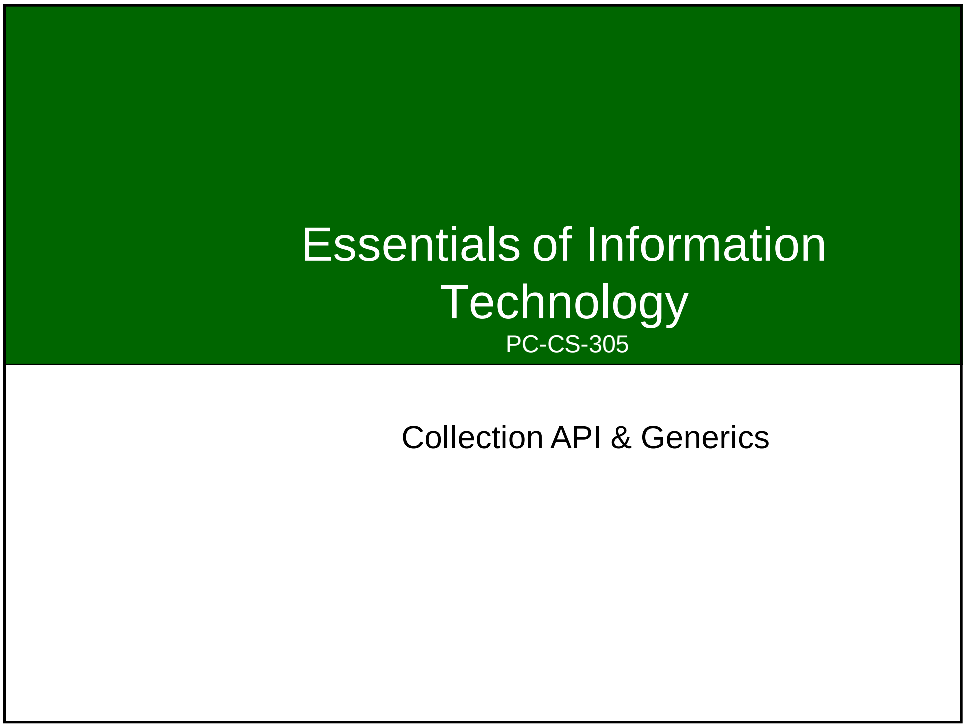
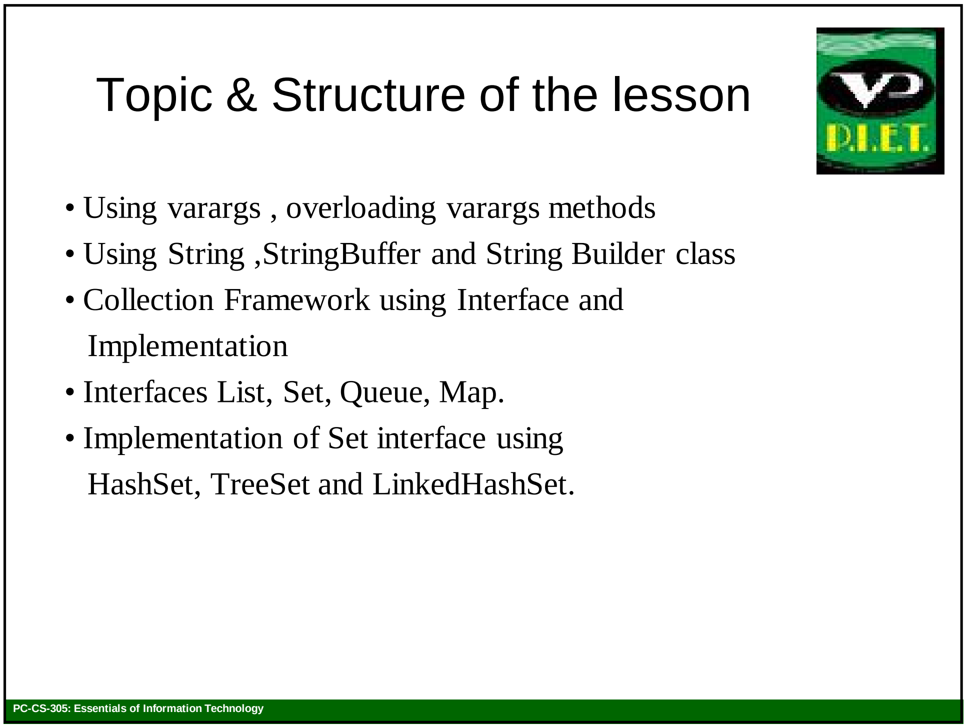


Essentials of Information Technology

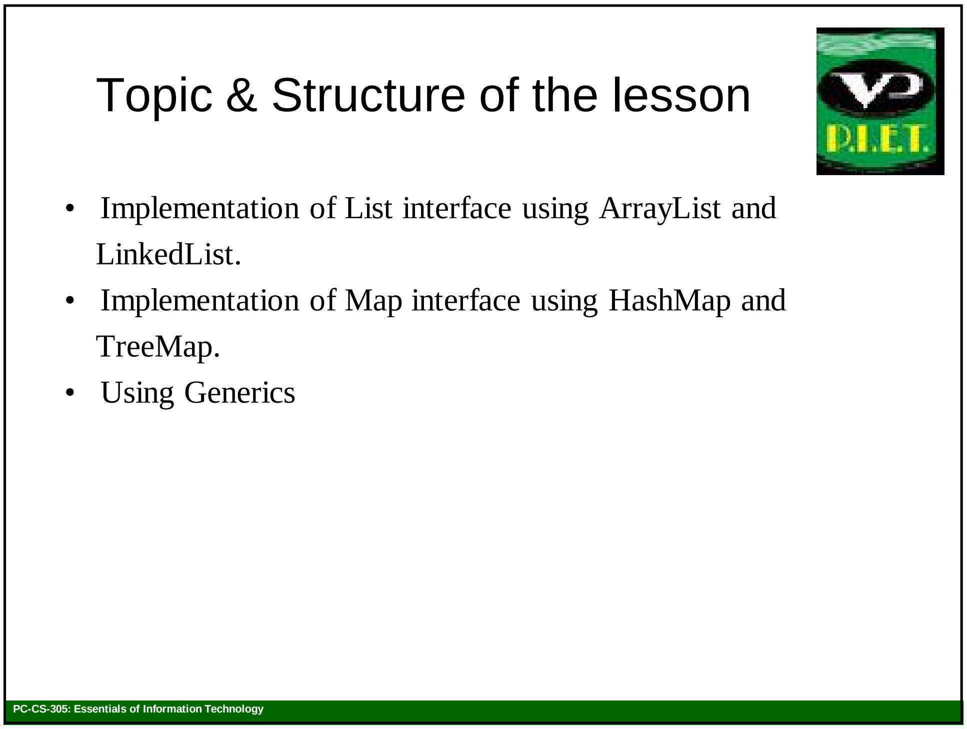
PC-CS-305

Collection API & Generics



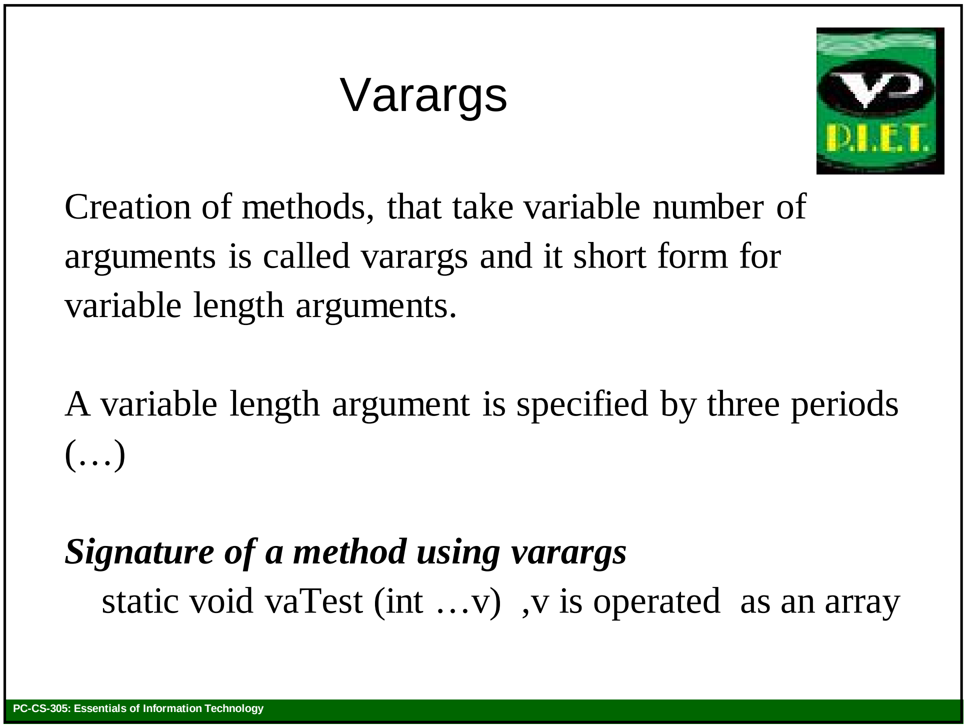
Topic & Structure of the lesson

- Using varargs , overloading varargs methods
- Using String ,StringBuffer and String Builder class
- Collection Framework using Interface and Implementation
- Interfaces List, Set, Queue, Map.
- Implementation of Set interface using HashSet, TreeSet and LinkedHashSet.



Topic & Structure of the lesson

- Implementation of List interface using ArrayList and LinkedList.
- Implementation of Map interface using HashMap and TreeMap.
- Using Generics



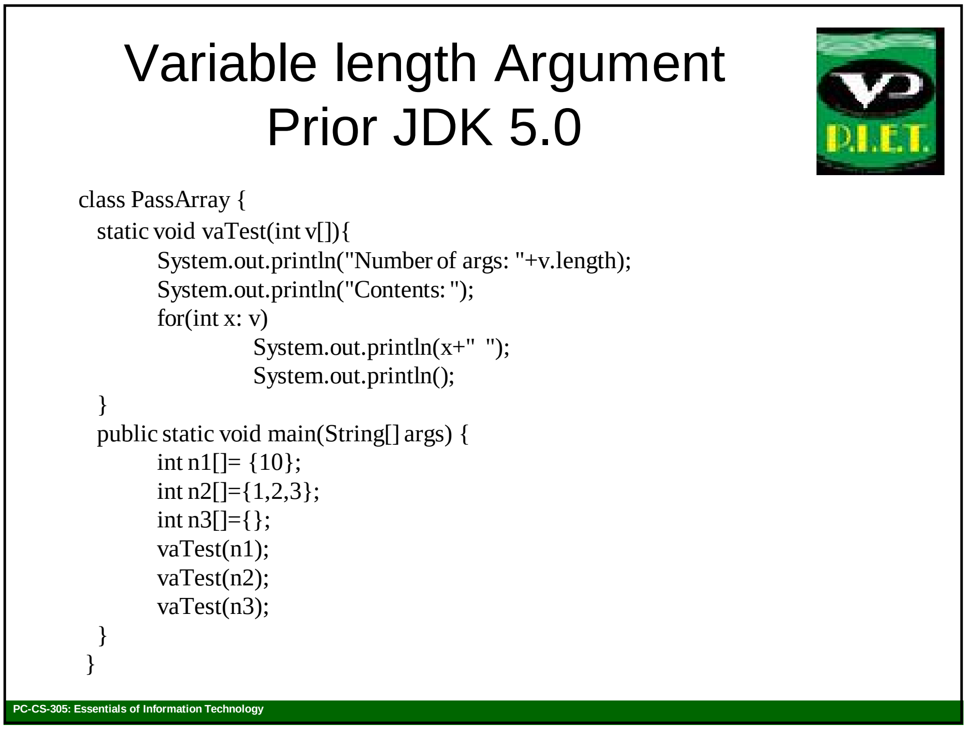
Varargs

Creation of methods, that take variable number of arguments is called varargs and it short form for variable length arguments.

A variable length argument is specified by three periods (…)

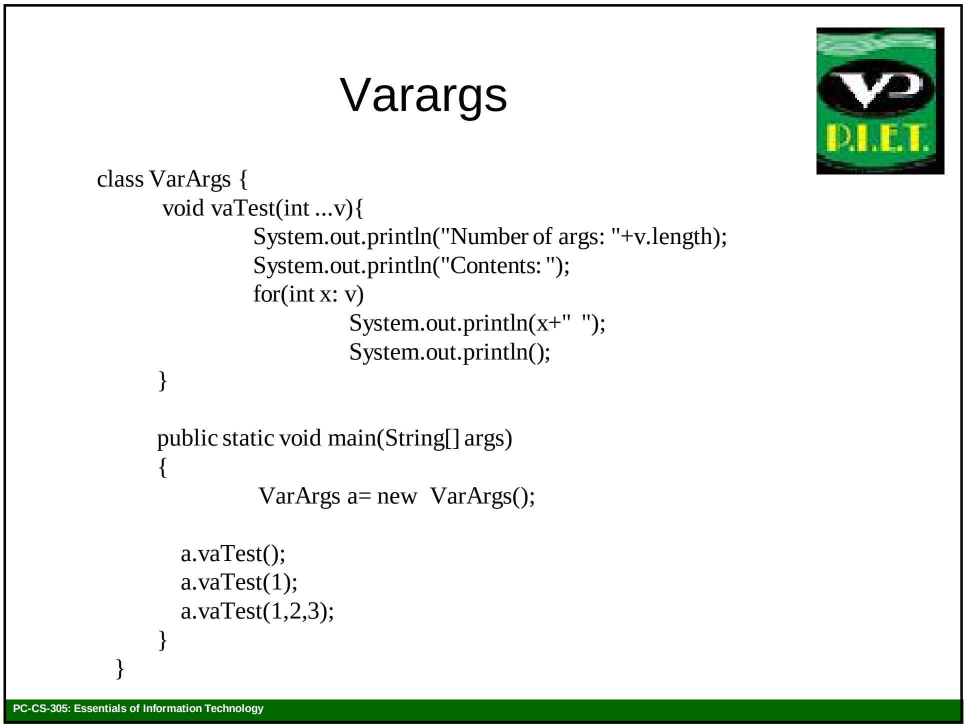
Signature of a method using varargs

static void vaTest (int …v) ,v is operated as an array



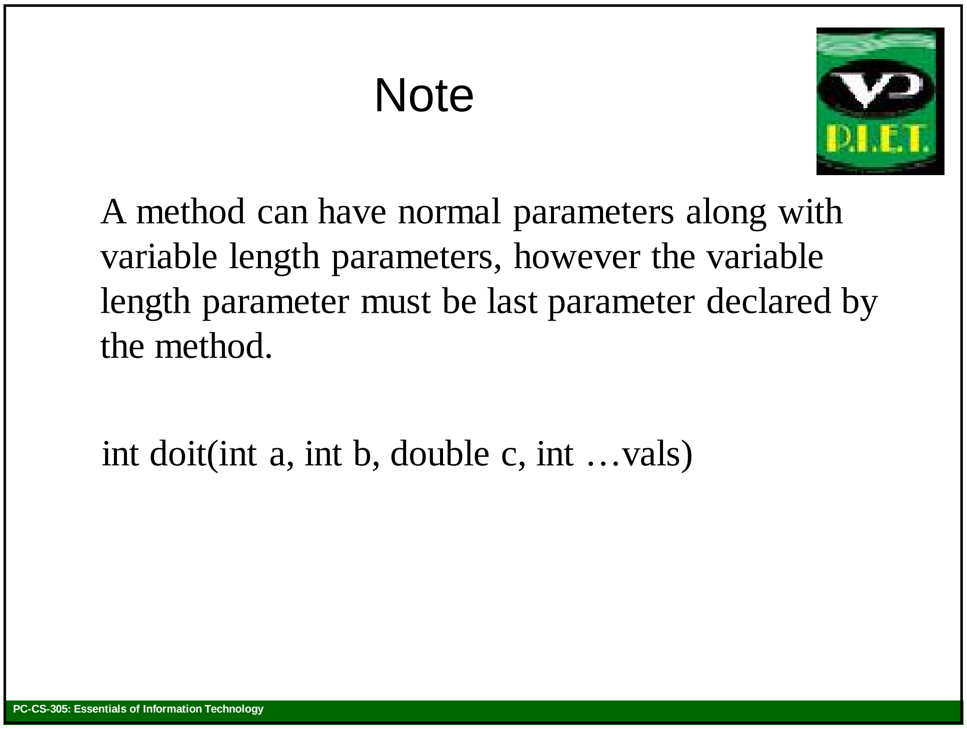
Variable length Argument Prior JDK 5.0

```
class PassArray {
  static void vaTest(int v[]){
          System.out.println("Number of args: "+v.length);
          System.out.println("Contents: ");
          for(int x: v)
                   System.out.println(x+"  ");
                   System.out.println();
  }
  public static void main(String[] args) {
          int n1[]= {10};
          int n2[]={1,2,3};
          int n3[]={};
          vaTest(n1);
          vaTest(n2);
          vaTest(n3);
  }
 }
```



Varargs

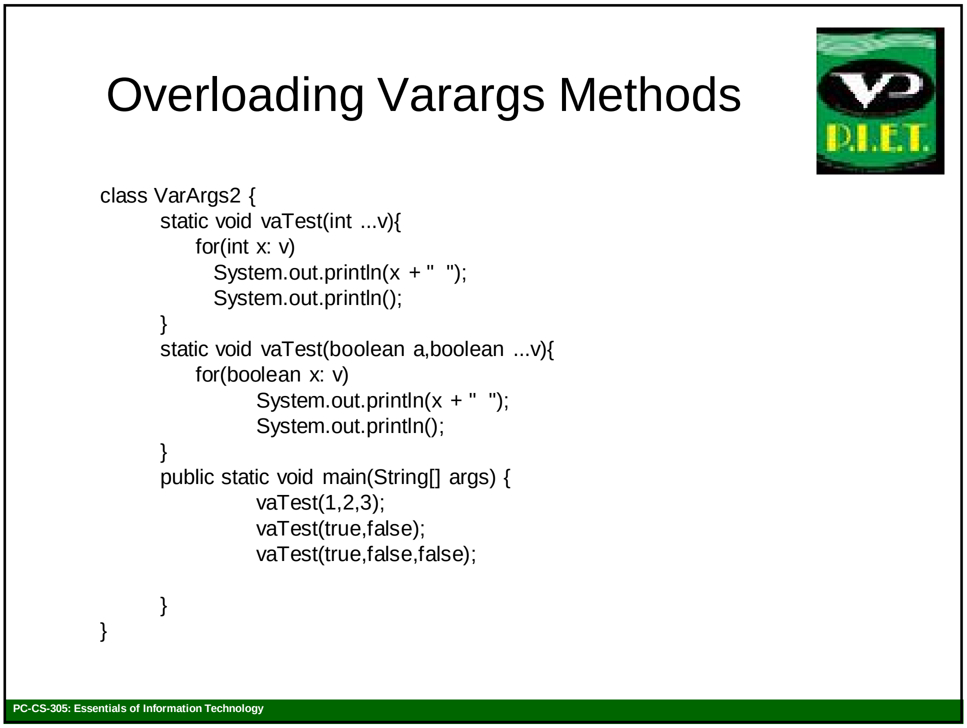
```
class VarArgs {
        void vaTest(int ...v){
                 System.out.println("Number of args: "+v.length);
                 System.out.println("Contents: ");
                 for(int x: v)
                          System.out.println(x+"  ");
                          System.out.println();
       }

       public static void main(String[] args)
       {
                 VarArgs a= new  VarArgs();

          a.vaTest();
          a.vaTest(1);
          a.vaTest(1,2,3);
       }
  }
```



Note

A method can have normal parameters along with variable length parameters, however the variable length parameter must be last parameter declared by the method.

int doit(int a, int b, double c, int …vals)



Overloading Varargs Methods

```
class VarArgs2 {
        static void vaTest(int ...v){
             for(int x: v)
                System.out.println(x + "  ");
                System.out.println();
        }
        static void vaTest(boolean a,boolean ...v){
             for(boolean x: v)
                     System.out.println(x + "  ");
                     System.out.println();
        }
        public static void main(String[] args) {
                     vaTest(1,2,3);
                     vaTest(true,false);
                     vaTest(true,false,false);

        }
}
```

5 Golden Rules

1. Primitive Widening > Boxing > Varargs.

2. Widening and Boxing (WB) not allowed.

3. Boxing and Widening (BW) allowed.

http://www.coderanch.com/t/417622/Programmer-Certification-SCJP/certification/Golden-Rules-widening-boxing-varargs

5 Golden Rules

4. While overloading, Widening + vararg **and** Boxing + vararg can only be used in a mutually exclusive manner i.e. not together.

5. Widening between wrapper classes not allowed

Examples

Overloaded methods	Invoked by saying	Called method
doX(Integer i) & doX(long l)	doX(5)	long (by Rule 1)
doX(int...i) & doX(Integer i)	doX(5)	Integer (by Rule 1)
doX(Long l) & doX(int...i)	doX(5)	int...i (Rule 2 & 1)
doX(Long l) & doX(Integer...i)	doX(5)	Integer...i(R. 2&1)
doX(Object o) & doX(Long l)	doX(5)	Object o (Rule 2&3)
doX(Object o) & doX(int...i)	doX(5)	Object o (Rule 3&1)
doX(Object o) & doX(long l)	doX(5)	long l (Rule 3&1)
doX(long...l) & doX(Integer...i)	doX(5)	ambiguous (Rule 4)
doX(long...l) & doX(Integer i)	doX(5)	Integer (Rule 1)

doX(Long l)	Integer i;
	doX(i)
		error (Rule 5)

doX(Long l) & doX(long...l)	Integer i;
	doX(i)
		long...l(Rule 5 & 1)

Collection API

- A *collection* — is simply an object that groups multiple elements into a single unit.
- They can dynamically grow and shrink, which is their advantage over arrays.

Collection Framework

All collection framework contain the following

Interface

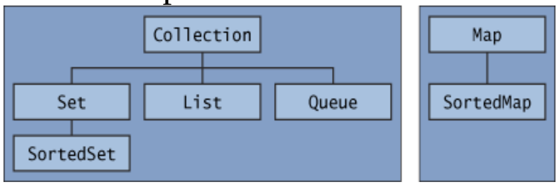
These are the abstract data types that represent collection.

Implementation

These are the concrete implementation of collection interface

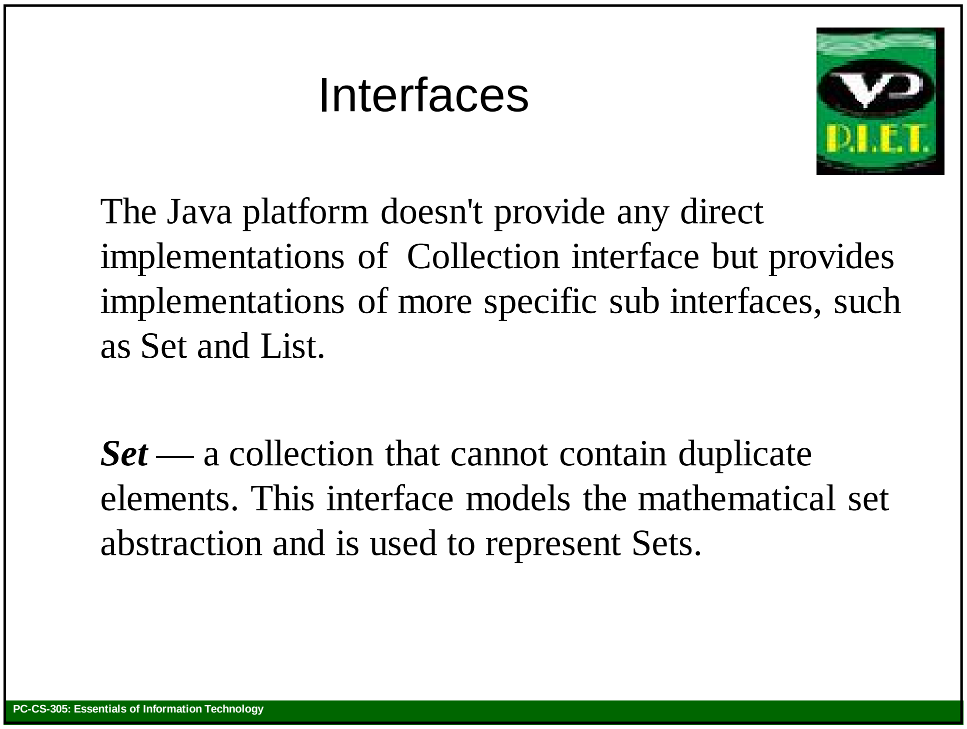
Interfaces

The core collection interfaces encapsulate different types of collections, These interfaces allow collections to be manipulated independently of the details of their representation.



A Set is a special kind of Collection.

A SortedSet is a special kind of Set.

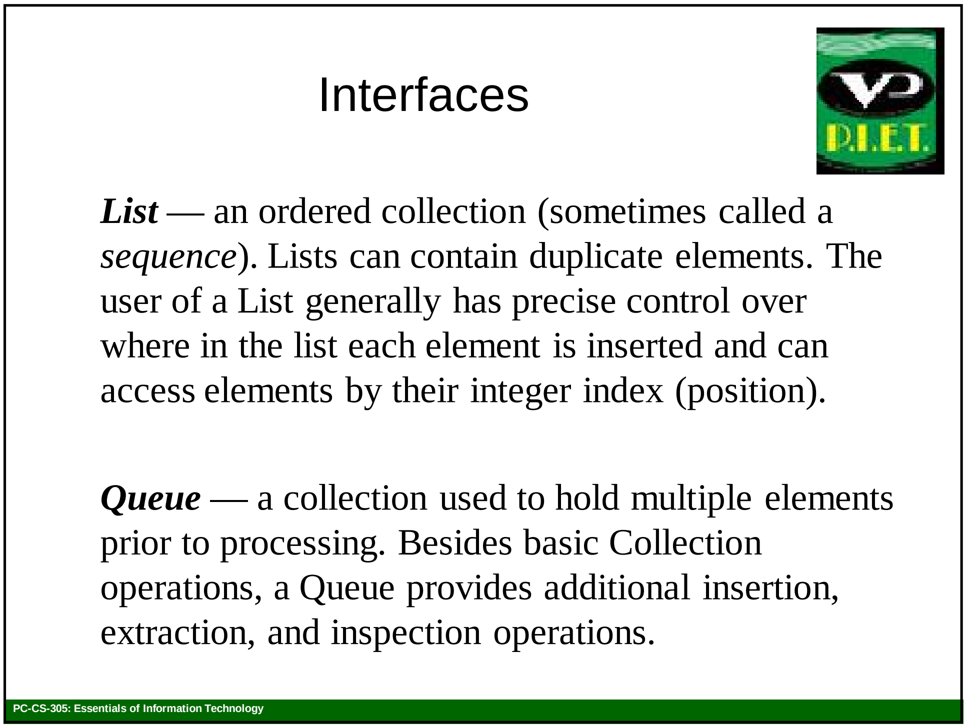


Interfaces

The Java platform doesn't provide any direct implementations of Collection interface but provides implementations of more specific sub interfaces, such as Set and List.

Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent Sets.

PC-CS-305: Essentials of Information Technology

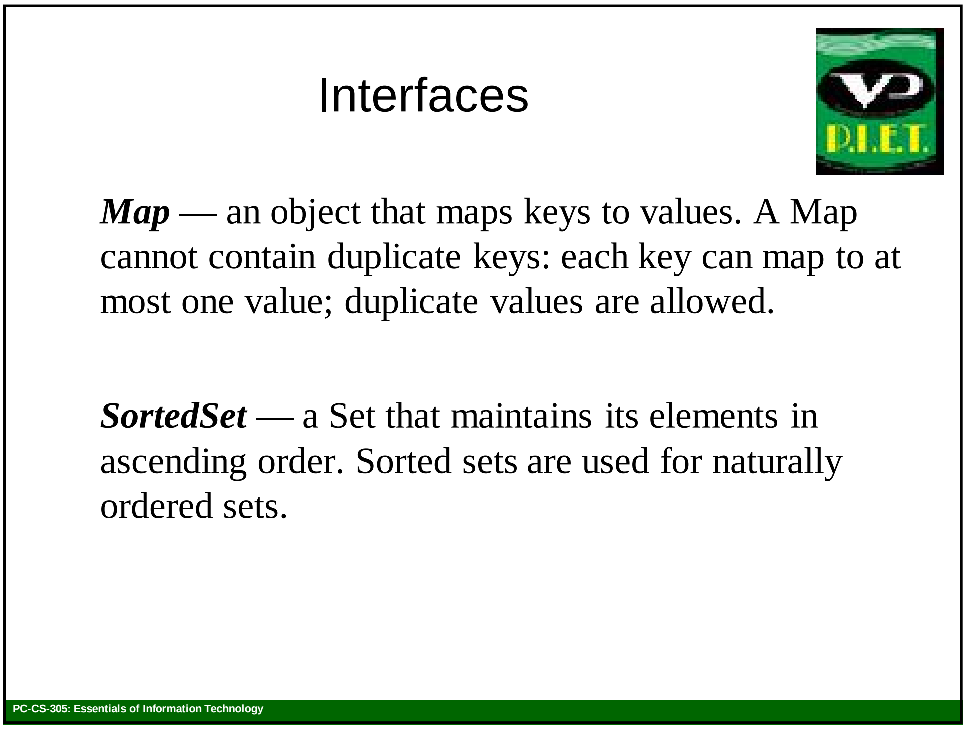


Interfaces

List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).

Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

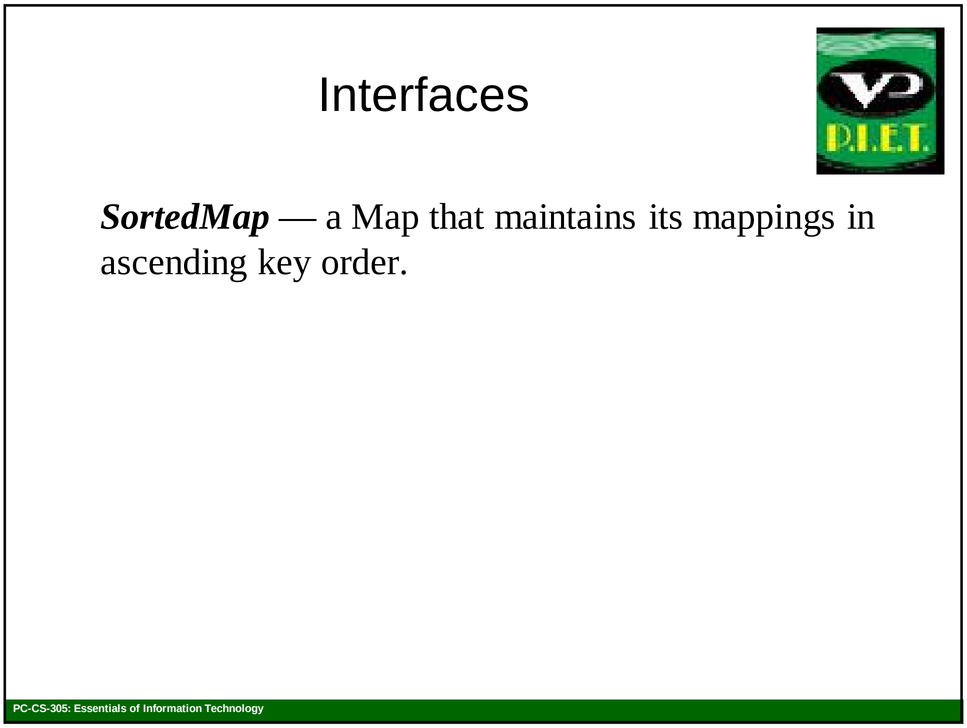
PC-CS-305: Essentials of Information Technology



Interfaces

Map — an object that maps keys to values. A Map cannot contain duplicate keys: each key can map to at most one value; duplicate values are allowed.

SortedSet — a Set that maintains its elements in ascending order. Sorted sets are used for naturally ordered sets.



Interfaces

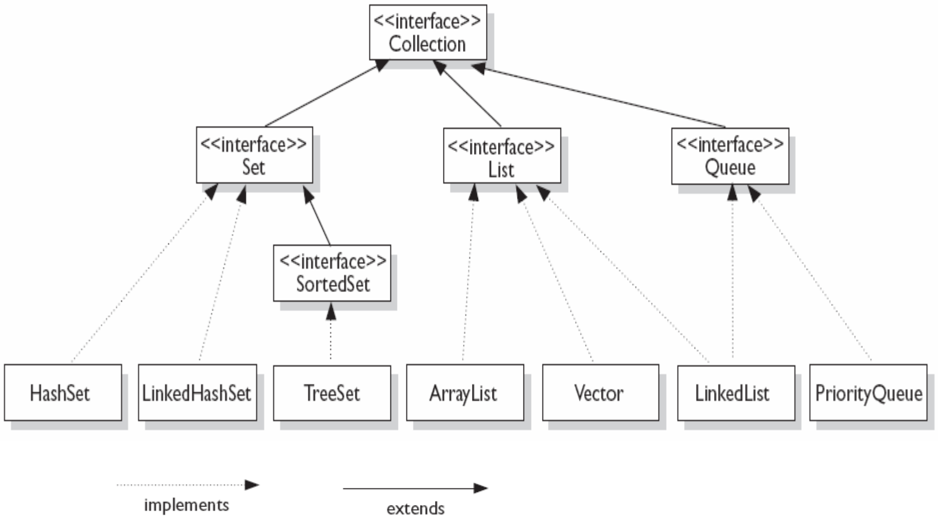
SortedMap — a Map that maintains its mappings in ascending key order.

Implementation

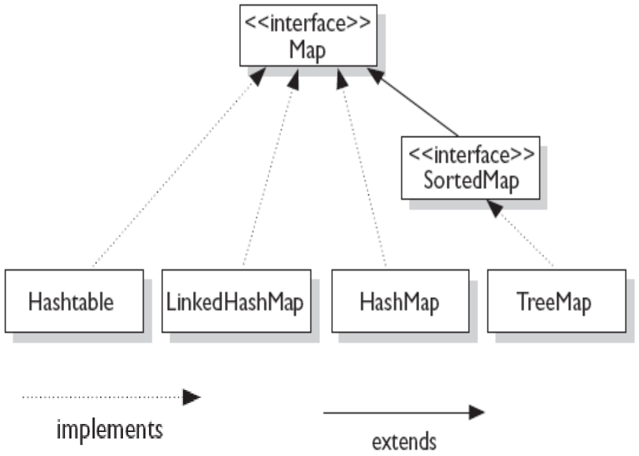
The classes that implement these interfaces are listed below:

Set	HashSet	LinkedHashSet	
List	Vector	ArrayList	LinkedList
Map	HashMap	HashTable	LinkedHashMap
SortedSet	TreeSet		
SortedMap	TreeMap		
Queue	LinkedList	PriorityQueue	

Implementation

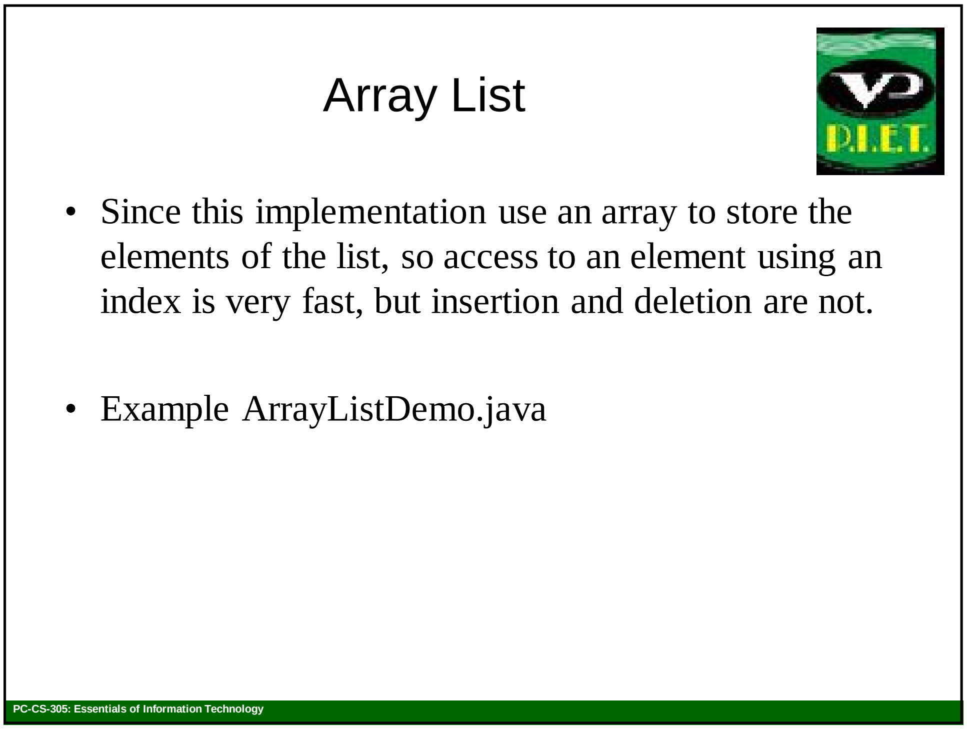


Implementation



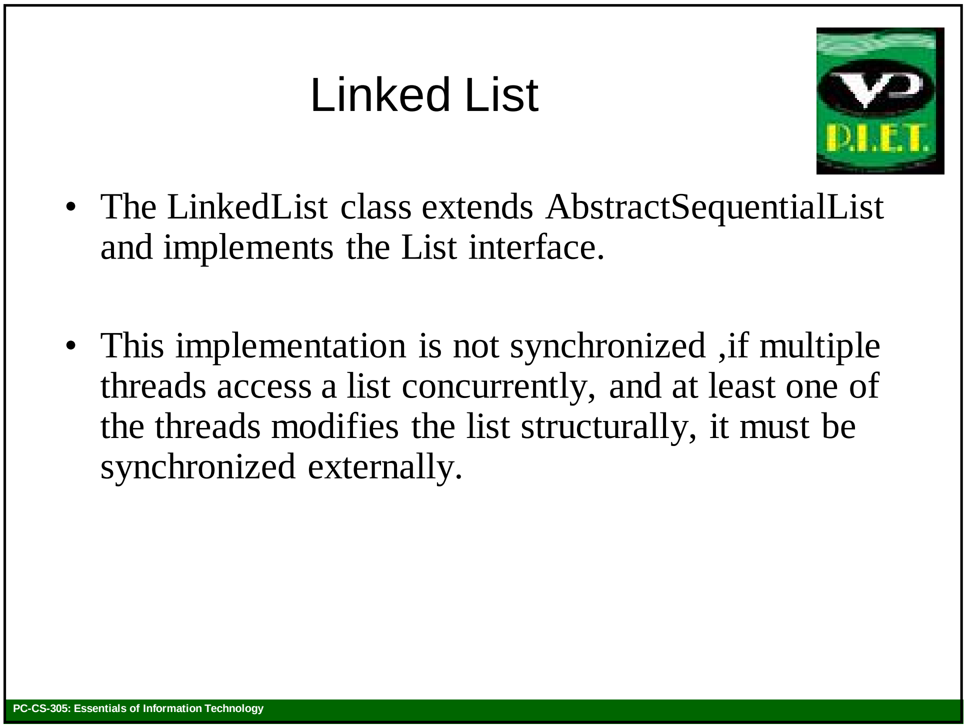
Array List

- The ArrayList class extends AbstractList and implements the List interface.
- ArrayList supports dynamic arrays that can grow as needed.
- An ArrayList is a variable-length array of object references.



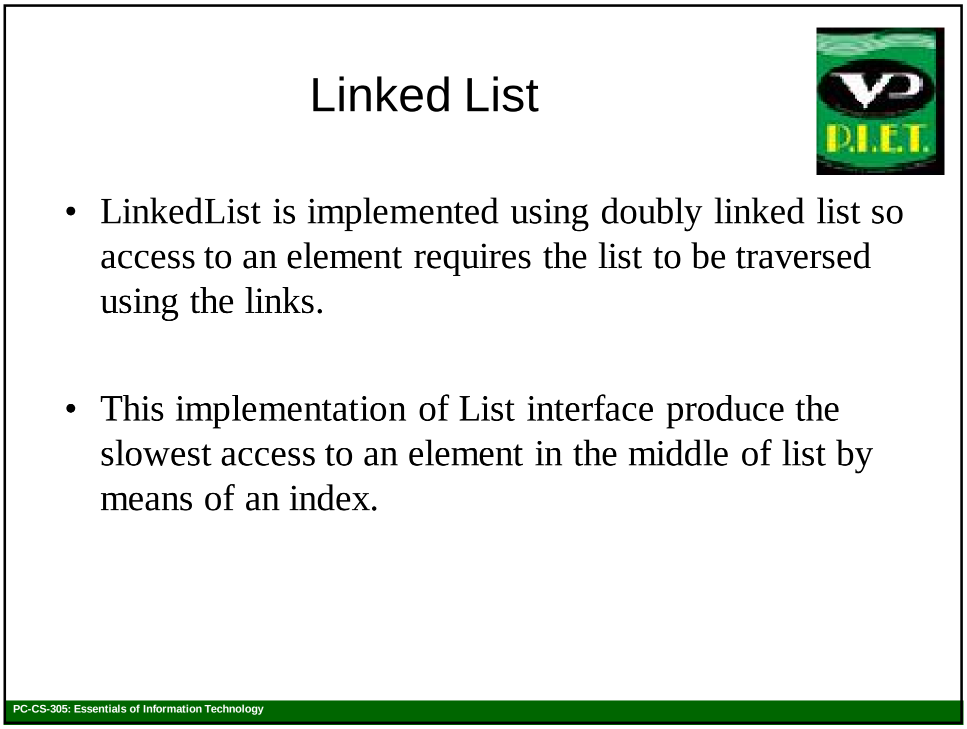
Array List

- Since this implementation use an array to store the elements of the list, so access to an element using an index is very fast, but insertion and deletion are not.
- Example ArrayListDemo.java



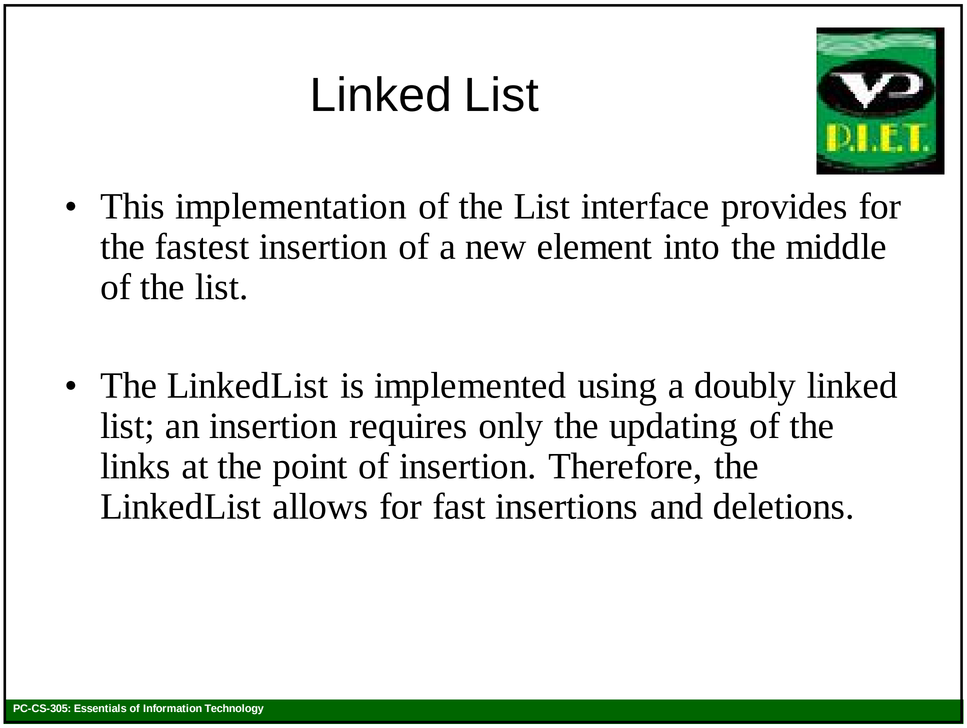
Linked List

- The LinkedList class extends AbstractSequentialList and implements the List interface.
- This implementation is not synchronized ,if multiple threads access a list concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.



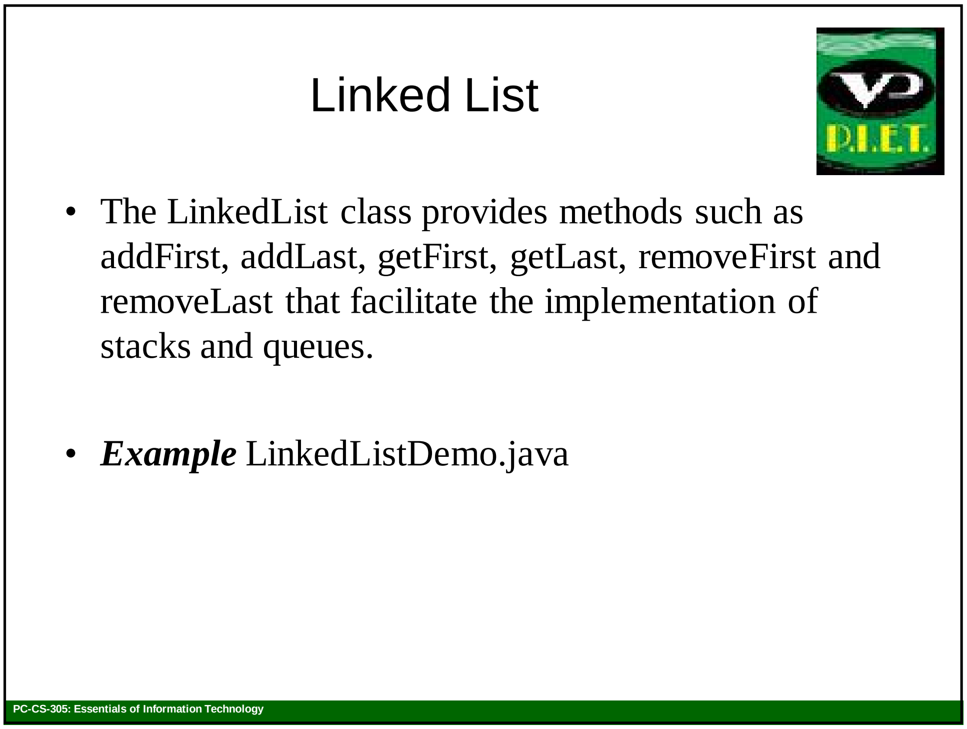
Linked List

- LinkedList is implemented using doubly linked list so access to an element requires the list to be traversed using the links.
- This implementation of List interface produce the slowest access to an element in the middle of list by means of an index.



Linked List

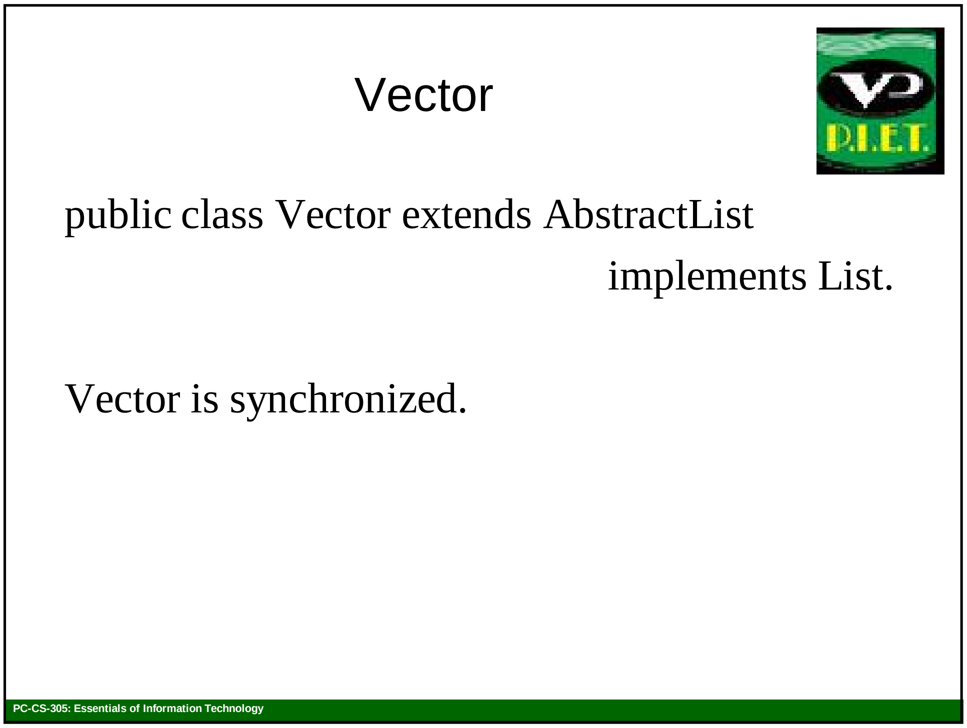
- This implementation of the List interface provides for the fastest insertion of a new element into the middle of the list.
- The LinkedList is implemented using a doubly linked list; an insertion requires only the updating of the links at the point of insertion. Therefore, the LinkedList allows for fast insertions and deletions.



Linked List

- The LinkedList class provides methods such as addFirst, addLast, getFirst, getLast, removeFirst and removeLast that facilitate the implementation of stacks and queues.

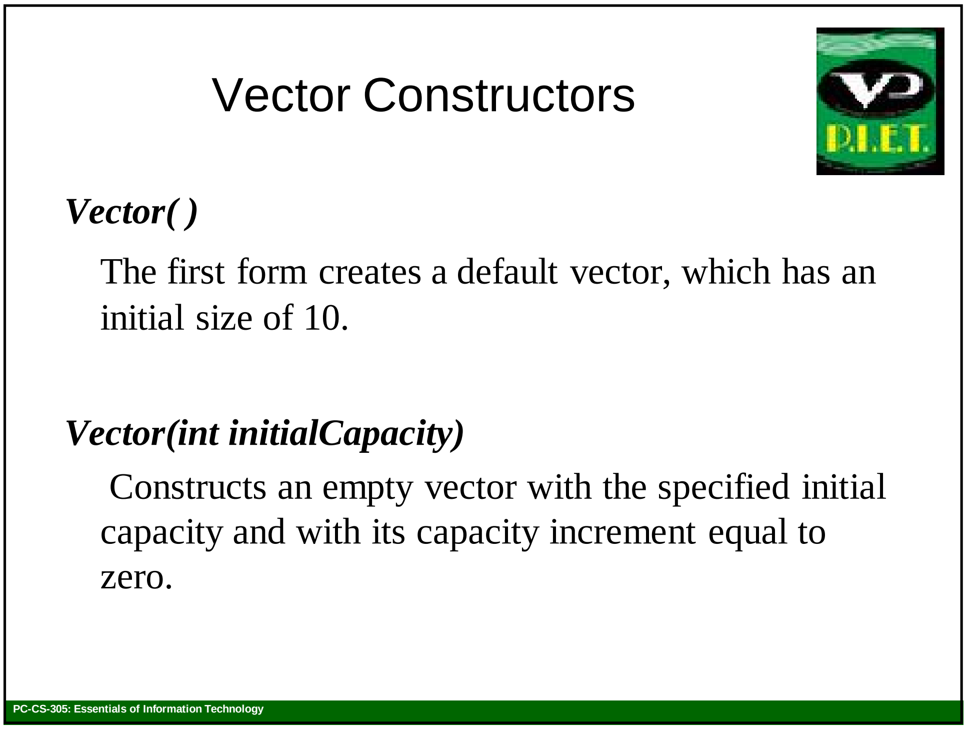
- ***Example*** LinkedListDemo.java



Vector

public class Vector extends AbstractList
implements List.

Vector is synchronized.



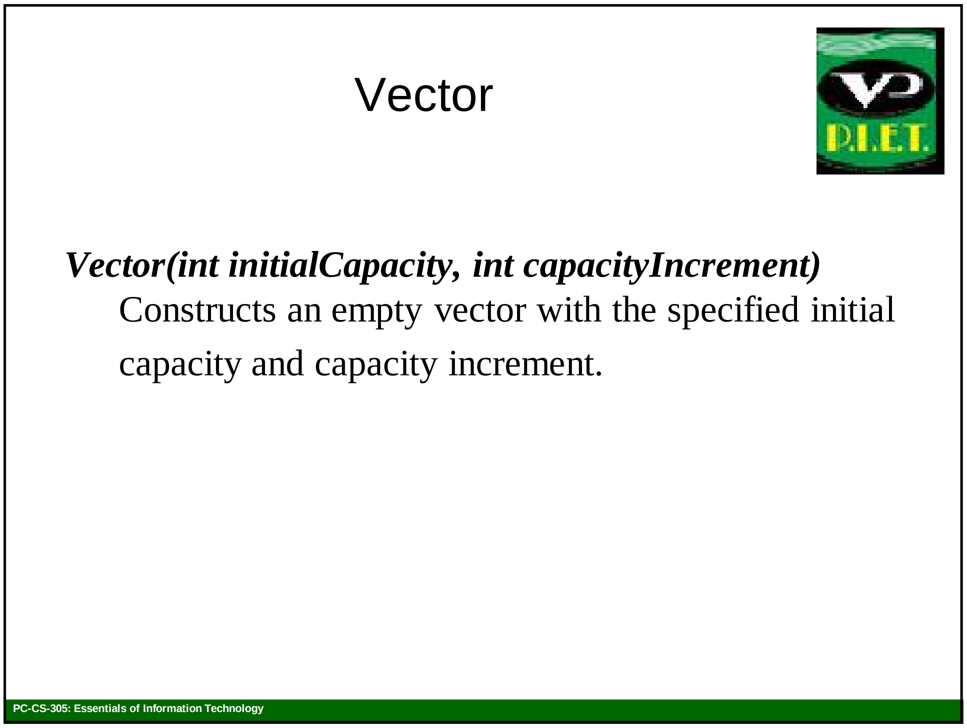
Vector Constructors

Vector( )

The first form creates a default vector, which has an initial size of 10.

Vector(int initialCapacity)

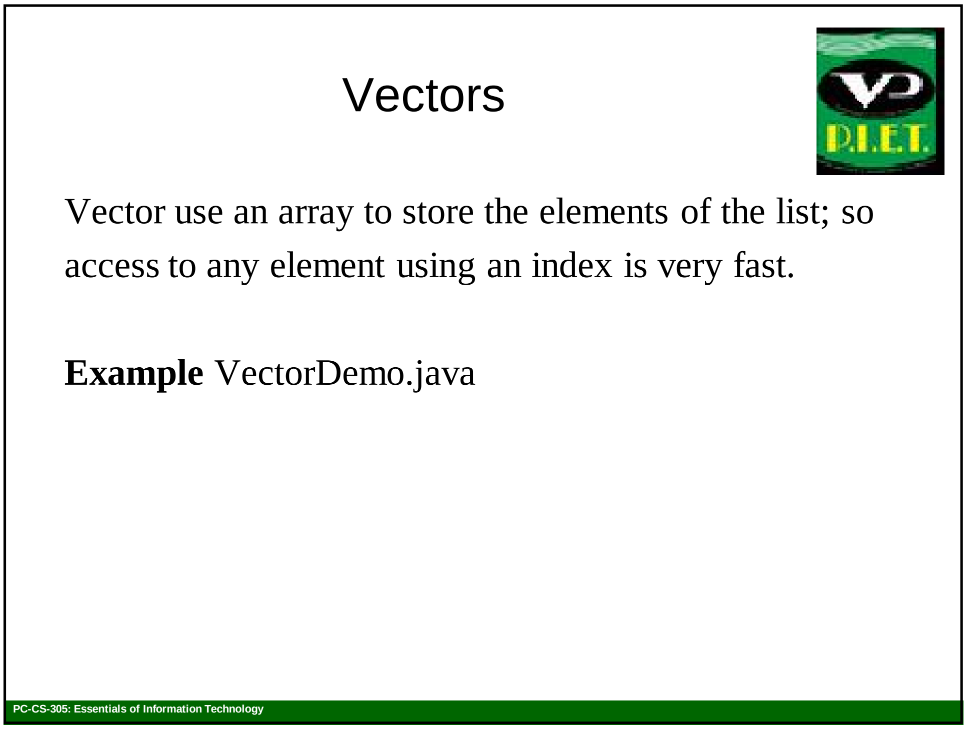
Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.



Vector

Vector(int initialCapacity, int capacityIncrement)

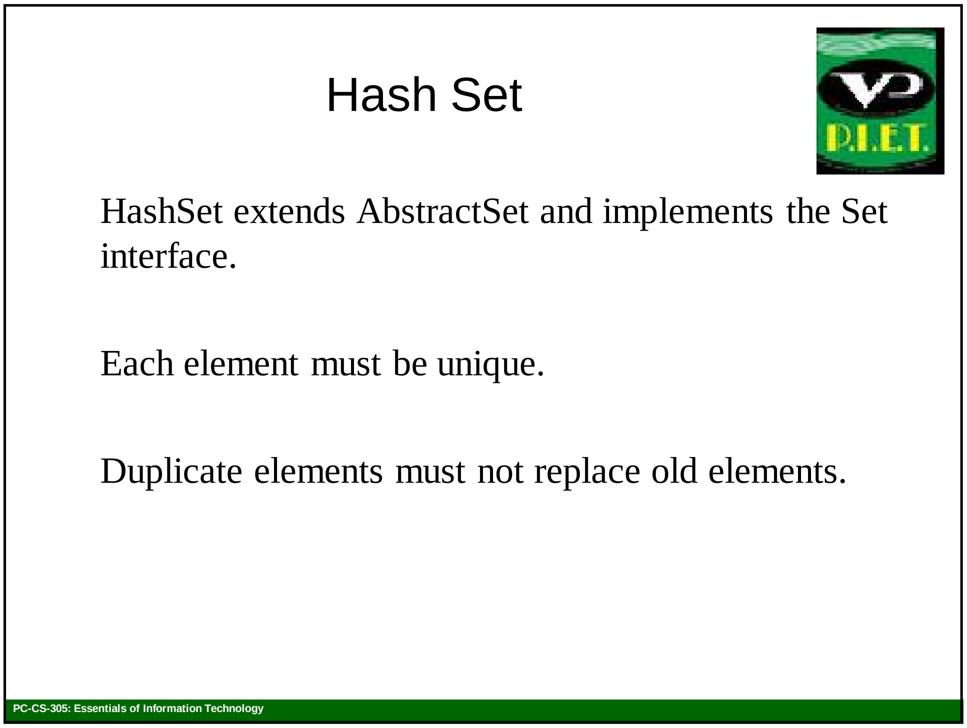
Constructs an empty vector with the specified initial capacity and capacity increment.



Vectors

Vector use an array to store the elements of the list; so access to any element using an index is very fast.

Example VectorDemo.java



Hash Set

HashSet extends AbstractSet and implements the Set interface.

Each element must be unique.

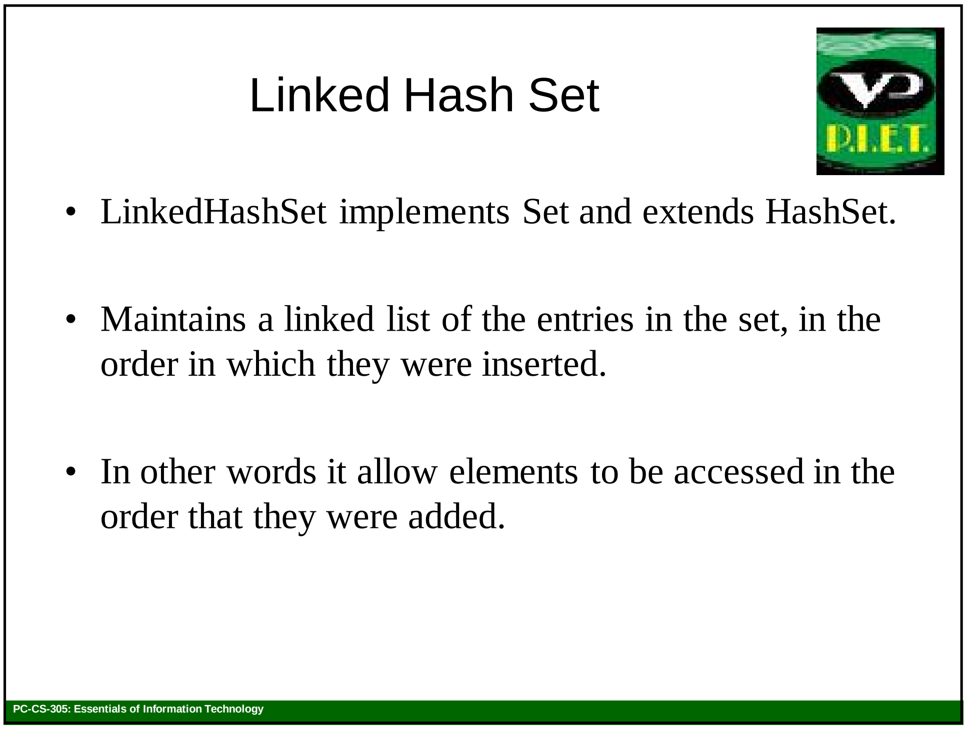
Duplicate elements must not replace old elements.

Hash Set

- Hash set does not guarantee the order of its elements, because the process of hashing doesn’t usually lend itself to the creation of sorted sets.
- This class permit the null elements.

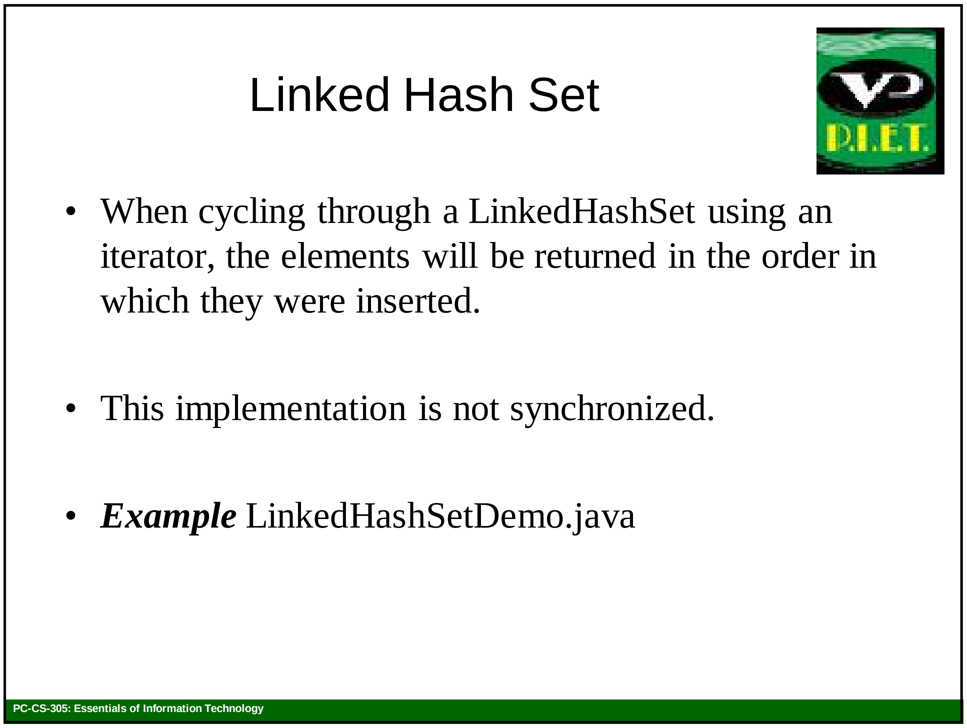
Hash Set

- Accessing an element is fast as compared to TreeSet.
- This implementation is not synchronized.
- Example HashSetDemo.java



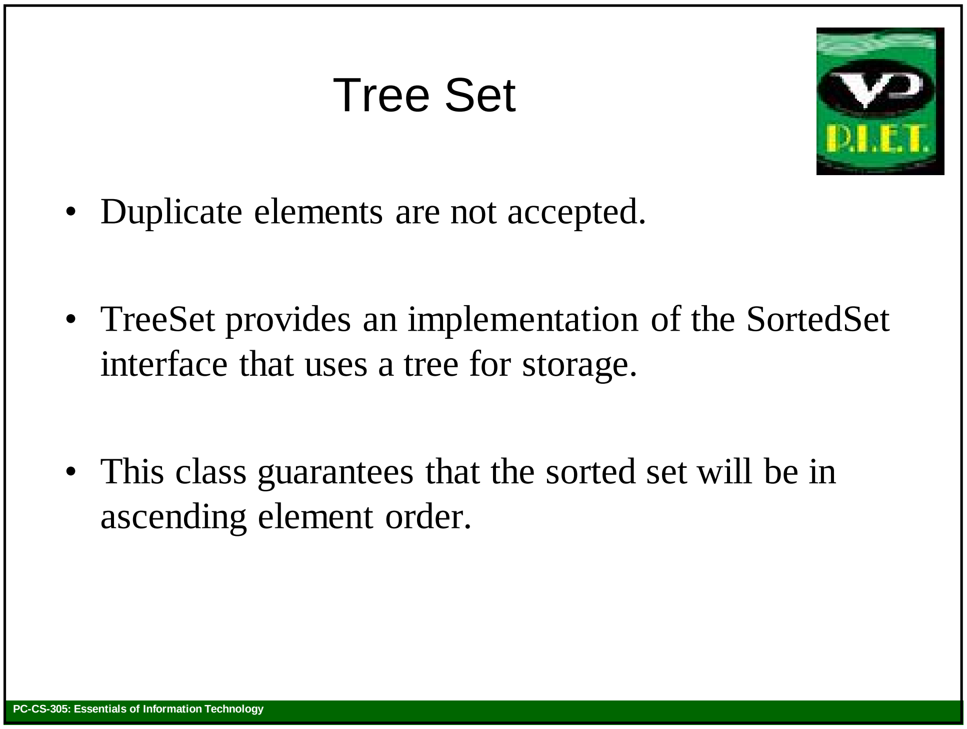
Linked Hash Set

- LinkedHashSet implements Set and extends HashSet.
- Maintains a linked list of the entries in the set, in the order in which they were inserted.
- In other words it allow elements to be accessed in the order that they were added.



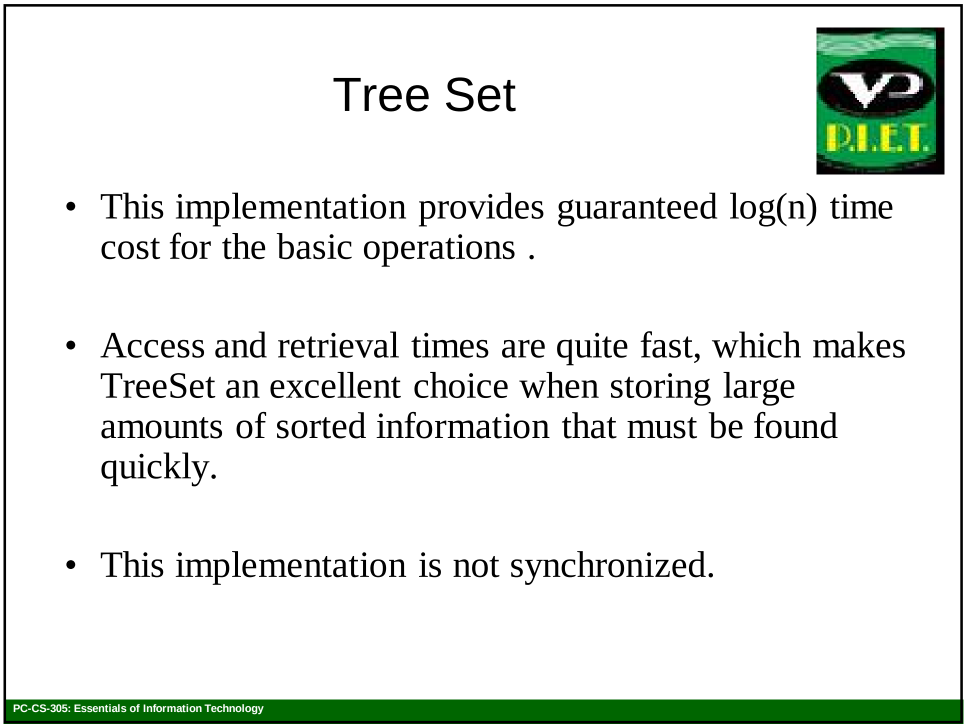
Linked Hash Set

- When cycling through a LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.
- This implementation is not synchronized.
- ***Example*** LinkedHashSetDemo.java



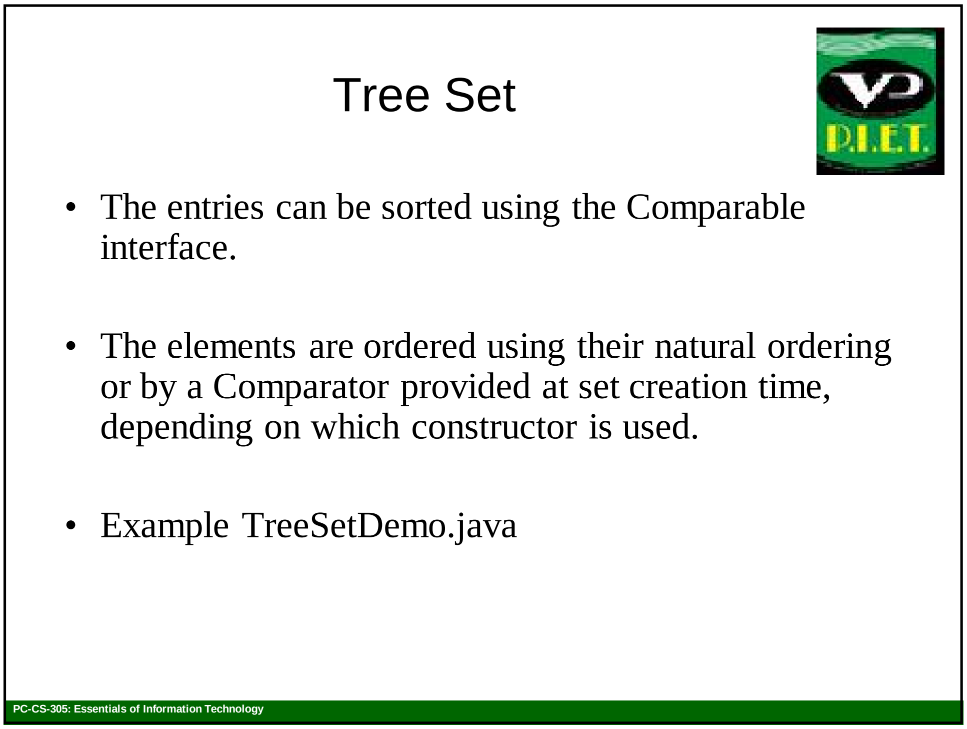
Tree Set

- Duplicate elements are not accepted.
- TreeSet provides an implementation of the SortedSet interface that uses a tree for storage.
- This class guarantees that the sorted set will be in ascending element order.



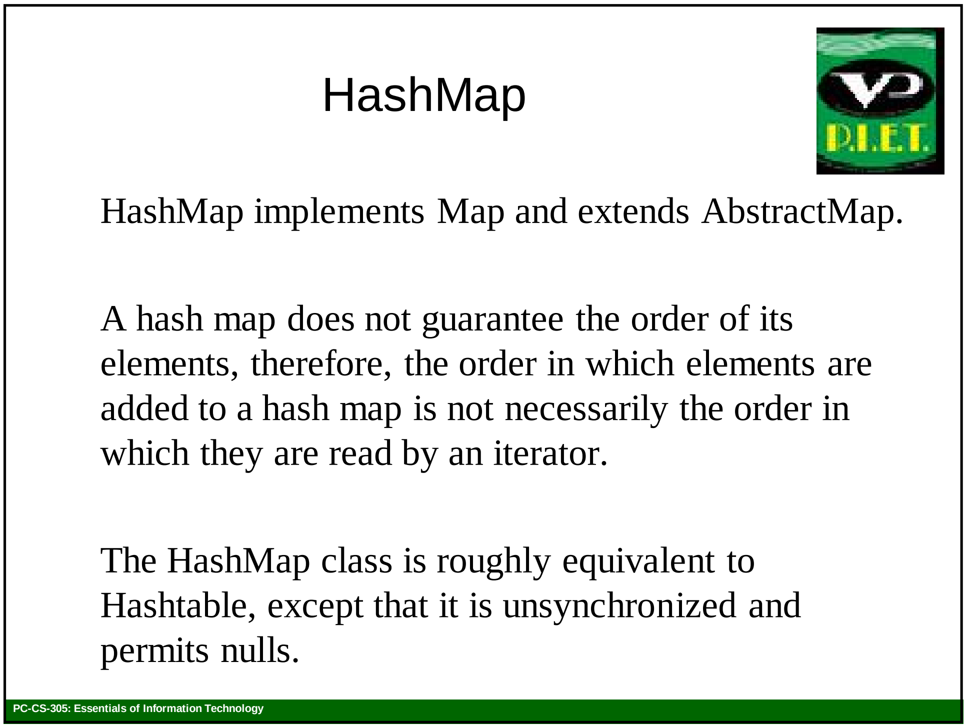
Tree Set

- This implementation provides guaranteed log(n) time cost for the basic operations .
- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.
- This implementation is not synchronized.



Tree Set

- The entries can be sorted using the Comparable interface.
- The elements are ordered using their natural ordering or by a Comparator provided at set creation time, depending on which constructor is used.
- Example TreeSetDemo.java



HashMap

HashMap implements Map and extends AbstractMap.

A hash map does not guarantee the order of its elements, therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.

Hash Map

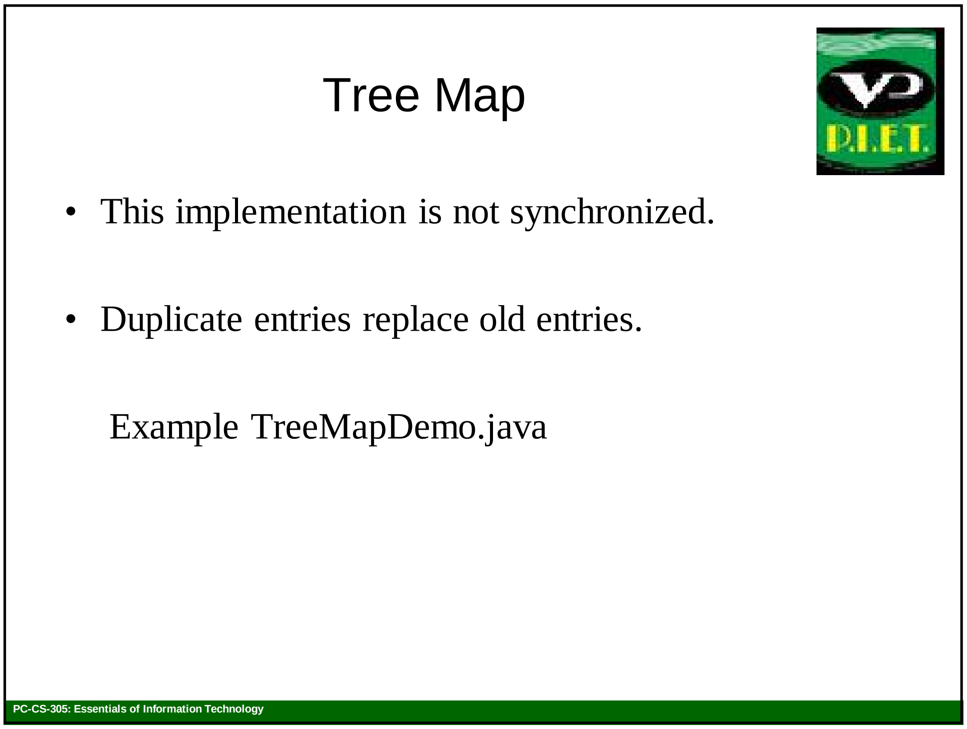
- HashMap lets you have null values as well as one null key

Example HashMapDemo.java

Tree Map

TreeMap implements SortedMap and extends AbstractMap.

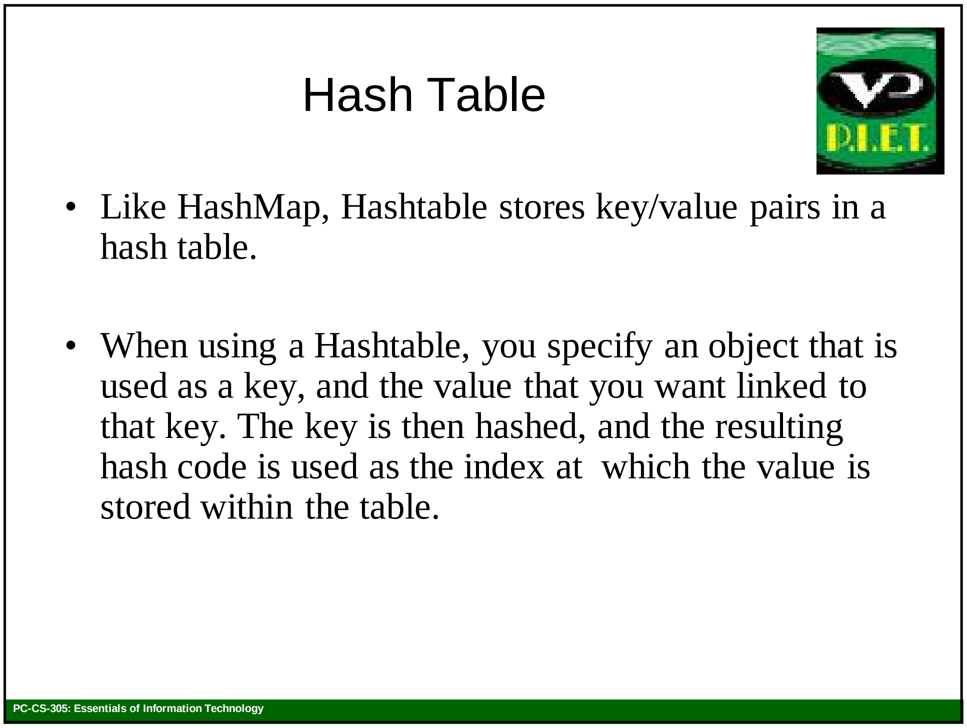
- A TreeMap provides an efficient means of storing key/value pairs in sorted order.
- Entries are sorted using Comparator or the Comparable interface.



Tree Map

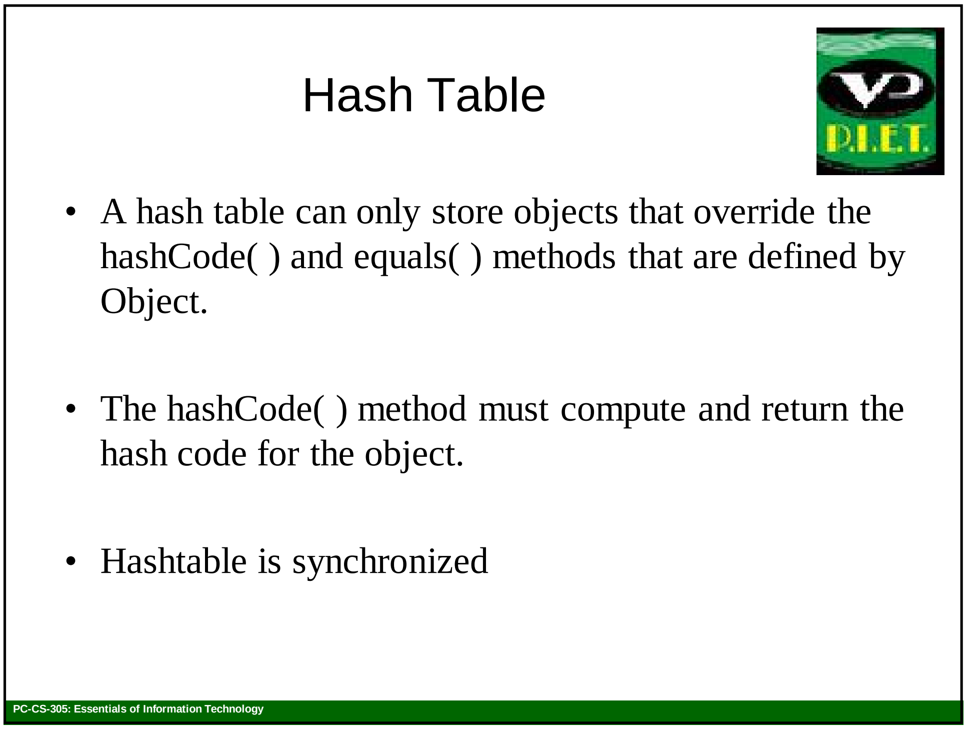
- This implementation is not synchronized.
- Duplicate entries replace old entries.

Example TreeMapDemo.java



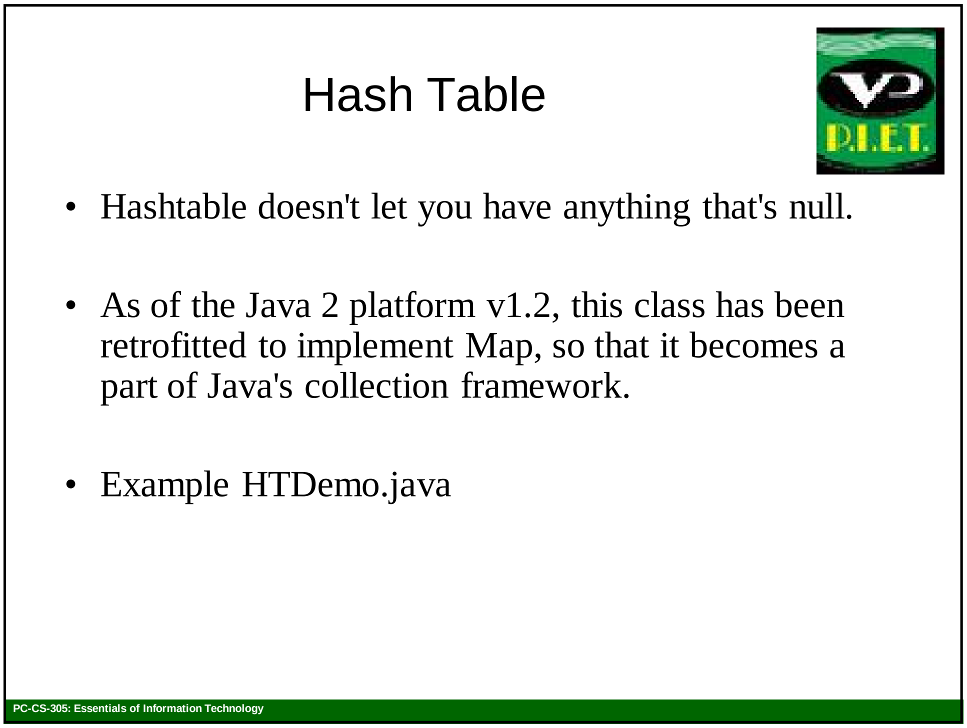
Hash Table

- Like HashMap, Hashtable stores key/value pairs in a hash table.
- When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.



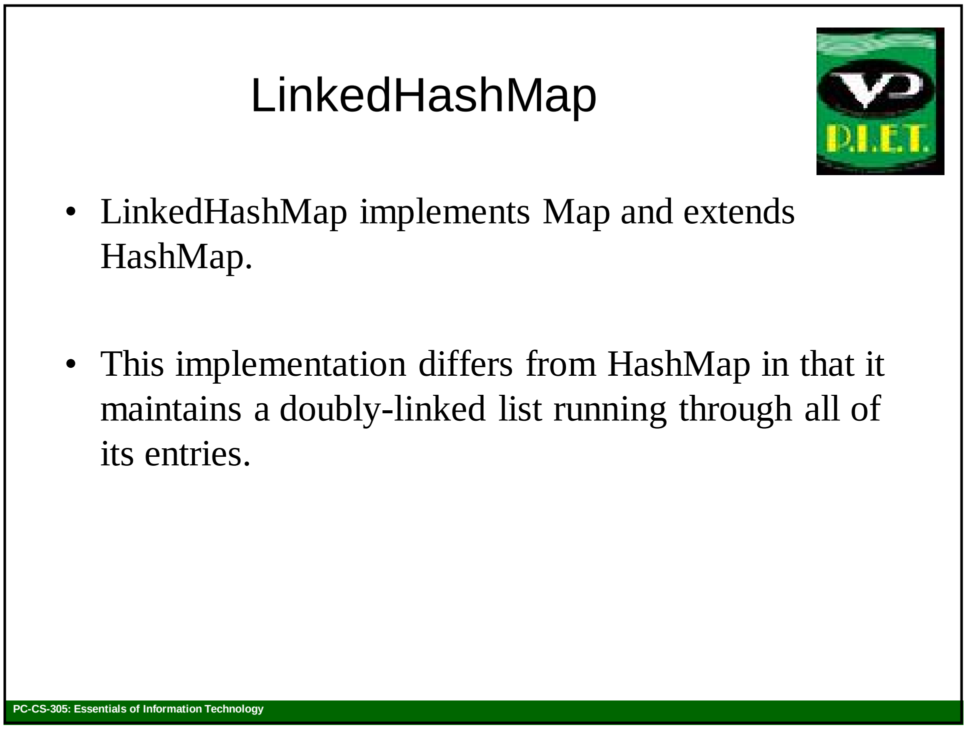
Hash Table

- A hash table can only store objects that override the hashCode( ) and equals( ) methods that are defined by Object.
- The hashCode( ) method must compute and return the hash code for the object.
- Hashtable is synchronized



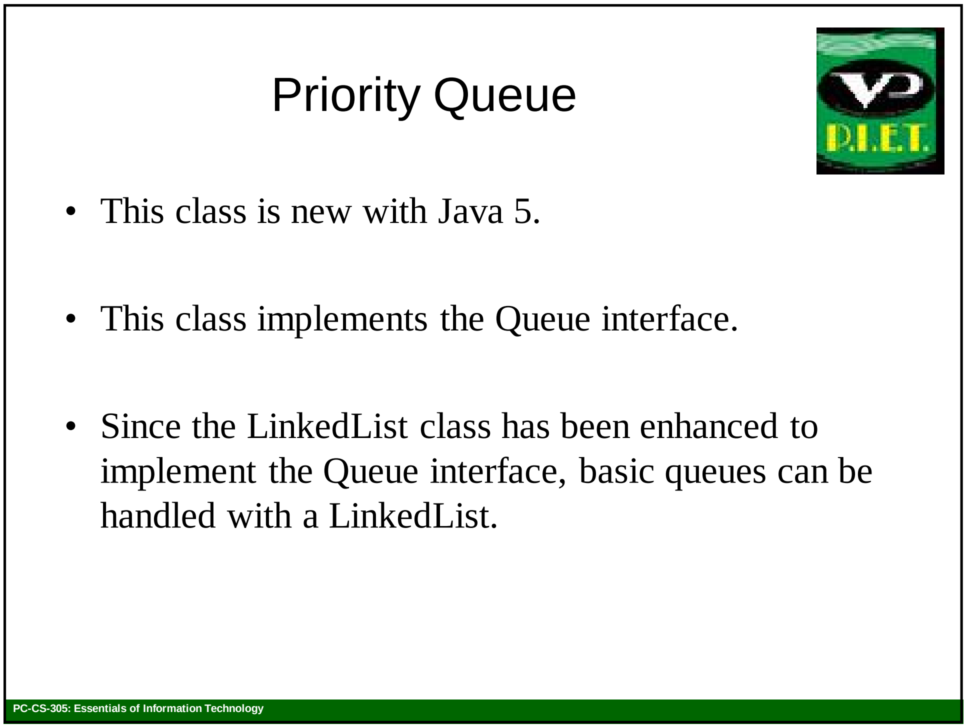
Hash Table

- Hashtable doesn't let you have anything that's null.
- As of the Java 2 platform v1.2, this class has been retrofitted to implement Map, so that it becomes a part of Java's collection framework.
- Example HTDemo.java



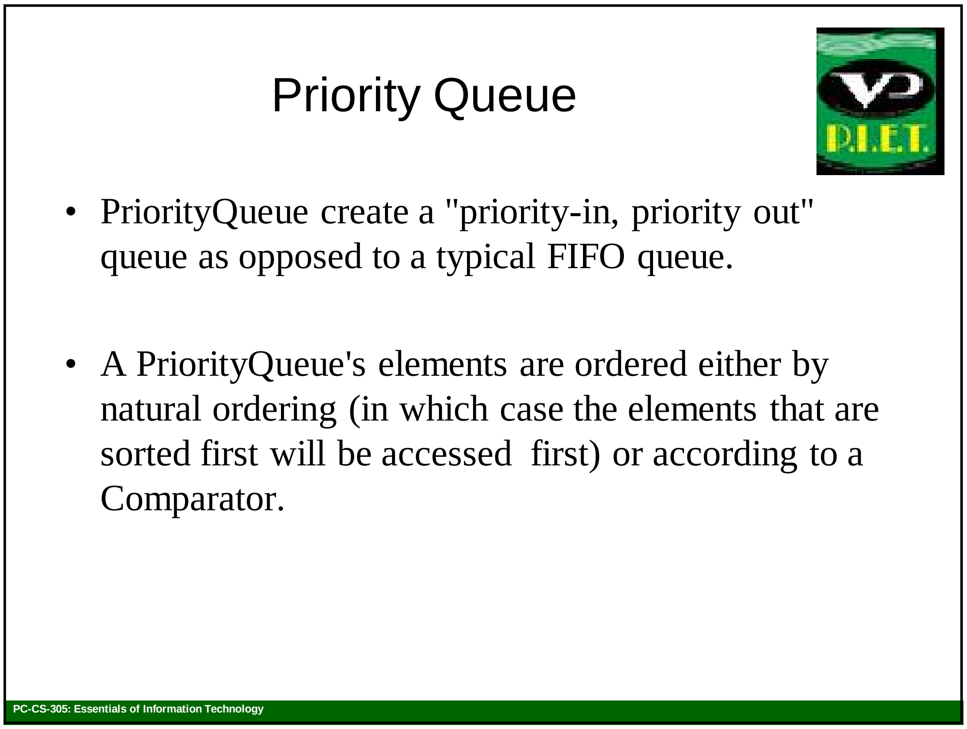
LinkedHashMap

- LinkedHashMap implements Map and extends HashMap.
- This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries.



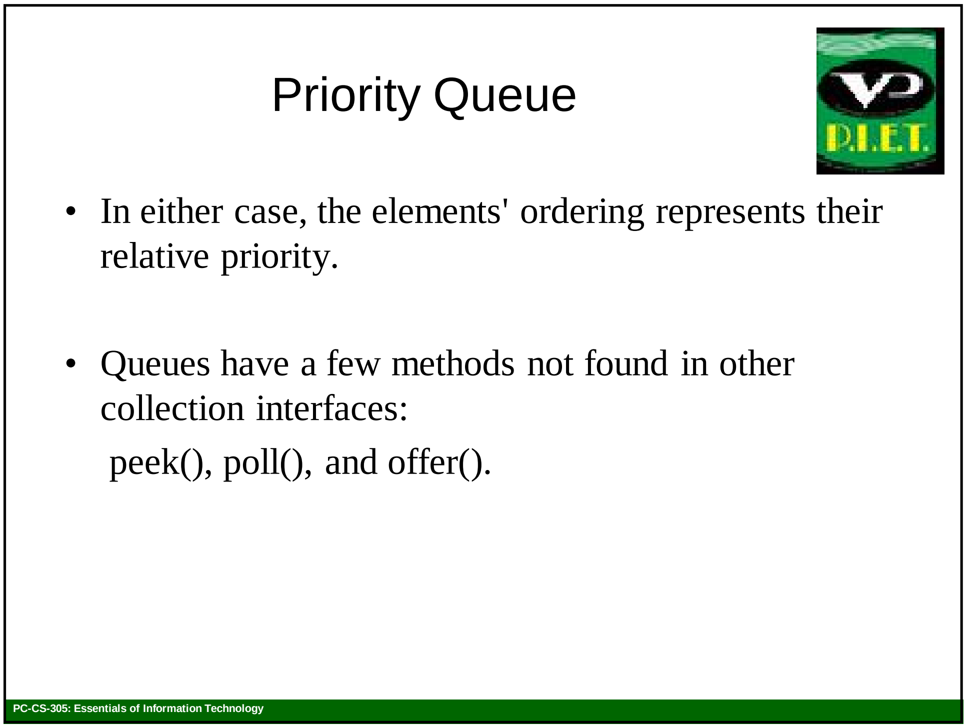
Priority Queue

- This class is new with Java 5.
- This class implements the Queue interface.
- Since the LinkedList class has been enhanced to implement the Queue interface, basic queues can be handled with a LinkedList.



Priority Queue

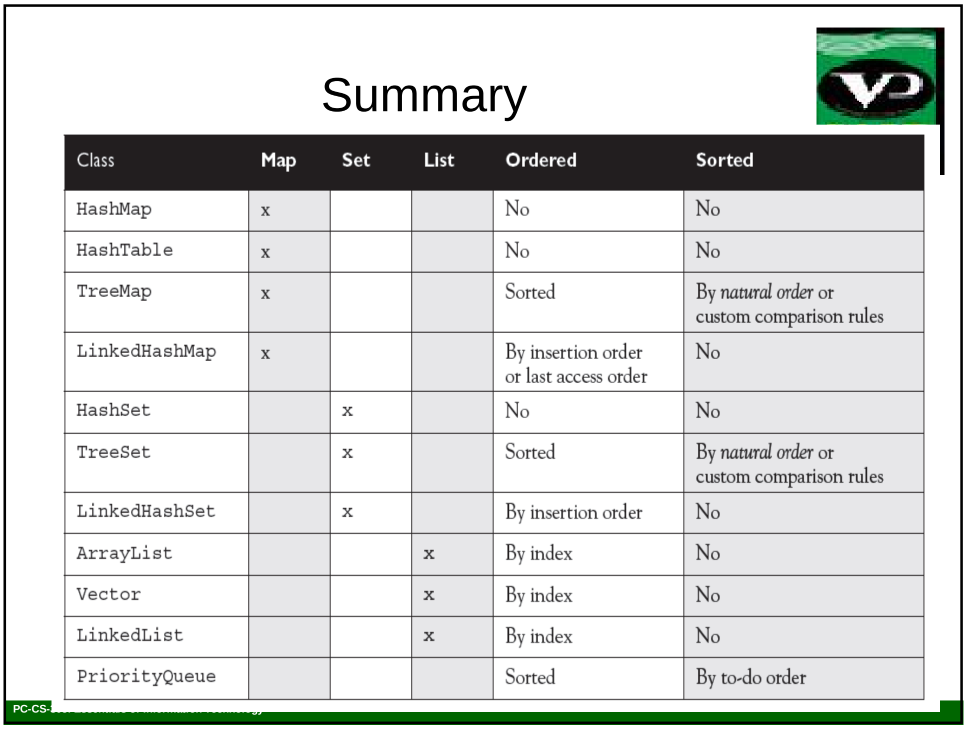
- PriorityQueue create a "priority-in, priority out" queue as opposed to a typical FIFO queue.
- A PriorityQueue's elements are ordered either by natural ordering (in which case the elements that are sorted first will be accessed first) or according to a Comparator.



Priority Queue

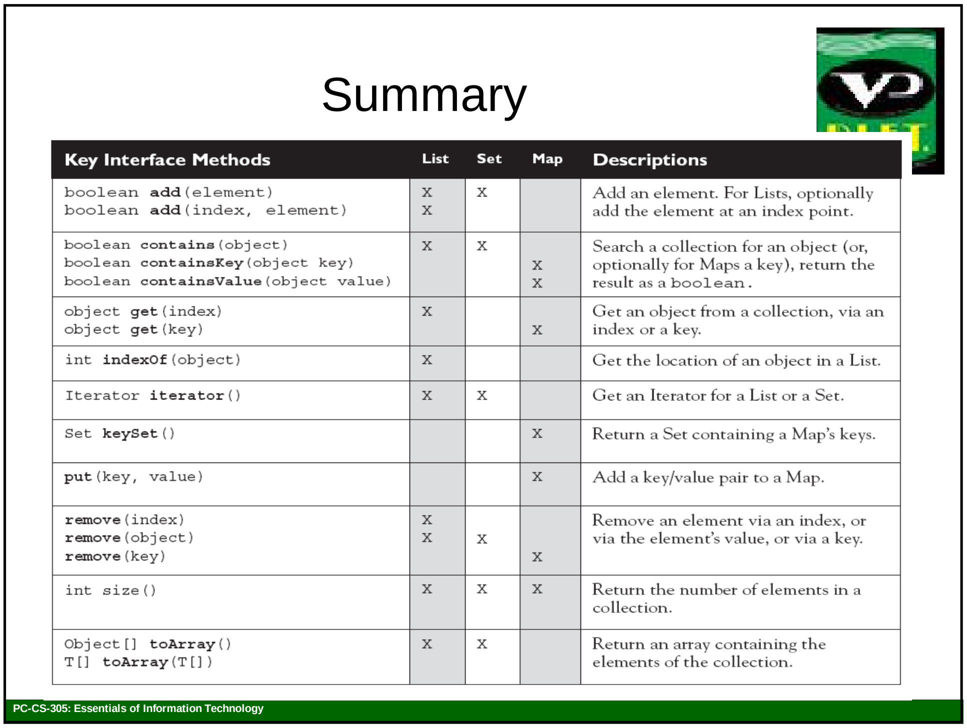
- In either case, the elements' ordering represents their relative priority.
- Queues have a few methods not found in other collection interfaces:

  peek(), poll(), and offer().



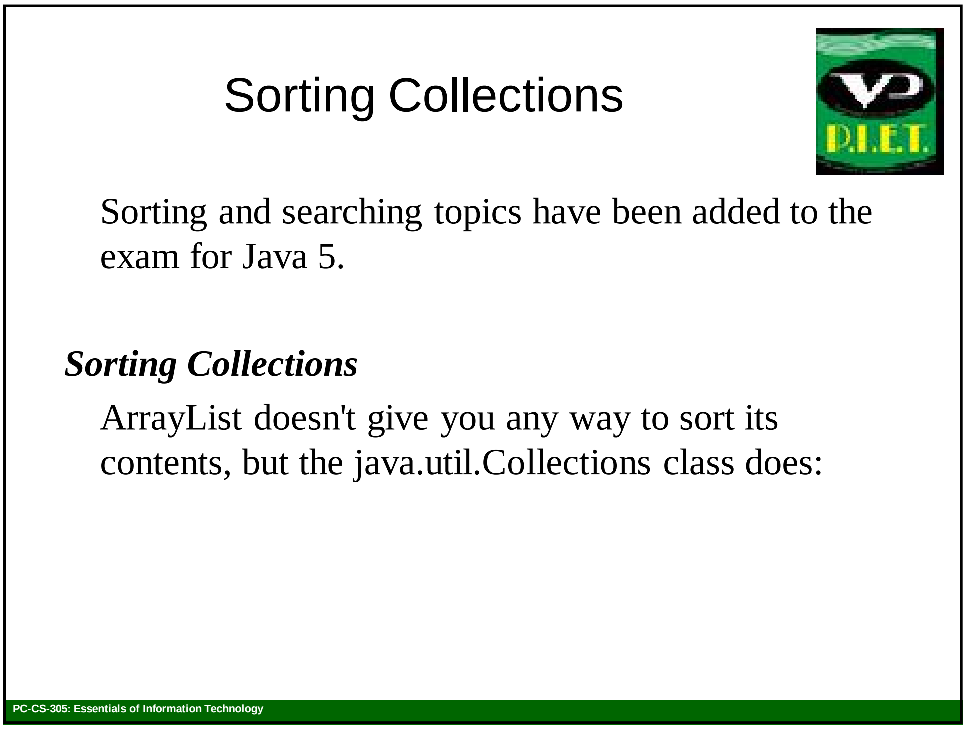
Summary

Class	Map	Set	List	Ordered	Sorted
HashMap	x			No	No
HashTable	x			No	No
TreeMap	x			Sorted	By *natural order* or custom comparison rules
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	By *natural order* or custom comparison rules
LinkedHashSet		x		By insertion order	No
ArrayList			x	By index	No
Vector			x	By index	No
LinkedList			x	By index	No
PriorityQueue				Sorted	By to-do order



Summary

Key Interface Methods	List	Set	Map	Descriptions
boolean **add**(element) boolean **add**(index, element)	X X	X		Add an element. For Lists, optionally add the element at an index point.
boolean **contains**(object) boolean **containsKey**(object key) boolean **containsValue**(object value)	X	X	 X X	Search a collection for an object (or, optionally for Maps a key), return the result as a boolean.
object **get**(index) object **get**(key)	X		 X	Get an object from a collection, via an index or a key.
int **indexOf**(object)	X			Get the location of an object in a List.
Iterator **iterator**()	X	X		Get an Iterator for a List or a Set.
Set **keySet**()			X	Return a Set containing a Map's keys.
put(key, value)			X	Add a key/value pair to a Map.
remove(index) **remove**(object) **remove**(key)	X X	 X	 X	Remove an element via an index, or via the element's value, or via a key.
int size()	X	X	X	Return the number of elements in a collection.
Object[] **toArray**() T[] **toArray**(T[])	X	X		Return an array containing the elements of the collection.

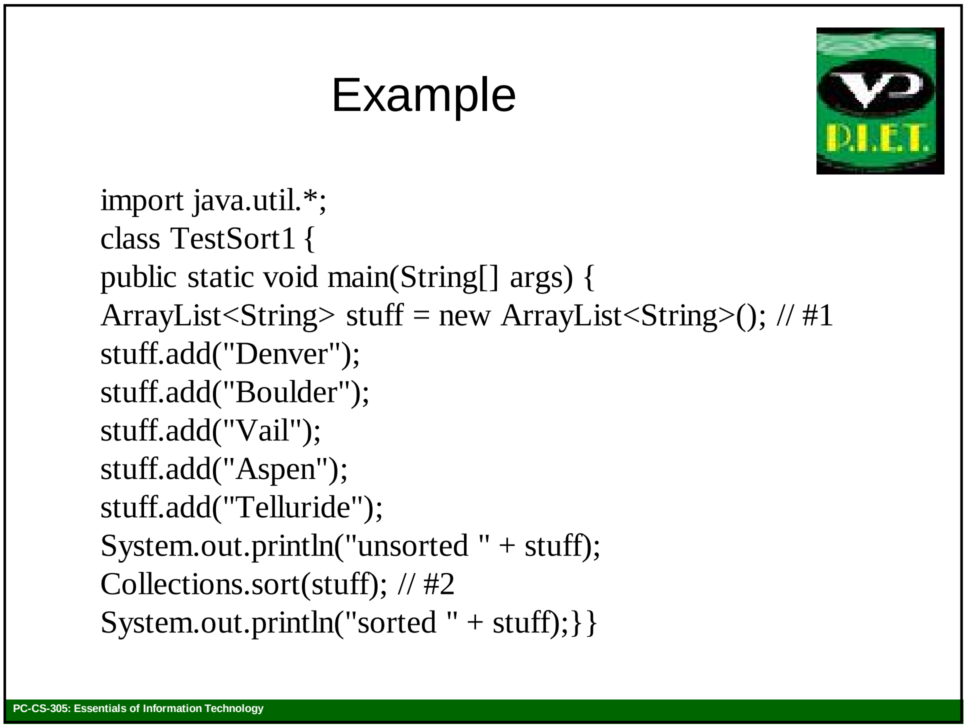


Sorting Collections

Sorting and searching topics have been added to the exam for Java 5.

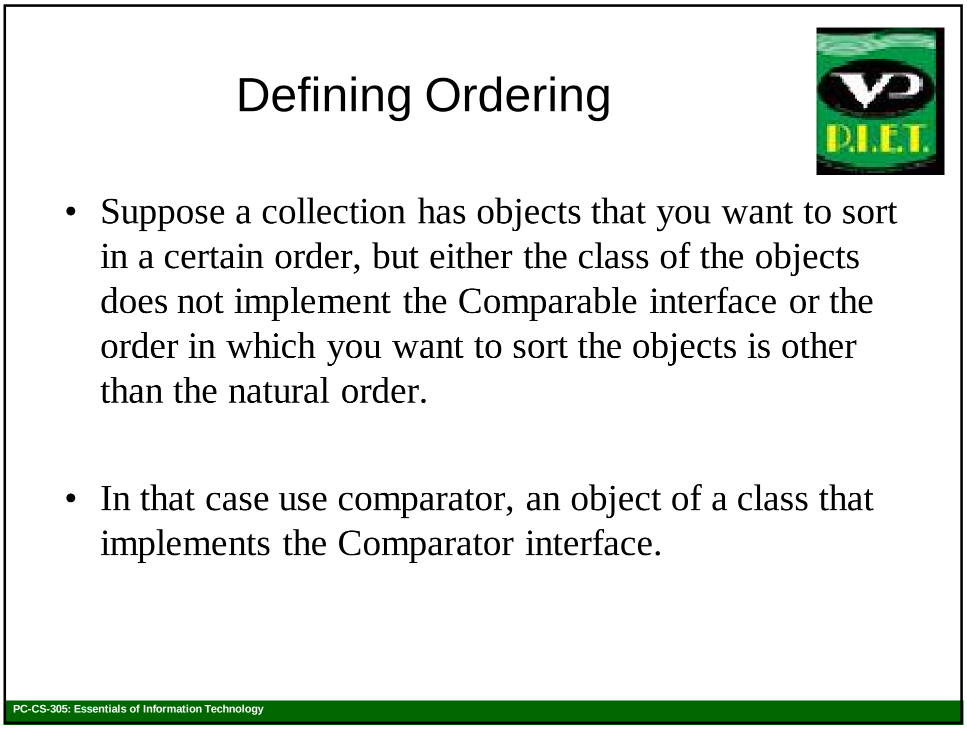
Sorting Collections

ArrayList doesn't give you any way to sort its contents, but the java.util.Collections class does:



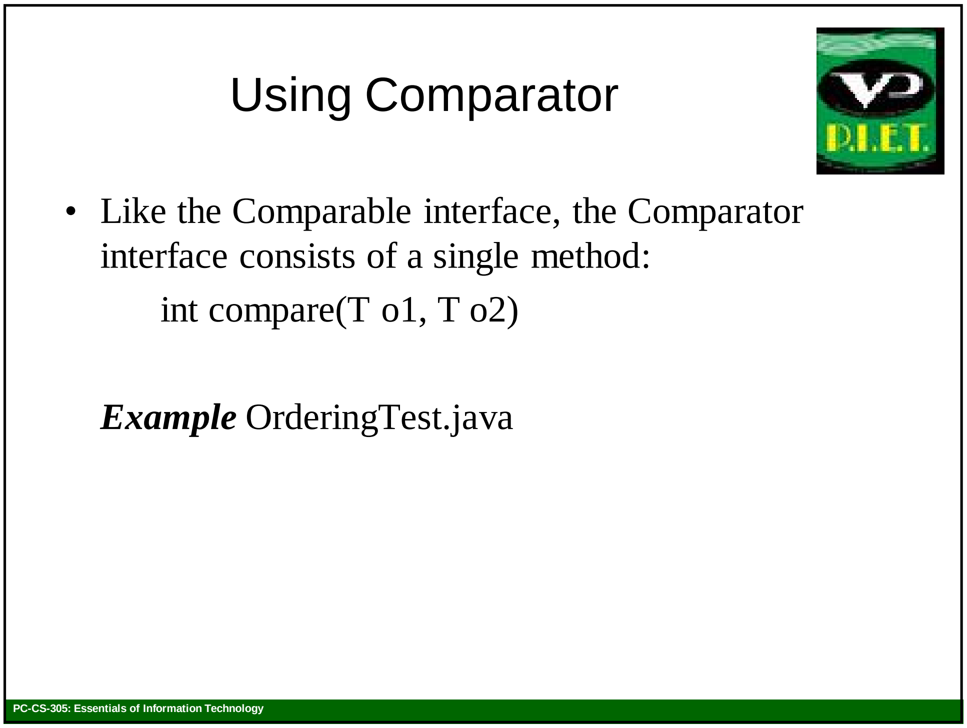
Example

```
import java.util.*;
class TestSort1 {
public static void main(String[] args) {
ArrayList<String> stuff = new ArrayList<String>(); // #1
stuff.add("Denver");
stuff.add("Boulder");
stuff.add("Vail");
stuff.add("Aspen");
stuff.add("Telluride");
System.out.println("unsorted " + stuff);
Collections.sort(stuff); // #2
System.out.println("sorted " + stuff);}}
```



Defining Ordering

- Suppose a collection has objects that you want to sort in a certain order, but either the class of the objects does not implement the Comparable interface or the order in which you want to sort the objects is other than the natural order.
- In that case use comparator, an object of a class that implements the Comparator interface.



Using Comparator

- Like the Comparable interface, the Comparator interface consists of a single method:

```
int compare(T o1, T o2)
```

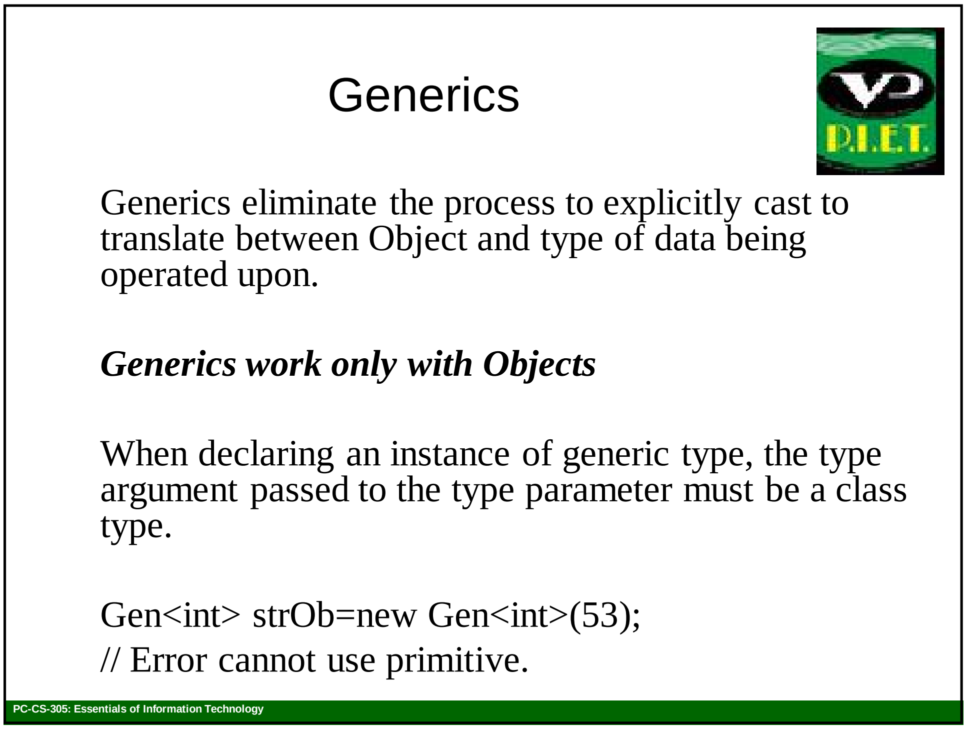
Example OrderingTest.java

Generics

- Generics means parameterized types.
- Parameterized types are important as they enable to create classes, interfaces, and methods in which the type of data upon which they operate is specified as parameter.
- Using Generics its possible to create a single class that automatically works with different types of data.

Generics

- The class, interface or methods that operate on a parameterized type is called generic.
- **Example** Gen, GenDemo



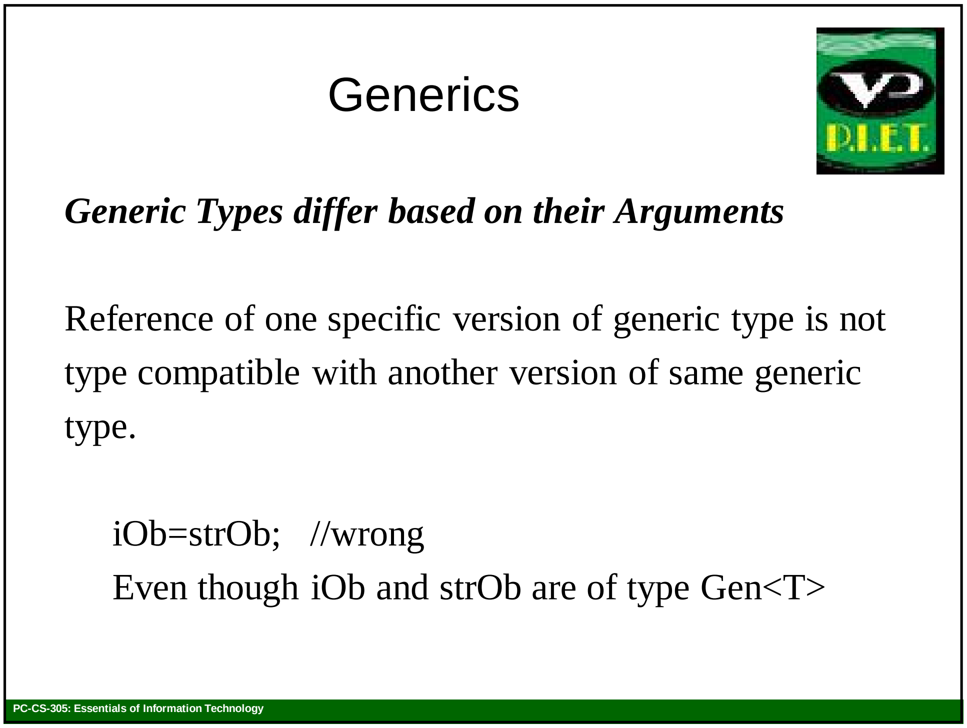
Generics

Generics eliminate the process to explicitly cast to translate between Object and type of data being operated upon.

Generics work only with Objects

When declaring an instance of generic type, the type argument passed to the type parameter must be a class type.

```
Gen<int> strOb=new Gen<int>(53);
// Error cannot use primitive.
```



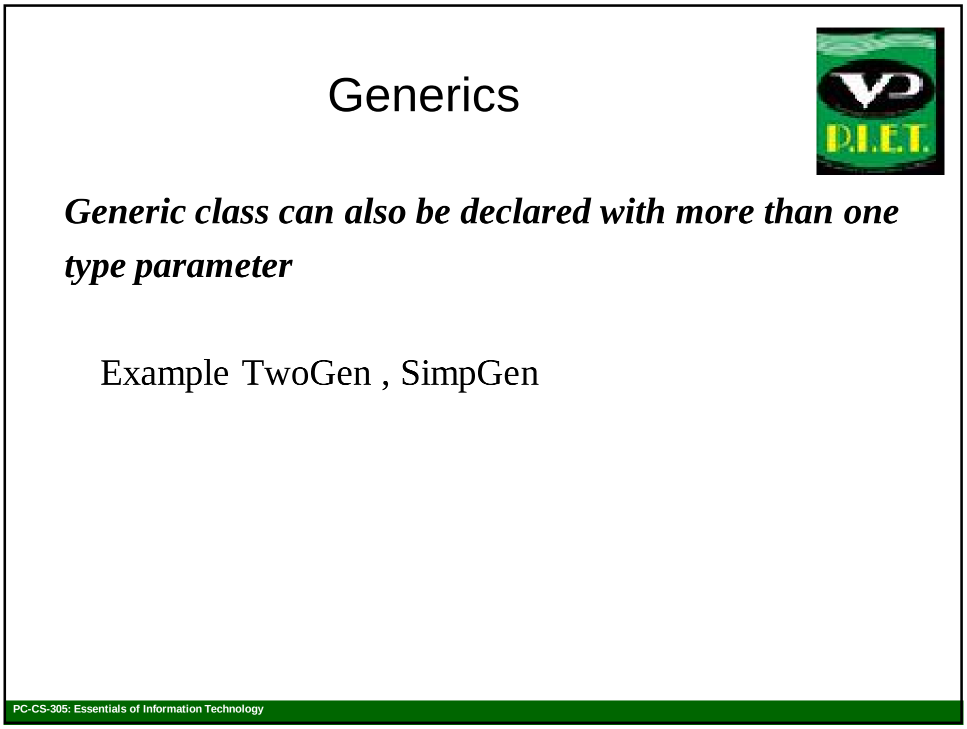
Generics

Generic Types differ based on their Arguments

Reference of one specific version of generic type is not type compatible with another version of same generic type.

```
iOb=strOb;   //wrong
```

Even though iOb and strOb are of type Gen<T>

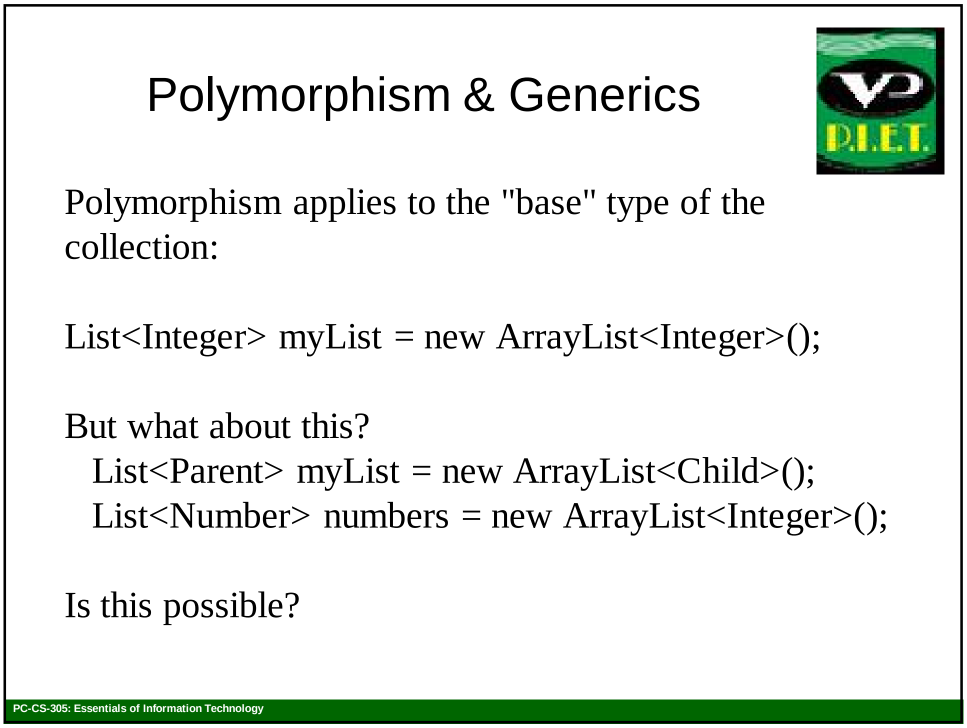


Generics

Generic class can also be declared with more than one type parameter

Example TwoGen , SimpGen

PC-CS-305: Essentials of Information Technology



Polymorphism & Generics

Polymorphism applies to the "base" type of the collection:

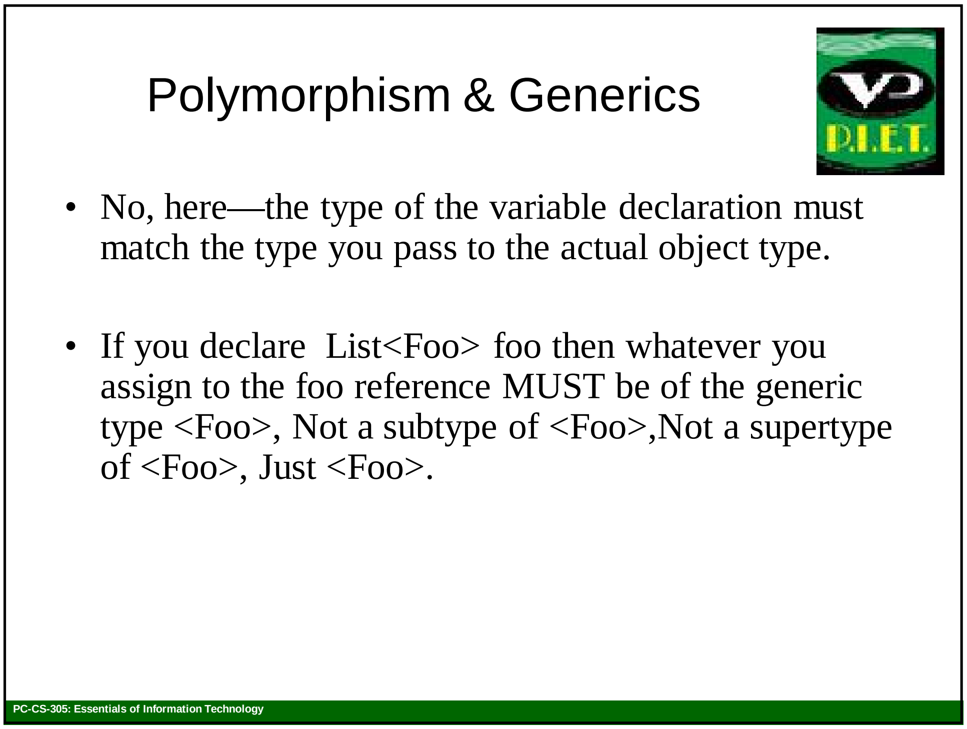
```
List<Integer> myList = new ArrayList<Integer>();
```

But what about this?

```
  List<Parent> myList = new ArrayList<Child>();
  List<Number> numbers = new ArrayList<Integer>();
```

Is this possible?

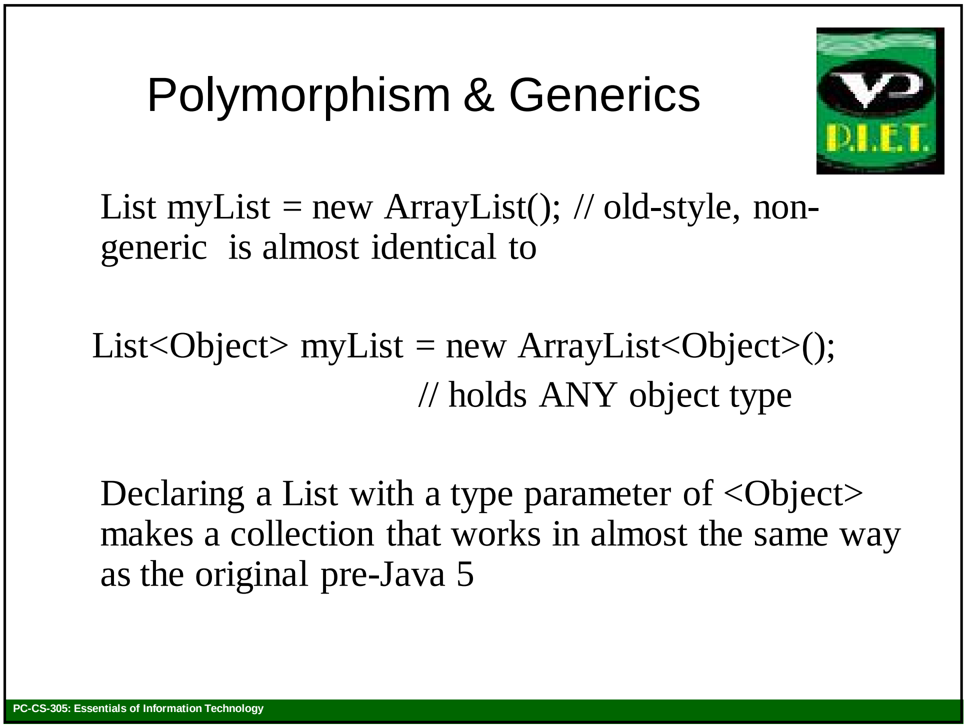
PC-CS-305: Essentials of Information Technology



Polymorphism & Generics

- No, here—the type of the variable declaration must match the type you pass to the actual object type.

- If you declare List<Foo> foo then whatever you assign to the foo reference MUST be of the generic type <Foo>, Not a subtype of <Foo>,Not a supertype of <Foo>, Just <Foo>.

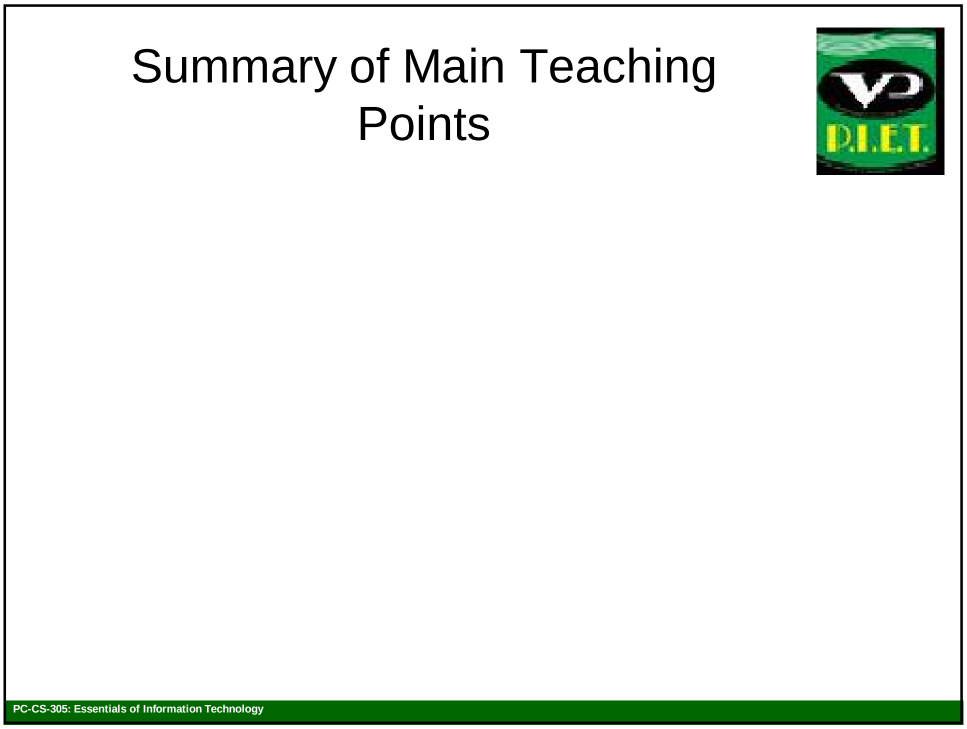


Polymorphism & Generics

List myList = new ArrayList(); // old-style, non-generic is almost identical to

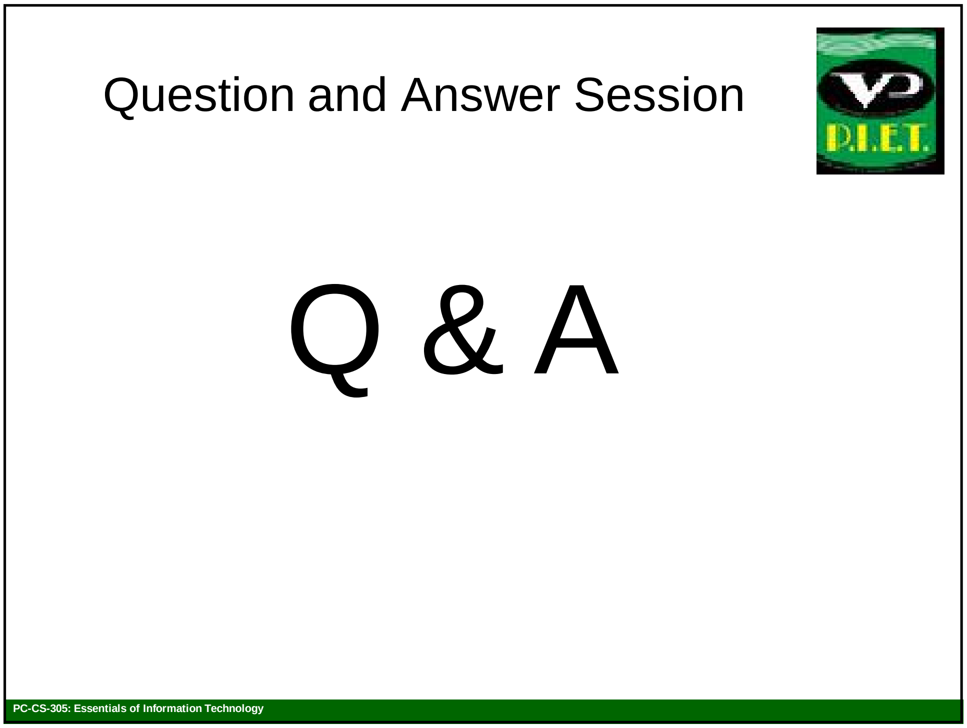
List<Object> myList = new ArrayList<Object>();
// holds ANY object type

Declaring a List with a type parameter of <Object> makes a collection that works in almost the same way as the original pre-Java 5



Summary of Main Teaching Points

PC-CS-305: Essentials of Information Technology



Question and Answer Session

Q & A

PC-CS-305: Essentials of Information Technology