

1. Java Features:-

(1) Java Is Small and Simple

- Java is modeled after C and C++. The object-oriented approach, and most of Java's syntax, is adapted from C++.
- Programmers who are familiar with that language (or with C) will have a much easier time learning Java because of the common features.

(2) Java Is Object Oriented

- Object-oriented programming OOP organizes a program as a set of components called objects.
- These objects exist independently of each other, and they have rules for communicating with other objects and for telling those objects to do things.
- Java inherits its object-oriented concepts from C++ The language includes a set of class libraries that provide basic variable types, system input and output capabilities, and other functions. It also includes classes to support networking, Internet protocols, and graphical user interface functions.

(3) Java Is Secure

- Java is a much more secure language. Java provides security on several different levels. First, the language was designed to make it extremely difficult to execute damaging code. Another level of security is the bytecode verifier.
- Java programs are compiled into sets of instructions called bytecodes. Before a Java program is run, a verifier checks each bytecode to make sure that nothing suspicious is going on. In addition to these measures, Java has several safeguards that apply to applets

(4) Java Is Platform Independent

1. Platform independence is the ability of the same program to work on different operating systems
2. Java is completely platform independent. Java's variable types have the same size across all Java development platforms. Also, a Java .class file of bytecode instructions can execute on any platform without alteration.

(5) Java Is Compiled & Interpreted

1. Before you can run a program written in the Java language, the program must be compiled by the Java compiler. The compilation results in a "byte-code" file that, while similar to a machine-code file, can be executed under any operating system that has a Java interpreter.
2. This interpreter reads in the byte-code file and translates the byte-code commands into machine-language commands that can be directly executed by the machine that's running the Java program. You could say, then, that Java is both a compiled and interpreted language.

(6) Java Is Multi-threaded

1. Java programs can contain multiple threads of execution, which enables programs to handle several tasks concurrently. For example, a multi-threaded program can render an image on the screen in one thread while continuing to accept keyboard input from the user in the main thread.
2. All applications have at least one thread, which represents the program's main path of execution.

(7) Java Is Garbage Collected

1. Java programs do their own garbage collection, which means that programs are not required to delete objects that they allocate in memory.
2. This relieves programmers of virtually all memory-management problems.

(8) Java Is Robust

1. Because the Java interpreter checks all system access performed within a program, Java programs cannot crash the system.
2. Instead, when a serious error is discovered, Java programs create an exception. This exception can be captured and managed by the program without any risk of bringing down the system.

(9) High Performance

1. Java is "high performance" because its bytecode is efficient and has multithreading built in for applications that need to perform multiple concurrent activities.

(10) Dynamic

1. Java is dynamic, so your applications are adaptable to changing environments because Java's architecture allows you to dynamically load classes at runtime from anywhere on the network, which means that you can add functionality to existing applications by simply linking in new classes

2 **Compare Java with C/C++ ?**

	JAVA	C
Type of language	Java is object oriented programming language	C is structured programming language
struct/union	Java does not contain the data types struct & union	C contain the data types struct & union
Pointer	Java does not support explicit pointer type.	C support the concept of pointers .
Preprocessor	Java does not have a preprocessor & therefore we cannot use #define, #include, & #ifdef statements.	C has the preprocessor directives.
Header files	There are no header files to java	There are header files in c++.
Function without argument	Java requires that the functions with no arguments must be declared with empty parenthesis and not with the void keyword as done in c.	C uses the void keyword to specify function without argument .
New operator	Java adds new operators such as instanceof and >>>	C does not support such operators instanceof & >>>
Keywords	Java does not include the following c++ keywords delete, friend, inline, mutable, template, using virtual.	C++ uses all mentioned keywords.

	break /continue	Java adds labeled break & continue	C use simple break & continue.
		JAVA	C++
	Operator Overloading	Java does not support operator overloading.	C++ support operator overloading.
	Template Classes	Java does not have template classes	C++ has the concept of <u>template classes</u> Global
	variables	Java does not support global variables . every variable & method is declared within a class & form part of that class.	C++ has the concept of global variables.
	Pointer	Java does not support pointers	C++ uses the concept of pointers .
	Destructor	Java has replaced the destructors function with a finalize() function.	C++ uses the destructor to destroy objects.
	Header files	There are no header files to java	There are header files in C++.
	Multithreading	Java supports the concept of multithreading	C++ does not have multithreading concept
	Garbage collection	Java support automatic garbage collector & makes a lot of programming problems simply vanish .	C++ does not have automatic garbage collector
	Keywords	Java does not include the following c++ keywords delete,friend,inline, mutable,template,using virtual.	C++ uses all mentioned keywords.

3 Explain Data Types in java.

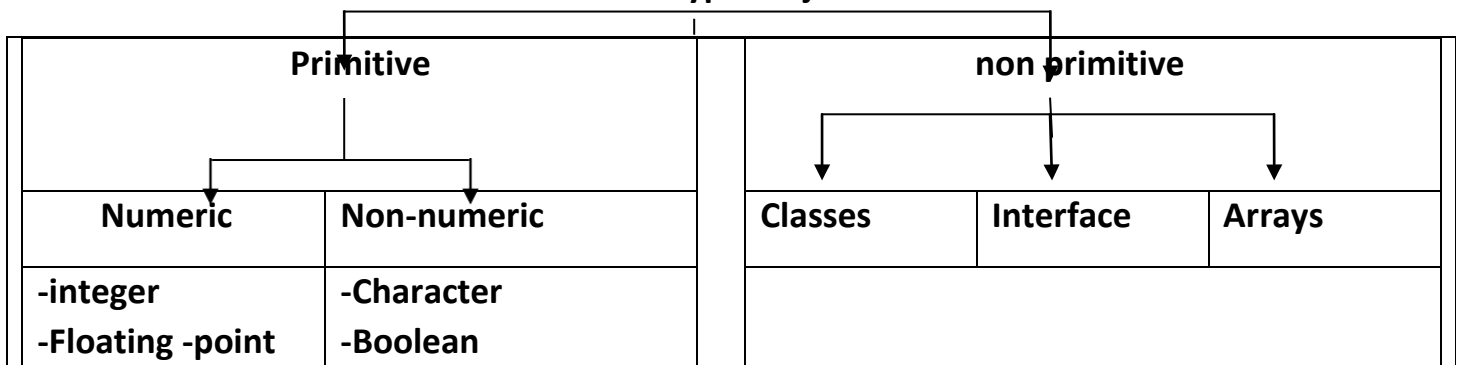
Data Types:-

Java defines eight simple (or elemental) types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. These can be put in four groups:

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

Data types in java under various categories. Basic 2 type primitive & non primitive

Data types in java



1)Integer Data Types

Java define four integer types: byte, short, int, and long. Java does not support the concept of unsigned types and therefore all java values are signed meaning they can be positive or negative . Each of these types takes up a different amount of space in memoryThe width and ranges of these integer types vary widely, as shown in this table:

Length	Data Type	Range
8 bits	byte	-128 to +127
16 bits	short	-32768 to+32767
32 bits	int	-2147483648 to +2147483647
64 bits	long	-9223372036854775808 to +9223372036854775807

byte:-

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127. 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.

Byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

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, since it is defined as having its high byte first (called big-endian format). This type is mostly applicable to 16-bit computers, which are becoming increasingly scarce.

Here are some examples of **short** variable declarations:

```
short s;
```

```
short t;
```

“Endianness” describes how multibyte data types, such as **short**, **int**, and **long**, are stored in memory. If it takes 2 bytes to represent a **short**, then which one comes first, the most significant or the least significant? To say that a machine is big-endian, means that the most significant byte is first, followed by the least significant one. Machines such as the SPARC and PowerPC are big-endian, while the Intel x86 series is little-endian.

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. Any time you have an integer expression involving **bytes**, **shorts**, **ints**, and literal numbers, the entire expression is promoted to **int** before the calculation is done.

The **int** type is the most versatile and efficient type, and it should be used most of the time when you want to create a number for counting or indexing arrays or doing integer math. It may seem that using **short** or **byte** will save space, but there is no guarantee that Java won't promote those types to **int** internally anyway. Remember, type determines behavior, not size. (The only exception is arrays, where **byte** is guaranteed to use only one byte per array element, **short** will use two bytes, and **int** will use four.)

long:-

long is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed.

We make integers long by appending the letter L & l at the end of the number.

2) Floating-Point Data 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 transcendental such as sine and cosine, result in a value whose precision requires a floating-point type.

Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Length	Data Type	Range
32 bits	float	3.4e-038 to 3.4e+038
64 bits	double	1.7e-308 to -1.7e+308

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.

3) Character Data Type

The character data type (char) is used to store single Unicode characters. Because the Unicode character set is composed of 16-bit values, the char data type is stored as a 16-bit unsigned integer

4) booleans:-

Java has a simple 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, such as **a < b**. **boolean** is also the type required by the conditional expressions that govern the control statements such as **if** and **for**.

4 Explain JVM in brief ?

Bytecode :-

- A special machine language that can be understood by the Java Virtual Machine (JVM)
- independent of any particular computer hardware, so any computer with a Java interpreter can execute the compiled Java program, no matter what type of computer the program was compiled on .

Java Virtual Machine (JVM):-

- All languages compiler translate source code into machine code for a specific computer. Java compiler also does the same thing. Then how does java achieve architecture neutrality? The answer is that the java compiler produces an intermediate code known as bytecode for machine does not exist. this so called virtual machine is known as the java virtual machine (JVM) & it exists only inside the computer memory. The machine language for the java virtual machine is called java bytecode.
- The process of compiling a java program into bytecode which is also referred to as virtual machine code is not machine specific. The machine specific code generated by the java interpreter by acting as intermediary between the virtual machine & the real machine as given in following fig. the interpreter is different for different machine.

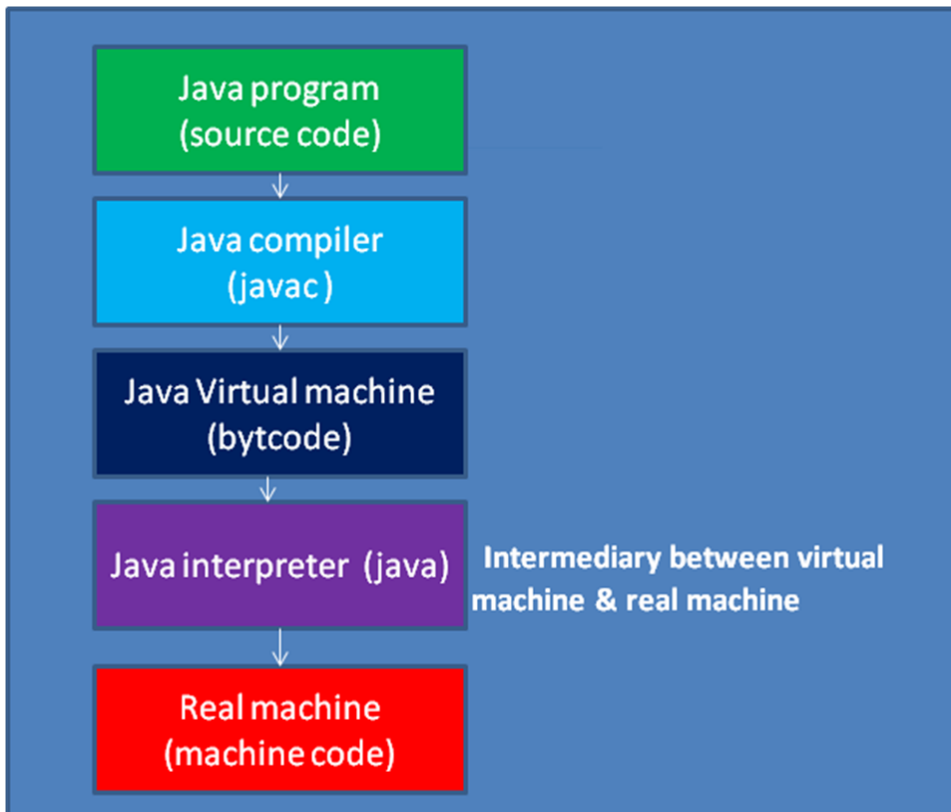


Fig:- process of compilation & converting bytecode into machine code

- Java bytecode run on any computer that has an interpreter this is one of the essential features of java: the same compiled program can be run on many different types of computer.

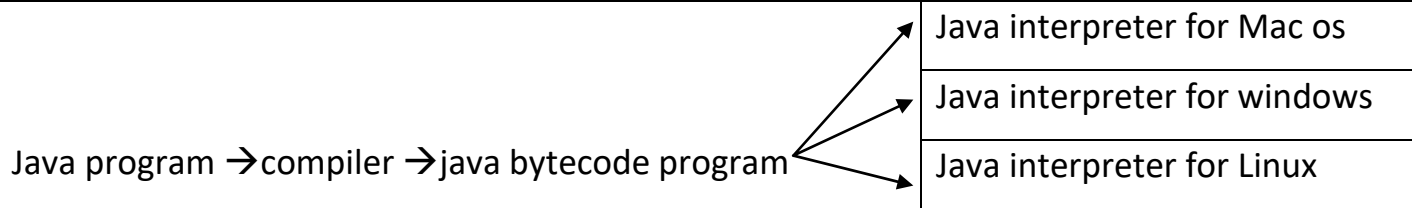


Fig: java compiler & interpreter

Note :only for reference

A Java Virtual Machine (JVM) is a set of computer software programs and data structures that use a virtual machine model for the execution of other computer programs and scripts. The model used by a JVM accepts a form of computer intermediate language commonly referred to as Java bytecode. This language conceptually represents the instruction set of a stack-oriented, capability architecture. As of 2006, there are an estimated four billion JVM-enabled devices worldwide.

Overview:

Java Virtual Machines operate on Java bytecode, which is normally (but not necessarily) generated from Java source code; a JVM can also be used to implement programming languages other than Java. For example, a source code can be compiled to Java bytecode, which may then be executed by a JVM. JVMs can also be released by other companies besides Sun (the developer of Java) — JVMs using the "Java" trademark may be developed by other companies as long as they adhere to the JVM specification published by Sun (and related contractual obligations).

The JVM is a crucial component of the Java Platform. Because JVMs are available for many hardware and software platforms, Java can be both middleware and a platform in its own right — hence the trademark *write once, run anywhere*. The use of the same bytecode for all platforms allows Java to be described as "compile once, run anywhere", as opposed to "write once, compile anywhere", which describes cross-platform compiled languages. The JVM also enables such features as Automated Exception Handling that provides 'root-cause' debugging information for every software error (exception) independent of the source code.

The JVM is distributed along with a set of standard class libraries that implement the Java API (Application Programming Interface). An application programming interface is what a computer system, library or application provides in order to allow data exchange between them. They are bundled together as the Java Runtime Environment.

Execution environment:-

Programs intended to run on a JVM must be compiled into a standardized portable binary format, which typically comes in the form of .class files. A program may consist of many classes in different files. For easier distribution of large programs, multiple class files may be packaged together in a .jar file (short for Java archive).

The JVM runtime executes .class or .jar files, emulating the JVM instruction set by interpreting it, or using a just-in-time compiler (JIT) such as Sun's Hotspot. JIT compiling, not interpreting, is used in most JVMs today to achieve greater speed. Ahead-of-time compilers that enable the developer to precompile class files into native code for a particular platforms also exist.

Like most virtual machines, the Java Virtual Machine has a stack-based architecture akin to a microcontroller/microprocessor.

The JVM, which is the instance of the *JRE (Java Runtime Environment)*, comes into action when a Java program is executed. When execution is complete, this instance is garbage-collected. JIT is the part of the JVM that is used to speed up the execution time. JIT compiles parts of the byte code that

have similar functionality at the same time, and hence reduces the amount of time needed for compilation.

Bytecode verifier:-

-tests the format of the code fragments and checks the code fragments for illegal code that can violate access rights to object

JIT:

Although Java was designed for interpretation, there is technically nothing about Java that prevents on-the-fly compilation of bytecode into native code. Along these lines, Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the Java 2 release. When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not possible to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. Instead, the JIT compiles code as it is needed, during execution. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the run-time system (which performs the compilation) still is in charge of the execution environment. Whether your Java program is actually interpreted in the traditional way or compiled on-the-fly, its functionality is the same.

5 Explain how java is architectural neutral?

The java designers worked hard in attaining their goal “write once run anywhere, anytime, forever” and as a result the java virtual machine (JVM) was developed. A main issue for java designers was that code longevity & portability. One of the main problems is the execution speed of the program. Since java is architectural neutral it generates bytecode that resembles machine code, and are not specific to any processor. Fig. focuses on this point more elaborately. Java bytecode (.class files) which can be interpreted by java virtual machine (JVM). Once the JVM is loaded on your computer (irrespective of operating systems), you can execute java program.

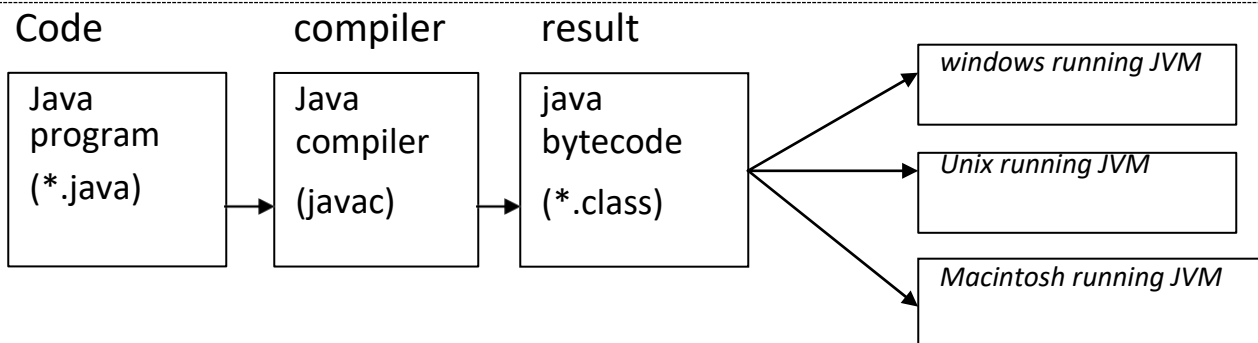


Fig: java is architectural – neutral & portable language

8 Explain **Type Conversion** and **Type Casting** in Java with examples ?

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. 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 conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a conversion between incompatible 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, the numeric types are not compatible with **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**, or **long**.

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.

```

/* Program to demonstrate casting Incompatible Types */

class Casting

{public static void main(String args[])

{byte b; int

i = 258;

double d = 340.142;

System.out.println("\n Conversion of int to byte ..... ");

b = (byte) i;

System.out.println(" integer i : "+i+" is converted to byte b : "+b);

System.out.println("\n Conversion of double to int .....");

i = (int) d;

System.out.println(" double d : "+d+" is converted to integer i : "+i);

System.out.println("\n Conversion of double to byte .....");

b = (byte) d;

System.out.println(" double d : "+d+" is converted to byte b : "+b);

}

```

Output:

Conversion of int to byte.....

integer i :258 is converted to byte b :2

Conversion of double to int.....

double d :340.142 is converted to integer i : 340

Conversion of double to byte.....

double d :340.142 is converted to byte b: 84

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** or **short** operand to **int** when evaluating an expression. This means that the sub expression **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:-

In addition to the elevation of **bytes** and **shorts** to **int**, Java defines several type promotion rules that apply to expressions. They are as follows. First, all **byte** and **short** 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.

Only For Reference :

TYPE CAST OPERATOR:

situation where there is need to store a value of one type into a variable of another type in such situations we must cast the value to be. stored by proceeding it with the type name in parentheses.

The syntax :

type variable =(type) variable 2;

the process of converting one data type to another is casting.

example :

```
int m =50;
```

```
byte n=(byte)m; long
```

```
count =(long)m;
```

casting is often necessary when a method returns a type different than the one we require.

-for integer type can be cast any other type except boolean.casting into smaller type may result in a loss of data .

- similarly the float & double type can be cast any other type except boolean.

-casting a floating point value to an integer will result in a loss of fractional part .

following table list those casts which are guaranteed to result in no loss of information.

from	to
byte	short ,char,int,long,float,double
short	int,long,float,double
char	int,long,float,double
int	long ,float, double
long	float, double
float	double

Automatic Conversion

for some types, it is possible to assign a value of one type to a variable of type without a cast.java does the conversion of the assigned value automatically .this is known as automatic type conversion. automatic type conversion is possible only if the destination type has enough precision to store the source value.

e.g

```
byte b=75;
```

```
int a=b;
```

Primitive Type Casting

Casting between primitive types enables you to convert the value of one data from one type to another primitive type. Commonly occurs between numeric types.(integer , floating point)There is one primitive data type that we cannot do casting though, and that is the boolean data type.

Types of Casting:

Implicit Casting

Explicit Casting

Implicit Casting

C++ permits explicit type conversion of variable or expression using the Type cast operator

Suppose we want to store a value of int data type to a variable of data type double.

```
int numInt = 10;
```

```
double numDouble = numInt; //implicit cast
```

In this example, since the destination variable's data type (double) holds a larger value than the value's data type (int),

the data is implicitly casted to the destination variable's data type double. Example:

```
int numInt1 = 1;
```

```
int numInt2 = 2;
```

```
//result is implicitly casted to type double
```

```
double numDouble = numInt1/num
```

Explicit Casting

When we convert a data that has a large type to a smaller type, we must use an explicit cast.

Explicit casts take the following form:

(Type)value

where,

Type (target type)- is the name of the type you're converting to

value -is an expression that results in the value of the source type

Explain how java is platform independent.

(11) Java Is Platform Independent

1. *Platform independence* is the ability of the same program to work on different operating systems
2. Java is completely platform independent. Java's variable types have the same size across all Java development platforms. Also, a Java .class file of bytecode instructions can execute on any platform without alteration.