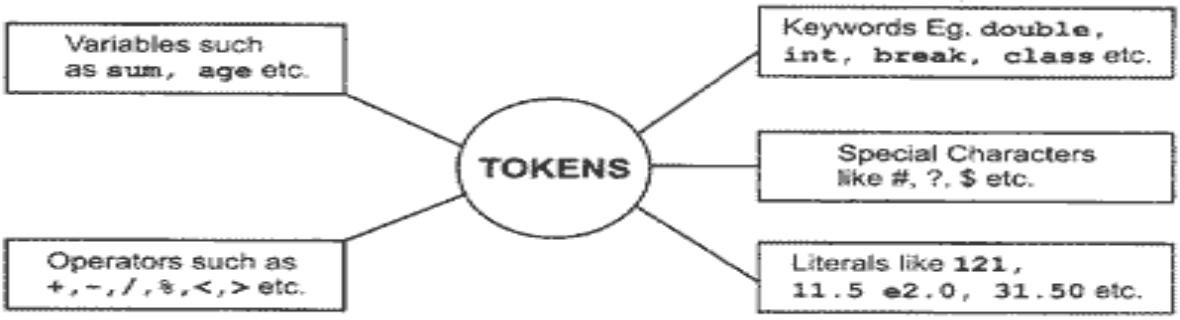
- ASCII (American Standard Code for Information Interchange) for the United States.
- ISO 8859-1 for Western European Language.
- KOI-8 for Russian.
- GB18030 and BIG-5 for chinese, and so on.

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

2.2 Java Tokens

A java Program is made up of Classes and Methods and in the Methods are the container of the various statements and a statement is made up of variables, constants, operators etc.

Tokens are the various Java program elements which are identified by the compiler. *A token is the smallest element of a program that is meaningful to the compiler*. Tokens supported in Java include keywords, variables, constants, special characters, operations etc.



Tokens in Java

When you compile a program, the compiler scans the text in your source code and extracts individual tokens. While tokenizing the source file, the compiler recognizes and subsequently removes whitespaces (spaces, tabs, newline and form feeds) and the text enclosed within comments. Now let us consider a program

```
//Print Hello
public class Hello
{
public static void main(String args[])
{
System.out.println(“Hello Java”);
}
}
```

The source code contains tokens such as public, class, Hello, {, public, static, void, main, (, String, [], args, {, System, out, println, (, "Hello Java", }, }. The resulting tokens· are compiled into Java bytecodes that is capable of being run from within an interpreted java environment. Token are useful for compiler to detect errors. When tokens are not

arranged in a particular sequence, the compiler generates an error message.

Java language includes five types of tokens and they are:

- Reserved Keywords
- Identifiers
- Literals
- Operators
- Separators

2.2.1 Keywords or Reserved Words

Java reserved words are keywords that are reserved by Java for particular uses and cannot be used as identifiers (e.g., variable names, function names, class names). Each of these 50 keywords has a specific meaning in the Java programming language. You can't use a keyword for anything other than the meaning described in table 3.1.

Also, you can't make up new meanings for the words **false**, **null**, and **true**. But for technical reasons, the words false, null, and true aren't called keywords. Whatever!

List of Java Keywords

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

Table 2.1 shows a list of the words and their purpose.

Keyword	Purpose
abstract	Indicates that the details of a class, a method, or an interface are given elsewhere in the code.
assert	Tests the truth of a condition that the programmer believes is true.
boolean	Indicates that a value is either true or false.
Break	Jumps out of a loop or switch.
Byte	Indicates that a value is an 8-bit whole number.
Case	Introduces one of several possible paths of execution in a switch statement.
Catch	Introduces statements that are executed when something interrupts the flow of execution in a try clause.
Char	Indicates that a value is a character (a single letter, digit, punctuation symbol, and so on) stored in 16 bits of memory.
Class	Introduces a class – a blueprint for an object.
Const	You can't use this word in a Java program. The word has no meaning. Because it's a keyword, you can't create a const variable.
Continue	Forces the abrupt end of the current loop iteration and begins another iteration.
Default	Introduces a path of execution to take when no case is a match in a switchstatement.
Do	Causes the computer to repeat some statements over and over again (for example, as long as the computer keeps getting unacceptable results).
Double	Indicates that a value is a 64-bit number with one or more digits after the decimal point.
Else	Introduces statements that are executed when the condition in an if statement isn't true.

Enum	Creates a newly defined *type* — a group of values that a variable can have.
extends	Creates a subclass — a class that reuses functionality from a previously defined class.
final	Indicates that a variable's value cannot be changed, that a class's functionality cannot be extended, or that a method cannot be overridden.
finally	Introduces the last will and testament of the statements in a try clause.
float	Indicates that a value is a 32-bit number with one or more digits after the decimal point.
For	Gets the computer to repeat some statements over and over again (for example, a certain number of times).
goto	You can't use this word in a Java program. The word has no meaning. Because it's a keyword, you can't create a goto variable.
If	Tests to see whether a condition is true. If it's true, the computer executes certain statements; otherwise, the computer executes other statements.
implements	Reuses the functionality from a previously defined interface.
import	Enables the programmer to abbreviate the names of classes defined in a package.
instanceof	Tests to see whether a certain object comes from a certain class.
int	Indicates that a value is a 32-bit whole number.
interface	Introduces an interface, which is like a class, but less specific. (Interfaces are used in place of the confusing multiple-inheritance feature that's in C++.)
long	Indicates that a value is a 64-bit whole number.
native	Enables the programmer to use code that was written in another language (one of those awful languages other than

	Java).
new	Creates an object from an existing class.
package	Puts the code into a package — a collection of logically related definitions.
private	Indicates that a variable or method can be used only within a certain class.
protected	Indicates that a variable or method can be used in subclasses from another package.
Public	Indicates that a variable, class, or method can be used by any other Java code.
Return	Ends execution of a method and possibly returns a value to the calling code.
Short	Indicates that a value is a 16-bit whole number.
Static	Indicates that a variable or method belongs to a class, rather than to any object created from the class.
Strictfp	Limits the computer's ability to represent extra large or extra small numbers when the computer does intermediate calculations on float and doublevalues.
Super	Refers to the superclass of the code in which the word *super* appears.
Switch	Tells the computer to follow one of many possible paths of execution (one of many possible cases), depending on the value of an expression.
synchronized	Keeps two threads from interfering with one another.
This	A self-reference — refers to the object in which the word *this* appears.
throw	Creates a new exception object and indicates that an exceptional situation (usually something unwanted) has occurred.
throws	Indicates that a method or constructor may pass the buck when an exception is thrown.

transient	Indicates that, if and when an object is serialized, a variable's value doesn't need to be stored.
try	Introduces statements that are watched (during runtime) for things that can go wrong.
void	Indicates that a method doesn't return a value.
volatile	Imposes strict rules on the use of a variable by more than one thread at a time.
while	Repeats some statements over and over again (as long as a condition is still true).

51. false :- In Java, false keyword is the literal for the data type Boolean. Expressions are compared by this literal value.

52. null :- In Java, null keyword is the literal for the reference variable. Expressions are compared by this literal value. It is a reserved keyword.

53. true :- In Java, true keyword is the literal for the data type Boolean. Expressions are compared by this literal value.

2.2.2 Data Types

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

Primitive Data Types

Reference/Object Data Types

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. Let's see what this means. 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 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.

2.2.2.1 The Primitive Types:

Data Type	Characteristics	Range
byte	8 bit signed integer	-128 to 127
short	16 bit signed integer	-32768 to 32767
int	32 bit signed integer	-2,147,483,648 to 2,147,483,647 (About) -2 million to 2 million
long	64 bit signed integer	-9,223,372,036,854,775,808 to -9,223,372,036,854,775,807 (about) -10E18 to 10E18
float	32 bit floating point number	± 1.4E-45 to ± 3.4028235E+38

double	64 bit floating point number	± 4.9E-324 to ± 1.7976931348623157E+308
boolean	true or false	NA, note Java Booleans cannot be converted to or from other types
char	16 bit, Unicode	Unicode character, \u0000 to \uFFFF Can mix with integer types

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword.

Data types in Java details

Data Type	Memory Size	Default value	Declaration
Byte	8 bits	0	byte a=9;
Short	16 bits	0	short b=89;
Int	32 bits	0	int c=8789;
Long	64 bits	0	long=9878688;
Float	32 bits	0.0f	float b=89.8f;
Double	64 bits	0.0	double c =87.098
Char	16 bits	'u0000'	char a ='e';
Boolean	JVM Dependent	false	boolean a =true;

Let us now look into detail about the eight primitive data types.

Byte

- Byte data type is an 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0

- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100 , byte b = -50

short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example: short s = 10000, short r = -20000

int:

- int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example: int a = 100000, int b = -200000

long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example: long a = 100000L, int b = -200000L

float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.

- Float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f

double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

3.2.2.2 Reference Data Types

Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Student, String etc.

- Class objects, and various type of array variables come under reference data type.

- Default value of any reference variable is **null**.

- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example: Student s1 = new Student("Amit");

2.2.3 Identifier

An Identifier is used to identify the program element. They are used for naming package, class, interface, method, or variable. Java Identifiers follow the following rules:

1. Use only the characters 'a' through 'z', 'A' through 'Z', '0' through '9', character '_', and character '$'.
2. A name cannot include the space character.
3. Do not begin with a digit.
4. A name can be of any realistic length.
5. Upper and lower case count as different characters.
6. A name cannot be a reserved word (keyword).
7. A name must not previously be in utilized in this block of the program.

Naming Conventions

Naming conventions specify the rules to be followed by a Java programmer while writing the names of packages, classes, methods etc.

- Package names are written in small letters.

  e.g.: java.io, java.lang, java.awt etc

- Each word of class name and interface name starts with a capital

  e.g.: Sample, AddTwoNumbers

- Method names start with small letters then each word start with a capital

  e.g.: sum (), sumTwoNumbers (), minValue ()

- Variable names also follow the same above method rule

  e.g.: sum, count, totalCount

- Constants should be written using all capital letters

  e.g.: PI, COUNT

- Keywords are reserved words and are written in small letters.

e.g.: int, short, float, public, void

2.2.4 Literals

Literals in Java are a sequence of characters (digits, letters and other characters) that represent constant values to be stored in variables. Literal is a programming term that essentially means that *what you type is what you get*.

OR

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Java language specifies five major types of literals. They are:

- Character Literals
- String Literals } Non Numeric Literals
- Boolean Literals
- Integer Literals } Numeric Literals
- Float Literals

Each of them has a type associated with it.

Note: Three reserved identifiers are used as predefined literals:
true and **false** representing boolean values.
null representing the null reference.

2.2.3.1 Character Literals

Character literals have the primitive data type character. A character is quoted in single quote (').

Note: Characters in Java are represented by the 16-bit Unicode character set.

For example:

‘A’
‘7’
‘\’

Backslash character Literals:
A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler. Java support some special character literals which are given in following table.

Escape Sequence Literal	Description
‘\b’	Back space
‘\t’	Tab
‘\n’	New line
‘\\’	Backslash
‘\”’	Single coute
‘\””’	Double coute

2.2.3.2 String Literals

A string literal is a sequence of characters which has to be double-quoted (") and occur on a single line.

Examples of string literals:

"false or true"
"result = 0.01"
"a"
"Java is a Object Oriented Programming Language"

String literals are objects of the class String, so all string literals have the type String.

2.2.3.3 Boolean Literals

As mentioned before, true and false are reserved literals representing the truth-values **true** and **false** respectively. Boolean literals fall under the primitive data type: **boolean**.

2.2.3.4 Integer Literals:

Integer data types consist of the following primitive data types: int,long, byte, and short.
int is the default data type of an integer literal.

An Integer constant refers to a series of digits. There are three types of integer as follows:

a) Decimal integer: Embedded spaces, commas and characters are not allowed in between digits. For example:
23 411
7,00,000
17.33

b) Octal integer
It allows us any sequence of numbers or digits from 0 to 7 with leading 0 and it is called as Octal integer. For example:
011
00
0425

c) Hexadecimal integer
It allows the sequence which is preceded by 0X or 0x and it also allows alphabets from 'A' to 'F' or 'a' to 'f' ('A' to 'F' stands for the numbers '10' to '15') it is called as Hexadecimal integer.
For example:
0x7
00X
0A2B

An integer literal, let's say 3000, can be specified as long by appending the suffix L (orl) to the integer value: so 3000L (or 3000l) is interpreted as a long literal.

There is no suffix to specify short and byte literals directly; ~~3000S~~ , ~~3000B~~, ~~3000s~~,~~3000b~~

3.2.3.5 Floating-point Literals

Floating-point data consist of **float** and **double** types. The default data type of floating-point literals is **double**, but you can designate it explicitly by appending the D (or d) suffix. However, the suffix F (or f) is appended to designate the data type of a floating-point literal as float.

We can also specify a floating-point literal in scientific notation using Exponent (short E ore), for instance: the double literal 0.0314E2 is interpreted as 0.0314 *10^2 (i.e 3.14).

Examples of double literals:

0.0 0.0D 0d
0.7 7D .7d
9.0 9 . 9D
6.3E-2 6.3E-2D 63e-1

Examples of float literals:
0.0f 0f 7F .7f
9.0f 9.F 9f
6.3E-2f 6.3E-2F 63e-1f

Note: The decimal point and the exponent are both optional and at least one digit must be specified.

2.2.4 Operator

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Misc Operators

Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

The Relational Operators:

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit	(A | B) will give 61 which is 0011 1101

	if it exists in either operand.	
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right	A << 2 will give 240 which is 1111 0000

	operand.	
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up	A >>>2 will give 15 which is 0000 1111 00111100

	with zeros.	

The Logical Operators:

The following table lists the logical operators:
Assume Boolean variables A holds true and variable B holds false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

The Assignment Operators:

There are following assignment operators supported by Java language:

Operator	Description	Example
=	Simple assignment operator, Assigns	C = A + B will assign value of A + B into C

	values from right side operands to left side operand	
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C – A
*=	Multiply AND assignment	C *= A is equivalent to C = C * A

	operator, It multiplies right operand with the left operand and assign the result to left operand	
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to	C %= A is equivalent to C = C % A

	left operand	
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Misc Operators

There are few other operators supported by Java Language.

Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Following is the example:

```
public class Test {

   public static void main(String args[]){
      int a , b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

This would produce the following result:

```
Value of b is : 30
Value of b is : 20
```

instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). **instanceof** operator is written as:

```
( Object reference variable ) instanceof  (class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the example:

```
public class Test {

  public static void main(String args[]){
    String name = "James";
    // following will return true since name is type of String
    boolean result = name instanceof String;
    System.out.println( result );
  }
}
```

This would produce the following result:

```
true
```

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more example:

```
class Vehicle {}

public class Car extends Vehicle {
  public static void main(String args[]){
    Vehicle a = new Car();
    boolean result =  a instanceof Car;
    System.out.println( result );
  }
}
```

This would produce the following result:

```
true
```

2.2.5 Separator:

Separators are symbols. It shows the separated code. They describe function of our code.

Name	Use
()	Parameter in method definition, containing statements for conditions,etc.
{}	It is used for define a code for method and classes
[]	It is used for declaration of array
;	It is used to show the separate statement
,	It is used to show the separation in identifier in variable declararion
.	It is used to show the separate package name from subpackages and classes, separate variable and method from reference variable.

2.2.6 Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	**Operator**	**Associativity**
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right

Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	

2.2.7 Type Conversion and Casting:

If you have previous programming experience, then you already know that 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.

Int i = 10

Long l = i

i = l

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 in-compatible types. To do so, you must use a cast, which performs **an explicit conversion** between incompatible types.

Let's look at both automatic type conversions and casting.

Java's Automatic Conversions

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.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you 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**. This kind of conversion is sometimes called a **narrowing conversion**, since you are explicitly making the value narrower so that it will fit into the target type.

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. For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a;
byte b;
// ...
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be **reduced modulo the target type's range**.

The following program demonstrates some **type conversions that require casts:**

```
// 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);
}
}
```

This program generates the following **output:**

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
```

d and b 323.142 67

Let's look at each conversion. When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case.

When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

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.

The Type Promotion Rules

Java defines several type promotion rules that apply to expressions.

They are as follows:

First, all *byte*, *short*, and *char* values are promoted to *int*, as just described. Then, if one operand is a *long*, the whole expression is promoted to *long*. If one operand is a *float*, the entire expression is promoted to *float*. If any of the operands is *double*, the result is *double*.

The following program demonstrates how each value in the expression gets promoted

to match the second argument to each binary operator:

```
class Promote {
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;

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

System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));

System.out.println("result = " + result);

}

}
```

Let's look closely at the type promotions that occur in this line from the program:

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

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 double.

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.

Conclusion of Type Conversion:

- Size Direction of Data Type
  - Widening Type Conversion (Casting down)
    - Smaller Data Type → Larger Data Type
  - Narrowing Type Conversion (Casting up)
    - Larger Data Type → Smaller Data Type
- Conversion done in two ways
  - Implicit type conversion

- Carried out by compiler automatically

– Explicit type conversion

- Carried out by programmer using casting

- Widening Type Conversion

– Implicit conversion by compiler automatically

```
byte -> short, int, long, float, double
short -> int, long, float, double
char -> int, long, float, double
int -> long, float, double
long -> float, double
float -> double
```

- Narrowing Type Conversion

– Programmer should describe the conversion explicitly

```
byte -> char
short -> byte, char
char -> byte, short
int -> byte, short, char
long -> byte, short, char, int
float -> byte, short, char, int, long
double -> byte, short, char, int, long, float
```

- byte and short are always promoted to int
- if one operand is long, the whole expression is promoted to long
- if one operand is float, the entire expression is promoted to float
- if any operand is double, the result is double
- General form: (targetType) value
- Examples:
- 1) integer value will be reduced module bytes range:

```
int i;
byte b = (byte) i;
```

- 2) floating-point value will be truncated to integer value:

```
float f;
int i = (int) f;
```