Object Oriented Programming Methodology

Module 3 : Arrays, Strings and Vectors

Faculty In-charge:
Pragya Gupta
E-mail: pragya.g@somaiya.edu

Contents

- Arrays

Arrays

- Array is a memory space allocated, which can store multiple values of same data type, in contiguous locations
- This memory space, which can be perceived to have many logical contiguous locations, can be accessed with a common name
- A specific element in an array is accessed by the use of a subscript or index used in brackets, along with the name of the array
  - For example, marks[5] would mean marks of 5th student
- While the complete set of values is called an array, the individual values are known as elements
- Arrays can be two types:
  - one dimensional array
  - multi-dimensional array

1-D Arrays

- In one-dimensional array, a single subscript or index is used, where each index value refers to individual array element
- The indexation will start from 0 and will go up to n-1, i.e. the first value of the array will have an index of 0 and the last value will have an index of n–1, where ŋ is the number of elements in the array
- So, if an array named marks has been declared to store the marks of 5 students, the computer reserves five contiguous locations in the memory, as shown below

1-D Arrays

60	58	50	78	89
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

```
Marks[0] = 60;
Marks[1] = 58;
Marks[2] = 50;
Marks[3] = 78;
Marks[4] = 89;
```

Creation of Arrays

- Creating an array, similar to an object creation, can inherently involve three steps:
  - Declaring an array
  - Creating memory locations
  - Initializing/assigning values to an array

Declaring an Array

- Declaring an array is same as declaring a normal variable except that you must use a set of square brackets with the variable type
- There can be two ways in which an array can be declared:
  - type arrayname[ ];
  - type[ ] arrayname;
- For e.g.
  - int marks[ ]; or
  - int[ ] marks;

Creating memory locations

- An array is more complex than a normal variable, so we have to assign memory to the array when we declare it
- You assign memory to an array by specifying its size
- Interestingly, our same old *new* operator helps in doing the job, just as shown below:
  - Arrayname = new type [size];
- So, allocating space and size for the array named as **marks** can be done as,
  - marks = new int[5];

Initializing/assigning values to an array

- Assignment of values to an array, which can also be termed as initialization of array, can be done as follows,
  - Arrayname[index] = value;
- The creation of list of marks to be assigned in array, named as *marks* has already been shown in the section above

Setting values in an array

```
public class Array {
	public static void main(String[] args) {
	int[] marks = new int[5];
	marks[0] = 60;
	marks[1] = 58;
	marks[2] = 50;
	marks[3] = 78;
	marks[4] = 89;
	}	}
```

Alternatively you can also use:

```
int marks[] = {60, 58, 50, 78, 89}
```

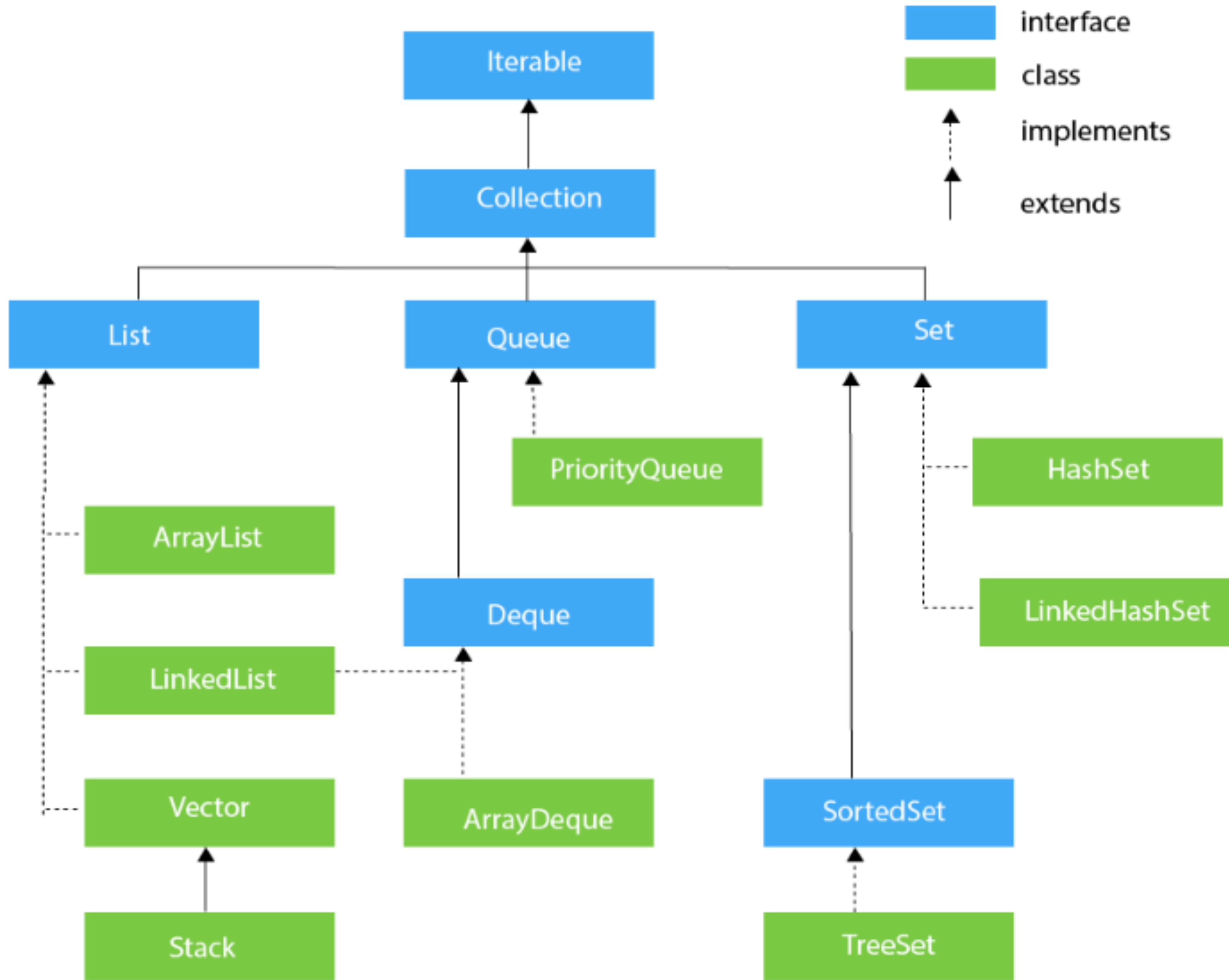
Using for-each with Arrays

- The format of for-each is as follows:
  - for (type var : arr){
  - // Body of loop
  - }
- For example, to calculate the sum of the elements of the array, for-each can be used as follows.
  - int[] arr= {2,3,4,5,6};
  - int sum = 0;
  - for(int a : arr)
  - // a gets successively each value in arr
  - {
  - sum += a;
  - }
- The disadvantage of for-each approach is that it is possible to iterate in forward direction only by single steps

Multidimensional Arrays/ Jagged Arrays

Collection Framework

- Provides an architecture to store and manipulate a group of objects
- Achieve all the operations that you perform on a data such as **searching, sorting, insertion, manipulation, and deletion**
- Java Collection means a single unit of objects
- Java Collection framework provides many **interfaces** (Set, List, Queue, Deque) and **classes** (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet)



interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Iterator interface

- Iterator interface provides the facility of iterating the elements in forward direction only

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Collection Interface

- Interface which is implemented by all the classes in the collection framework
- It declares the methods that every collection will have
- Collection interface builds the foundation on which the collection framework depends
- Some of the methods of Collection interface are
  - Boolean add (Object obj)
  - Boolean addAll (Collection c)
  - void clear(), etc.

  which are implemented by all the subclasses of Collection interface

List Interface

- List interface is the child interface of Collection interface
- It inhibits a list type data structure in which we can store the ordered collection of objects
- **It can have duplicate values**
- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
- To instantiate the List interface, we must use :

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

ArrayList

- Java **ArrayList** class uses a **dynamic array for storing the elements**(there is *no size limit)*
- We can add or remove elements anytime, much more flexible than the traditional array
- It is found in the ***java.util*** **package**
- The ArrayList in Java can have the **duplicate elements** also
- It **implements the List interface** so we can use all the methods of List interface here.
- The ArrayList maintains the insertion order internally

LinkedList

- Java LinkedList class uses a doubly linked list to store the elements
- It provides a linked-list data structure

Method	Description
addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list
removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list
getFirst()	Get the item at the beginning of the list
getLast()	Get the item at the end of the list

ArrayList v/s LinkedList

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Vectors

- Vector implements a **dynamic array of objects**
- Vector proves to be very useful **if you don't know the size of the array in advance** or you just need one that **can change sizes o**ver the lifetime of a program
- Vector can contain heterogeneous objects
- We **cannot store elements of primitive data type;** first it need to be converted to objects. A vector can store any objects
- Its defined in ***java.util*** package and class member of the Java Collections Framework

Vectors

- Vector implements List Interface
- A vector has an initial capacity, if this capacity is reached then size of vector automatically increases
- This default initial capacity of vectors are 10
- Each vector tries to optimize storage management by maintaining a *capacity* and a *capacityIncrement* arguments
- To traverse elements of a vector class we use **Enumeration** interface

Vector Methods

void **addElement**(Object *element*)	The object specified by *element* is added to the vector
int **capacity()**	Returns the capacity of the vector
boolean **contains(**Object *element*)	Returns **true** if **element** is contained by the vector, else **false**
void **copyInto**(Object *array[]*)	The elements contained in the invoking vector are copied into the array specified by **array[]**
elementAt(int *index*)	Returns the element at the location specified by *index*
Object **firstElement().**	Returns the first element in the vector

Vector Methods

void **insertElementAt**(Object *element*, int *index*)	*Adds element* to the vector at the location specified by *index*
boolean **isEmpty()**	Returns **true** if Vector is empty, **else false**
Object **lastElement()**	Returns the last element in the vector
void **removeAllElements()**	Empties the vector. After this method executes, the size of vector is zero.
void **removeElementAt**(int *index*)	Removes element at the location specified by *index*
void **setElementAt**(Object *element*, int *index*)	The location specified by *index* is assigned *element*

Vector Methods

void **setSize**(int *size*)	*Sets the number of elements in the vector to **size**. If the new size is less than the old size, elements are lost. If the new size is larger than the old, null elements are added*
int **size()**	Returns the number of elements currently in the vector

Access Protection

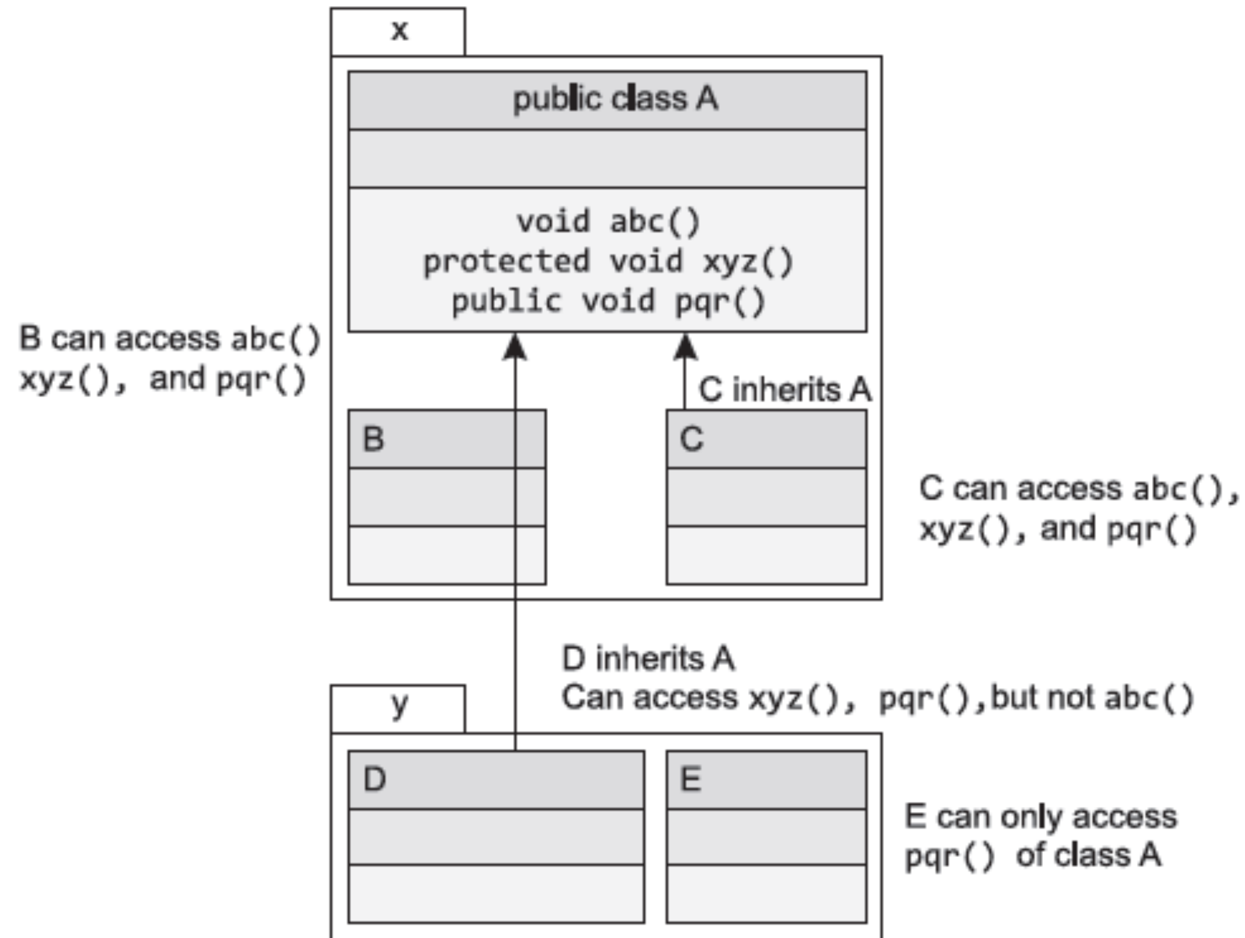
- Four Access specifier in Java

public(maximum access)	applied to variables,contructors, methods, and classes
protected	applied to variables,contructors, methods, and inner classes (not top- level classes)
default	applied to variables,contructors,methods, and classes
private(minimum access)	can be applied to variables, methods and inner classes (not top-level classes)

Access Specifiers

- public means accessibility for all
- private means accessibility from within the class only
- default (blank) access specifiers are accessible only from within the package
- protected means accessibility outside the packages but only to subclasses

Access Specifiers



Access Protection

- method abc() is accessible from A, B as well as C, but neither from D nor E
- protected method are accessible outside the package also, but only to subclasses outside the package. For example, the method xyz() accessible from classes A, B, C, D, but not from E
- public method pqr() is accessible from all classes A, B, C, D and E

toString() Method

- If you want to represent any object as a string, **toString() method** comes into existence
- The toString() method returns the String representation of the object
- By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code

Java Wrapper Classes

- For each primitive type, there is a corresponding *wrapper* class designed
- Are wrapper around primitive data types
- Allow for situations where primitives cannot be used but their corresponding objects are required
- Normally Used to convert a numeric value to a String or vice-versa
- Just like String, Wrapper objects are also immutable
- All the wrapper classes except Character and Float have two constructors—one that takes the primitive value and another that takes the String representation of the value
- Character has one constructor and float has three

Java Wrapper Classes

Primitive	Wrapper
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

Java Wrapper Classes

- For each primitive type, there is a corresponding *wrapper* class designed
- Are wrapper around primitive data types
- Allow for situations where primitives cannot be used but their corresponding objects are required
- Normally Used to convert a numeric value to a String or vice-versa
- Just like String, Wrapper objects are also immutable
- All the wrapper classes except Character and Float have two constructors—one that takes the primitive value and another that takes the String representation of the value
- Character has one constructor and float has three

Java Wrapper Classes

- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
- The classes in java.util package handles only objects and hence wrapper classes help in this case also.
- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading

Java Wrapper Classes

- Converts primitive to wrapper
  - double a = 4.3;
  - Double wrp = new Double(a);
- Each wrapper provides a method to return the primitive value.
  - double r = wrp.doubleValue();

Autoboxing and Unboxing

- **Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing
- **Unboxing:** It Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing

String Class

- Are basically immutable objects in Java
- Immutable means once created the, strings cannot be changed
- Whenever we create strings, it is this class that is instantiated
- In Java strings can be instantiated in two ways:
  - String x= “String Literal Object”;
  - String y=new String (“String object is created here”);

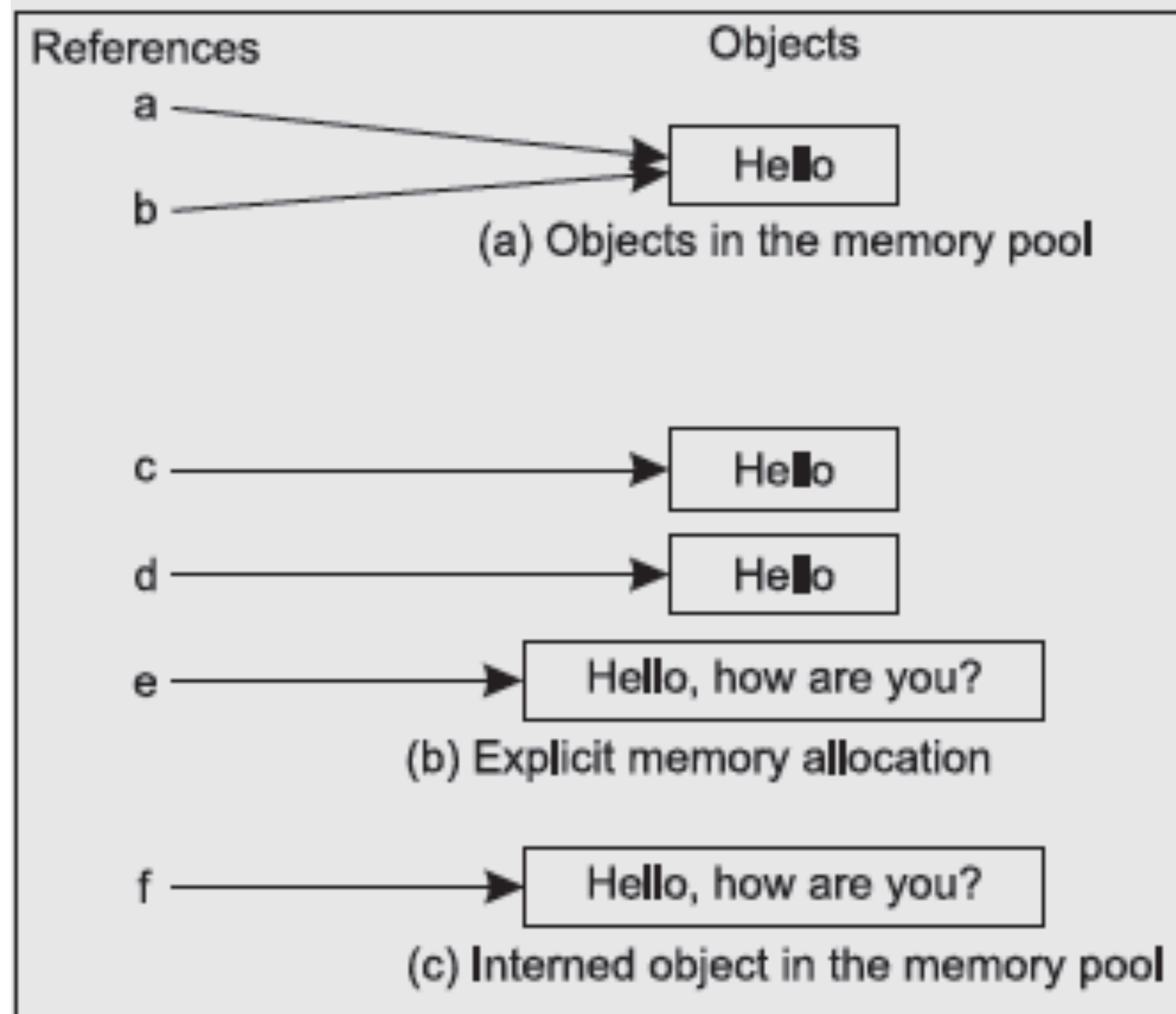
String Example

```
String a=”Hello”;                String b=”Hello”;
String c=new String(“Hello”);
String d=new String(“Hello”);
String e=new String(“Hello, how are you?”);
if(a==b)
        System.out.println(“object is same and is being shared by a &    b”);
else
        System.out.println(“Different objects”);
if(a==c)
        System.out.println(“object is same and is being shared by a &    c”);
else
        System.out.println(“Different objects”);
```

String Example

```
if(c==d)
        System.out.println("same object");
else
        System.out.println("Different objects");
String f=e.intern();
if(f==a)
System.out.println("Interned object f refer to the already created object a in the pool");
else
System.out.println("Interned object does not refer to the already created objects, as literal was not present in the pool. It is a new object which has been created in the pool");
```

String Example



String Manipulation

- Strings in Java are immutable (read only) in nature, Once defined cannot be changed.
- Let us take an example:
  - String x = “Hello”; // ok
  - String x = x +”World”; // ok, but how?
- the ‘+’ operator concatenates If at least one of the operand is a string.
- The second statement gets converted to the following statement automatically
  - String x=new StringBuffer(). append(x). append(“World”). toString();

Common Methods of String Class

Method Name with Signature	Method Details
int length()	to find length of the string (Line 1, Example 6.8)
boolean equals(String str)	Used to check equality of String objects. In contrast to == operator, the check is performed character by character. If all the characters in both the Strings are same, true is returned else false. (Line 2, Example 6.8)
int comparetTo(String s)	Used to find whether the invoking String (Figure 6.2) is Greater than, less than or equal to the String argument. It returns an integer value. If the integer value is a) < than zero – invoking string is less than String Argument b) > than Zero – invoking String is greater than String Argument c) = to Zero – invoking String and String argument are Equal (Line 3 – 9, Example 6.8)
boolean regionMatches(int startingIndx, String str,int strStartingIndx,int numChars)	Matches a specific region of String with specific region of the invoking String. The argument details : **startingIndx** – specifies the region from the invoking String to be matched. **str** – is the second string to be matched **strStartingIndx** – specifes the region from str to be matched with invoking String. **numChars** – specifies the number of character to be matched in both strings from their respective starting indexes. (Line 10, Example 6.8)

Common Methods of String Class

int indexOf(char c)	to find the index of a character in the invoking String object. (Line 11, Example 6.8)
int indexOf(String s)	Overloaded method to find the starting index of a String argument in the invoking String object. (Line 12, Example 6.8)
int lastIndexOf(char c)	to find the last occurrence of character in the invoking String. (Line 13, Example 6.8)
int lastIndexOf(String s)	Overloaded method to find the last occurrence of String argument in the invoking String object. (Line 14, Example 6.8)
String substring(int sIndex)	to extract the String from the invoking String Object starting with sIndex till the End of the String. (Line 15, Example 6.8)
String substring(int startingIndex, int endingIndex)	Overloaded method to extract the String starting with startingIndex till the endingIndex from the invoking String Object String. (Line 16, Example 6.8)
int charAt(int pos)	to find the character at a particular position(pos). (Line 17, Example 6.8)
String toUpperCase()	to change the case of entire String to Capital letters. (Line 18)
String toLowerCase()	to change the case of entire String to small letters. (Line 19)
boolean startsWith(String ss)	to find whether invoking String starts with String argument (Line 20)
boolean endsWith(String es)	to find whether invoking String ends with String argument (Line 21)
Static String valueOf(int is)	Converts primitive type int value to String. (Line 22)
Static String valueOf(float f)	Overloaded static method to Convert Primitive type float value to String.
Static String valueOf(long l)	Overloaded static method to Convert Primitive type long value to String.
Static String valueOf(double d)	Overloaded static method to Convert Primitive type double value to String.

StringBuffer Class

- StringBuffer class is used for representing changing strings
- StringBuffer offers more performance enhancement whenever we change Strings, because it is this class that is actually used behind the curtain
- Just like any other buffer, StringBuffer also has a capacity and if the capacity is exceeded, then it is automatically made larger
- The initial capacity of StringBuffer can be known by using a method capacity()

Methods of StringBuffer Class

Method Name with Signature	Method Details
int capacity()	Returns the current capacity of the storage available for character in the Buffer. (Line 2). When the capacity is approached the capacity is automatically increased. (Line 6)
StringBuffer append(String str)	appends String argument to the Buffer. (Line 3)
StringBuffer replace(int sindx,int eIndx,String str)	The characters from start to end are removed and str is inserted at that position (Line 4)
StringBuffer reverse()	Reverses the buffer character by character (Line 5)
Char charAt(int index)	Returns the character at specified index (Line 7)
Void setCharAt(int indx,char c)	Sets the specified character at specified index (Line 8)

StringBuilder Class

- A substitute of StringBuffer class.
- This class is faster than StringBuffer class, as it is not synchronized
- append(), insert(), delete(), deleteCharAt(), replace(), and reverse() return StringBuilder objects rather than StringBuffer objects
- The line creates a StringBuilder object
  - StringBuilder s=new StringBuilder();
  - construct a StringBuilder object with an initial capacity of 16 characters. Similar to that of StringBuffer