CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 2)**

Prepared by Renetha J.B.

Topics

- Literals
- Variables
- Type Conversion and Casting
- Arrays
- Strings
- Vector class.

Literals

- A **constant value** in Java is created by using a *literal* representation.
  - **Integer Literals**
  - **Floating-Point Literals**
  - **Boolean Literals**
  - **Character Literals**
  - **String Literals**

Integer Literals

- Any whole number value is an integer literal.
- Examples are 1, 2, 3, and 42
- There are three bases which can be used in integer literals
  - **Decimal(base 10)**
  - **octal (base 8)**
  - **hexadecimal (base 16).**

Integer Literals

- Normal decimal numbers
  - **cannot** have a leading zero.
  - can use digits from 0 to 9
- Octal values
  - are denoted by a leading zero.
  - can use digits from 0 to 7
  - E.g 012, 0356
- Hexadecimal constant
  - are denoted with a leading zero-x, (**0x or 0X).**
  - use digits from 0 to 9 and letters *A* through *F (or a through f)* **E.g. 0x234, 0X3B5c**

Integer Literals

- An integer literal can always be assigned to a **long variable.**
  - Append an upper- or lowercase *L to the literal*
    - 92233720368547758O7L
- integer can also be assigned to a **char** as long as it is within range.
- literal value is assigned to a **byte** or **short variable as** long as it is within range.

Floating-Point Literals

- Floating-point numbers represent ***decimal values with a fractional component.***

- **Standard notation** consists of a **whole number** component followed by a **decimal point** followed by a **fractional component.**
  - **E.g.** 3.14159, 2.0
- **Scientific notation** uses a **standard-notation floating-point number** plus a *suffix (that specifies a power of 10 by which the number is to be multiplied.)*
  - The exponent is indicated by an *E or e followed by a* decimal number, which can be positive or negative
  - **E.g.** 6.022E23, 314159E–05, 2e+100.

Floating-Point Literals

- Floating-point literals in Java are **double precision by default.**
- To specify a **float** literal, we must append an **F** or **f** to the constant.
- We can also explicitly specify a **double** literal by appending a **D** or **d**.
- The default **double** type consumes 64 bits of storage, while the less-accurate **float** type requires only 32 bits

Boolean Literals

- Boolean literals are simple.
- There are only two logical values that a boolean value can have,
  - **true** , **false**.
- The values of true and false do not convert into any numerical representation.
- The **true** literal in Java ***does not equal 1***
- The **false** literal in Java ***does not equal 0.***

Character Literals

- Characters in Java are i**ndices into the Unicode character set**.
- They are 16-bit values that can be converted into integers
  - and manipulated with the integer operators, such as the addition and subtraction operators.
- A literal character is represented inside a pair of single quotes.
  - All of the visible ASCII characters can be directly entered inside the quotes, such as *'a'*, *'z'*, *and* '@'.

Character Literals

- **'\n' for** the newline character.
- '\'' for the single-quote character
- For octal notation, use the backslash followed by the three-digit number.
  - For example, *'141' is the letter 'a'.*
- *For hexadecimal, you enter a backslash-u (**\u**), **then*** exactly four hexadecimal digits.
  - *\u0061'*

String Literals

- String literals in Java are specified like they are in most other languages—by **enclosing** a sequence of characters **between a pair of double quotes**
- Examples of string literals are
  - "Hello World"
  - "two\nlines"
  - "\"This is in quotes\""

Variables

- The variable is the **basic unit of storage** in a Java program.
- A variable is defined by
  - the combination of an **identifier**, a **type**, and an ***optional*** **initializer**.
- All variables have a **scope,**
  - which defines their **visibility**, and a **lifetime**.

Declaring a Variable

- All variables **must be declared** before they can be used.
- The basic form of a variable declaration is :

type ***identifier*** *[[* ***= value****][,* ***identifier*** *[***= value****] ...] ;*

- *The type is one of Java's atomic types, or the name of a class or interface.*
- The *identifier is the name of the* variable.
- Square bracket denote that =Value is optional in declaration.

Example- variable declaration

- **int a, b, c;** *// declares three int, a, b, and c.*
- **int d = 3, e, f = 5;** *// declares three int,*

  // initializes d to 3 and f to 5.
- **byte z = 22;** // initializes z to 22
- **double pi = 3.14159;** *// declares an approximation of pi.*
- **char x = 'x';** // the variable x has the value 'x'.

Dynamic Initialization

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

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
        double a = 3.0, b = 4.0;
        double c = Math.sqrt(a * a + b * b);
        // Here c is dynamically initialized
        System.out.println("Hypotenuse is " + c);
}
}
```

The Scope and Lifetime of Variables

- All of the variables used have been declared at the start of the main( ) method.
- Java allows variables to be declared within any block.
    - a block begins with an opening curly brace and ended by a closing curly brace.
  - A block defines a *scope*.
    - *A block begins with{ and end with }*
- A **scope** determines ***what objects are visible to other parts of your program.***
- **Scope** also determines the **lifetime** of those objects.

The Scope and Lifetime of Variables(contd.)

- Two major scopes are
  - Scope defined by a **class**
  - Scope defined by a **method**.
- Variables **declared inside a scope** are **not visible (that is, accessible) to code that is defined outside that scope**.
  - Local variable

The Scope and Lifetime of variables(contd.)

- Scopes can be nested.
  - Each time you create a block of code, we are creating a new, nested scope.
  - 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***.

```
{//outer
    {//inner
        {//innermost
        }
    }
}
```

```
class Sample {
public static void main(String args[])
{
    int x;          // known to all code within main function
    x = 10;
    if(x == 10)
        { // start new scope
        int y = 20; // known only to this block
                            // x(OUTER SCOPE) 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);
    }
}
```

The Scope and Lifetime of variables(contd.)

// This fragment is **wrong**!

count = 100; *// cannot use **variable** before it is declared!*

int count;

- Variables are **created** when their scope is entered, and **destroyed** when their scope is left.
  - This means that a variable **will not hold its value once it has gone out of scope.**

// Demonstrate lifetime of a variable.

- **Variable can be reinitialized** each time it enters the block in which it is declared

```
class LifeTime {
public static void main(String args[]) {
int x;
    for(x = 0; x < 2; 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);
    }
} }
```

OUTPUT

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

- Although blocks can be nested, you cannot **declare a variable to have the** same name as one in an **outer scope**.

```
// This program will not compile
class ScopeErr {
public static void main(String args[])
{ int bar = 1;
   {   // creates a new scope
   int bar = 2; // Compile-time error
                // bar already defined in outer scope!
   }
}
}
```

Type Conversion and Casting

- If the two types are **compatible**, then Java will perform the **conversion automatically(implicitly**).
  - it is always possible to assign an **int value to a long variable.**
- The conversion between incompatible types are to be done **explicitly.**

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.

Destination = source

*(**same** type or **larger**)*

- When these two conditions are met, a ***widening conversion*** *takes place.*

Java's Automatic Conversions(contd.)

- For **widening conversions**, the **numeric types, including integer and floating-point types**, are **compatible** with each other.
  - No automatic conversions from the **numeric types to char or boolean.**
- Java also performs an **automatic** type conversion when a literal integer constant is stored into variables of type **byte, short, long, or char.**

byte → short →int →long → float → double

WIDENING CONVERSION

SMALL----------→ LARGE

Casting Incompatible Types

- If we want to assign an **int value** to a **byte variable .**
  - This conversion will **not** be performed **automatically**, because a ***byte is smaller than an int.***

byte variable=integer

(small) ⟵ (large)

- This is called ***narrowing conversion.***
- To create a conversion between two **incompatible types,** we must use a **cast**.

Casting Incompatible Types(contd.)

- A **cast** *is* simply an **explicit** **type conversion**. It has this general form:

  (***target-type***) *value*

  – *target-type specifies the **desired type** to which value is to be converted.*

  int a;

  byte b;

  b = (**byte**) a; ***//Here integer value in variable a is casted(converted) to byte type***

- If the **integer's value** is **larger** than the range of a byte, it will be reduced to modulo *(the remainder of an integer division)* by the byte's range(256).

Casting Incompatible Types(contd.)

- A different type of conversion will occur when a **floating-point value** is assigned to an **integer** type: ***truncation.***
  - If the value 1.23 is assigned to an integer, the resulting value will simply be 1.
  - ***int a=1.23;*** *// here variable a stores only 1*

    // .23 will have been truncated

Casting Incompatible Types(contd.)

- 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**.

  E.g.

  ```
  byte b;
  int i = 257;
  b=(byte)i;
  ```

  Here byte(-128 to 127) is smaller than 257, so the value stored in b is

  257 mod 256=1

- When the large value is cast into a **byte variable**, the ***result*** is the remainder of the division of value by 256

byte → short →int →long → float → double

WIDENING CONVERSION

(AUTOMATIC / IMPLICIT)

SMALL----------→ LARGE

double→ float →long →int → short → byte

NARROWING CONVERSION

LARGE ----------→SMALL

EXPLICIT

Automatic Type Promotion in Expressions


```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c; // conversions may occur in expressions.
```

Here intermediate term **a * b** (40*50=2000) exceeds the range of its byte operands(-128 to 127) a and b.

- To handle this kind of problem, Java automatically promotes each **byte, short,** or **char** operand to **int** when evaluating an expression.
- So no error.
- Variable d will contain 20

Automatic promotion

byte b = 50;

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

- In expression **b*2**, automatic promotion occurs . i.e. result of b*2 (50*2=100) is **promoted to integer.**
- **This result(integer value) is larger than byte** type variable b where it is to be stored**.**
  - **So ERROR is shown.**

- To **solve** this issue, **explicit conversion** is needed for result.

byte b = 50;

b = (byte)(b * 2);

NO ERROR

The Type Promotion Rules

- First, all **byte, short, and char values** are promoted to **int**.
- 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**.

```
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);

}

}
```

f * b , b is promoted to a float (result float)

i / c, c is promoted to int, and the result is of type **int**.

d * s, the value of s is promoted to double – result double

float plus an **int** is a **float.**

float minus the **double** is promoted to **double**

RESULT double

Arrays

- An array is a group of **like-typed(same type) variables** that are referred to by **a common name**.
- Arrays of **any type** can be created
- Arrays may have one or more **dimensions**.
- A specific element in an array is accessed by its **index**.
  - Index means position It starts from 0.
    - Index of first element is 0, second element is 1 etc.

Arrays

- **One-Dimensional Arrays**
  - create an array variable of the desired type.
  - **Declaration** syntax 1

type variablename[ ];

E.g. int a[];

  - **Declaration** syntax 2

type[ ] variablename;

- The following two declarations are equivalent:

int a[];

int[]a;

Here this declaration means that **a** is an array variable, but no array actually exists. No space is allocated for it in memory

Arrays(contd.)

- We have to link array **with an actual, physical array of integers.**
- So we must allocate space using **new** and assign it to array variable .
  - new is a special operator that allocates memory.

  ***variable*=new *type*[*size*];**

  E.g.

  int a[];

  a= new int[12];

  int a[]=new int[12];

  - After this statement executes, variable **a** will refer to an array of 12 integers

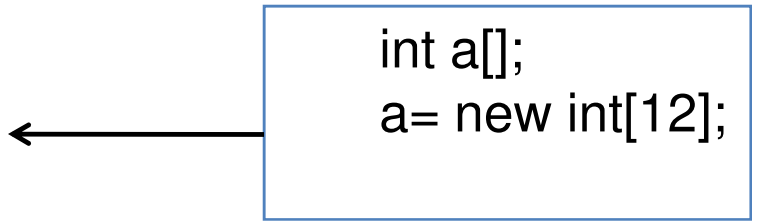
Array

- Obtaining an array is a two-step process.
  1. First, we must **declare** a variable of the desired array type.
  2. Second, we must **allocate the memory** that will hold the array, using **new**, and assign it to the array variable

- In Java all arrays are ***dynamically allocated.***
- It is possible to combine the declaration of the array variable with the allocation.

E.g.

```
int a= new int[12];
```



Store value in array

```
class Array {
public static void main(String args[])
    {
        int a[];
        a = new int[4];
        a[0] = 1;
        a[1] = 3;
        a[2] = 2;
        a[3]=5;
     }
     }
```

Array initilization

- Arrays can be initialized(give values) when they are declared.
- An **array initializer** is a list of ***comma-separated*** expressions surrounded by ***curly braces***.
- No need for **new** operator

```
class AutoArray {
    public static void main(String args[])
{
    int a[] = { 1,3,2,5};
}
}
```

Array(contd.)

- If you try to access elements **outside** the range of the array (negative numbers or numbers greater than the length of the array), it will cause a **run-time error**.
- E.g

```
int a[]=new int[10];
a[-3]=5;        //ERROR
a[11]=7;        //ERROR
```

ARRAY INDEX OUT OF BOUNDS

- // Average value in an array.

```
class Average {
public static void main(String args[])
{
      double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
      double result = 0;
      int i;
      for(i=0; i<5; i++)
          result = result + nums[i];
      System.out.println("Average is " + result / 5);
      }
}
```

Array(contd.)

int[] num1, nums2, nums3; // create three arrays

- creates three array variables num1,num2,num3 of type **int.**

- It is the same as writing

int num1[], nums2[], nums3[];

Multidimensional Arrays

- Multidimensional arrays are actually arrays of arrays.
- To declare a multidimensional array variable, specify each *additional index* using another set of **square brackets.**
- **E.g 2 D array** declaration

int b[][]= new int[4][5];

This allocates a 4 by 5 array and assigns it to variable **b.**

4 rows and 5 columns

Multidimensional Arrays(contd.)

- The following declarations are also equivalent:

```
char twod[][] = new char[3][4];
char[][] twod = new char[3][4];
```

Multidimensional Arrays

- When you allocate memory for a multidimensional array, you need only **specify the <u>memory for the first</u>** (leftmost) dimension.

```
int a[][] = new int[2][];

a[0] = new int[3];

a[1] = new int[3];
```

```
int a[][]= new int[2][3];
```

  - Here **a** is 2D array with two rows. First row **a[0]** has 3 columns. Second row **a[1]** has 3 columns.

```
class TwoDArray {
public static void main(String args[]) {
int a[][]= new int[2][3];
int i, j, k = 0;
for(i=0; i<2; i++)
    {
          for(j=0; j<3; j++)
          {
          a[i][j] = k;
          k++;
          }
    }
for(i=0; i<2; i++)
    { for(j=0; j<3; j++)
          {System.out.print(a[i][j] + " ");}
      System.out.println();
    }                    } }
```

```
OUTPUT
0 1 2
3 4 5
```

Array(cont.)

- When you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension.
- E.g.

  int a[][] = new int[2][];

  a[0] = new int[1];

  a[1] = new int[2];

  – Here array **a** has 2 rows.
  – First row a[0] has 1 column.
  – Second row a[0] has 1 column.

```
class TwoDAgain {
public static void main(String args[]) {
      int a[][] = new int[2][];
      a[0] = new int[1];
      a[1] = new int[2];
      int i, j, k = 0;
      for(i=0; i<2; i++)
      {
          for(j=0; j<i+1; j++)
          {a = k;
          k++;
      }
      for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
          {System.out.print(a[i][j] + " ");}
      System.out.println(); }
        } }
```

OUTPUT
0
1 2

Multidimensional array initialization

- Enclose each dimension's initializer(values) within its own set of curly braces.
- We can use **expressions** as well as **literal values** inside of array initializers.
- Eg.

```
int a[][]={ {1,2,3} , {3,4,5}} ;
```

```
class Matrix {
public static void main(String args[]) {
double m[][] = {
{ 0*0, 1*0, 2*0, 3*0 }, { 0*1, 1*1, 2*1, 3*1 },
{ 0*2, 1*2, 2*2, 3*2 }, { 0*3, 1*3, 2*3, 3*3 }
};
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<4; j++)
    {System.out.print(m[i][j] + " ");}
System.out.println();
}
}
}
```

OUTPUT

```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

String class

- String is a **class**.
- It can **defines an object.**
- The String type is used to **declare string variables**
- A quoted string constant(E.g. “hello”) can be assigned to a **String variable.**
- A variable of *type String* can be assigned to another variable of *type String*.
- We can use an object of type String as an argument to println( )
- E.g.

```
String str = "this is a test";
System.out.println(str);
```

Here, str is an object of type String.
It is assigned the string “this is a test”.
This string is displayed by the println( ) statement.

String E.g.

```
class Sample {
public static void main(String args[])
    {
    String s="Hello"
    System.out.print(s);
    }
}
```

OUTPUT
Hello

String(contd.)

- In Java, string is basically an **object** that represents sequence of char values .
- An array of characters works same as Java string.
- For example:

char[] ch={'H','e','l','l','o'};

String s=new String(ch);

//This statement converts character array ***ch*** *to string and store in string object s.*

This is same as

String s="Hello"; //creating string by java string literal

String methods

- length()
  - The length of a string can be found with the length() method.

```
class Sample {
    public static void main(String args[])
     {
     String s="Hello";
     System.out.print("Length=",s.length());
     }
}
```

OUTPUT
Length=5

String methods(contd.)

- toUpperCase() and toLowerCase()
  - To convert from lower to upper and upper to lower respectively

```
class Sample {
public static void main(String args[])
    {
    String s="Hello World";
    System.out.println(s. toUpperCase());
    System.out.println(s. toLowerCase());
    }
}
```

OUTPUT
HELLO WORLD
hello world

String methods(contd.)

- indexOf()
  - The indexOf() method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace)

```
class Sample {
public static void main(String args[])
    {
    String s="I am fine.I am ok";
    System.out.println(s.indexOf("am"));
    }
}
```

OUTPUT
2

String concatenation

- Method 1:The + operator can be used between strings to combine them. This is called concatenation
- Method 2:We can use **concat()** method to concatenate two strings.

```
class Sample {
public static void main(String args[])
    {
    String s1="Computer" , s2="Science"
    System.out.println(s1+s2);
    }
}
```

OUTPUT
ComputerScience

String concatenation(contd.)


```
class Sample {
public static void main(String args[])
    {
    String s1="Computer " , s2="Science";
    System.out.println(s1+s2);
    }
}
```

OUTPUT
Computer Science

String concatenation(contd.)

- If we add a number and a string, the result will be a string concatenation.

```
class Sample {
public static void main(String args[])
    {
    String s1="10 " , s2="12";
    int a=13;
    System.out.println(s1+s2);
    System.out.println(s1+a);

    }
}
```

OUTPUT
1012
1013

Vector class

- Vector implements a **dynamic array**.
  - it can **grow** or **shrink** in size as required.
- It is similar to ArrayList class, but with two differences:
  - Vector is synchronized, and it contains many legacy methods that are not part of the Collections Framework.
    - Synchronized **means** if one thread is working on **Vector**, no other thread can get a hold of it.
  - Vector can **extend AbstractList class** and can **implement** the **List interface.**

Vector(contd.)

- All vectors **start with an initial capacity(size).**
- After this initial capacity is reached, the next time that you attempt to store an object in the vector, the vector automatically allocates space for that object **plus extra room for additional objects**.
- The amount of extra space allocated during each reallocation is determined by the ***increment*** that you specify when you create the vector.
- If we don't specify an ***increment***, **the vector's size is doubled** by each allocation cycle.

Vector(contd.)

- Vector is declared like this:

class Vector<E>

- Here, E specifies the type of element that will be stored.
- **Vector constructors** are

Vector( )

Vector(int *size)*

Vector(int *size, int incr)*

Vector(Collection<? extends E> *c)*

Vector(contd.)

- **Vector**( ) creates a **default vector**, which has an initial size of 10.
- **Vector(int *size)*** creates a vector whose initial capacity is specified by *size*.
- **Vector(int *size, int incr)*** creates a vector whose minitial capacity is specified by *size and whose increment is specified by incr.*
  - The **increment** specifies the number of elements to allocate each time that a vector is resized upward.
- **Vector(Collection<? extends E> *c)*** *c*reates a vector that contains the elements of collection *c*.

Vector(contd.)

- Vector defines these protected data members:

int **capacityIncrement;**

int **elementCount**;

Object[ ] **elementData**;

- The increment value is stored in capacityIncrement.
- The number of elements currently in the vector is stored in elementCount.
- The array that holds the vector is stored in elementData.

Vector(contd.)

- Vector defines several legacy methods

Method	Description
void addElement(E *element*)	The object specified by *element* is added to the vector.
int capacity( )	Returns the capacity of the vector.
Object clone( )	Returns a duplicate of the invoking vector.
boolean contains(Object *element*)	Returns **true** if *element* is contained by the vector, and returns **false** if it is not.
void copyInto(Object *array*[ ])	The elements contained in the invoking vector are copied into the array specified by *array*.
E elementAt(int *index*)	Returns the element at the location specified by *index*.
Enumeration<E> elements( )	Returns an enumeration of the elements in the vector.
void ensureCapacity(int *size*)	Sets the minimum capacity of the vector to *size*.
E firstElement( )	Returns the first element in the vector.
int indexOf(Object *element*)	Returns the index of the first occurrence of *element*. If the object is not in the vector, –1 is returned.
int indexOf(Object *element*, int *start*)	Returns the index of the first occurrence of *element* at or after *start*. If the object is not in that portion of the vector, –1 is returned.
void insertElementAt(E *element*, int *index*)	Adds *element* to the vector at the location specified by *index*.
boolean isEmpty( )	Returns **true** if the vector is empty, and returns **false** if it contains one or more elements.
E lastElement( )	Returns the last element in the vector.

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

The Scope and Lifetime of Variables(contd.)

- The **scope defined by a method** begins with its opening curly brace. {
  - If that method has **parameters**, they ***too are included within the method's scope.***