Topic Covered:
Literals and Escape Sequence, The Scope and Lifetime of Variables, Constants, Type conversion in Java, Widening or Automatic Type Conversion, Narrowing or Explicit Conversion, Type promotion in Expressions, Explicit type and casting in Expressions.

LITERALS
A constant value in Java is created by using a literal representation of it.

Integer Litrals:
byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well. Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example:
```
int decimal = 100;
int octal = 0144;
int hexa = 0x64;
```
Values of type long that exceed the range of int can be created from long literals by specifying L or l:
```
long var = 1234567654321L
```

Floating-Point Literals
A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d.
The floating point types (float and double) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).
```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1  = 123.4f;
```

Char literals
For char data types we can specify literals in 4 ways:
Single quote : We can specify literal to char data type as single character within single quote.
```
char ch = 'a';
```
Char literal as Integral literal : we can specify char literal as integral literal which represents Unicode value of the character and that integral literals can be specified either in Decimal, Octal and Hexadecimal forms. But the allowed range is 0 to 65535.
```
char ch = 062;
```
Unicode Representation : We can specify char literals in Unicode representation '\uxxxx'. Here xxxx represents 4 hexadecimal numbers.
```
char ch = '\u0061';// Here /u0061 represent a.
```
Escape Sequence : Every escape character can be specify as char literals.
```
char ch = '\n';
```

JAVA ESCAPE SEQUENCE

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a formfeed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\\\	Insert a backslash character in the text at this point.

EXERCISE 2-1:

Declare two short variable and try to assign their sum to byte variable and write down what happened.

```
short a=10, b=20;
byte = a+b;
```

Declare a long variable with value with int range then declare another long variable with value outside the range of int. Write down what happened.

Declare a float variable like this and write down what happened.

```
float  a = 12.54;
```

Declare all types of integers and floating point variables and try to give those values outside of their range and then write down the errors.

Create 5 character variable by giving it different hexadecimal codes and write down what characters are shown on screen.

Declare and initialize two character variable with any alphabet value. Then use this statement to print the result and write down what happened.

```
System.out.println (ch1 + ch2);
```

THE SCOPE AND LIFETIME OF VARIABLES

block defines a scope. A block is begun with an opening curly brace and ended by a closing curly brace 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.

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.

PROGRAM 1: Demonstrating 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);
}
}
```

PROGRAM 2: Demonstrating lifetime of a variable.

```
class LifeTime {
public static void main(String args[]) {
int x;
for(x = 0; x < 3; x++) {
int y = -1;         // y is initialized each time block is entered
System.out.println("y is: " + y);      // this always prints -1
y = 100;
System.out.println("y is now: " + y);
}
}
}
```

CONSTANTS

Sometimes you know the value of a data item, and you know that its value will not (and should not) change during program execution, nor is it likely to change from one execution of the program to another. In this case, it is a good software engineering practice to define that data item as a constant. Defining constants uses the same syntax as declaring variables, except that the data type is preceded by the keyword final.

```
final datatype CONSTANT_IDENTIFIER = VALUE;
```

TYPE CONVERSION IN JAVA

When you 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. For example, assigning an int value to a long variable.

WIDENING OR AUTOMATIC TYPE CONVERSION

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type.

VALID DATA TYPES FOR ASSIGNMENT

Data Type	Compatible Data Types
byte	Byte
short	byte, short
int	byte, short, int, char
long	byte, short, int, char, long
float	byte, short, int, char, long, float
double	byte, short, int, char, long, float, double
boolean	boolean
char	Char

PROGRAM 3: Demonstrating automatic 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);
    }
}
```

NARROWING OR EXPLICIT CONVERSION

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing. 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
```

PROGRAM 4: Java program to illustrate incompatible data type for explicit type conversion

```
public class Test {
  public static void main(String[] arg){
    char ch = 'c';
    int num = 88;
    ch = num;
  }
}
```

PROGRAM 5: Java program to illustrate explicit type conversion

```
class Test{
    public static void main(String[] args)    {
        double d = 100.04;

        //explicit type casting
        long l = (long)d;

        //explicit type casting
        int i = (int)l;
        System.out.println("Double value "+d);

        //fractional part lost
        System.out.println("Long value "+l);

        //fractional part lost
        System.out.println("Int value "+i);
    }
}
```

PROGRAM 6: 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);
    }
}
```

TYPE PROMOTION IN EXPRESSIONS

While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:

- Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
- If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

PROGRAM 7: 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);
    }
}
```

EXPLICIT TYPE CASTING IN EXPRESSIONS

While evaluating expressions, the result is automatically updated to larger data type of the operand. But if we store that result in any smaller data type it generates compile time error, due to which we need to type cast the result. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

In cases where you understand the consequences of overflow, you should use an explicit cast, such as:

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

EXERCISE 2-2:
Write down a program which has 4 int variables a, b, c and d with different values. Then store the result of equation "a\b + c\d" in a variable "ans" of float data type. The result should be accurate.