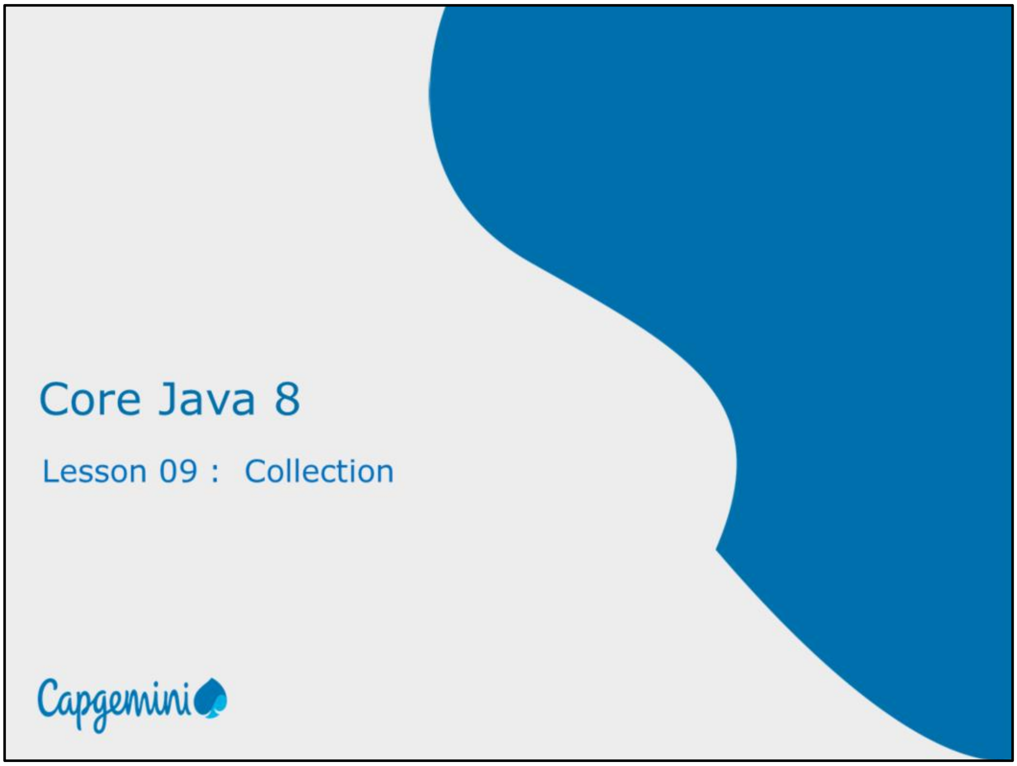

Core Java 8
Lesson 09 : Collection
Capgemini

Lesson Objectives

After completing this lesson, participants will be able to

- Understand collection framework
- Implement and use collection classes
- Iterate collections
- Create collection of user defined type
- Comparable and Comparator
- HashTable, HashMap ,TreeMap

This lesson discusses about collection framework in Java.

Lesson outline:

Collections Framework
Collection Interfaces
Implementing Classes
Iterating Collections
Comparable and Comparator
HashTable ,HashMap TreeMap
Best Practices

Overriding equals() and hashCode()

It is generally necessary to override the hashCode() method whenever equals() method is overridden, so as to maintain the general contract for the hashCode() method, which states that equal objects must have equal hash codes.

Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified.

This integer need not remain consistent from one execution of an application to another execution of the same application.

Overriding equals()

To achieve correct application behavior, we need to override equals() method as below:

```
public class Employee {
        private int empId;
        private String name;
        private double salary;
        @Override
        public boolean equals(Object obj) {
                if(obj==null)
                        return false;
                if(obj==this)
                        return true;
                Employee emp = (Employee)obj;
                return (emp.getEmpId()==this.getEmpId());
        }
```

We use the equals() method to compare objects in Java. In order to determine if two objects are the same, equals() compares the values of the objects' attributes:
In the first comparison, equals()checks to see whether the passed object is *null*, or if it's typed as a different class. If it's a different class then the objects are not equal.

In the fsecond comparison, equals() compares the current object instance with the object that has been passed. If the two objects have the same values, equals()will return true.
Finally, equals() compares the objects' fields. If two objects have the same field values, then the objects are the same.

Overriding hashCode()

Okay, so we override ***equals()*** and we get the expected behavior — even though the hash code of the two objects are different. So, what's the purpose of overriding ***hashcode()***?

It returns the hashcode value as an Integer. Hashcode value is mostly used in hashing based collections like HashMap, HashSet, HashTable....etc. This method must be overridden in every class which overrides equals() method.

Collections Framework

A Collection is a group of objects.

Collections framework provides a set of standard utility classes to manage collections.

Collections Framework consists of three parts:

- Core Interfaces
- Concrete Implementation
- Algorithms such as searching and sorting

Collections Framework:
A Collection (sometimes called a container) is an object that groups multiple elements into a single unit. Collection is used to store, retrieve objects, and to transmit them from one method to another.
The Collections API (also called the Collections framework) standardizes the way in which groups of objects are handled by your programs. It presents a set of standard utility classes to manage such collections. This framework is provided in the java.util package and comprises three main parts:

- The core interfaces, which allow collections to be manipulated independent of their implementation. These interfaces define the common functionality exhibited by collections, and facilitate data exchange between collections.
- A small set of implementations, which are concrete implementations of the core interfaces, providing data structures that a program can use. Eg LinkedLists, Arrays etc
- An assortment of algorithms, which can be used to perform various operations on collections, such as sorting & searching.

The collection classes are the fundamental building blocks of the more complicated data structures used in the other Java packages in your own applications. There are several types of collections. They vary in storage mechanisms used, in the way they access data, and in the rules about what data may be stored.
Note: The Java Collection technology is similar to the Standard Template Library (STL) defined by C++.

Advantages of Collections

Collections provide the following advantages:

- Reduces programming effort
- Increases performance
- Provides interoperability between unrelated APIs
- Reduces the effort required to learn APIs
- Reduces the effort required to design and implement APIs
- Fosters Software reuse

Collections Framework:

Advantages of Collections:

Collections provide the following advantages:

Reduces programming effort by providing useful data structures and algorithms so you do not have to write them yourself.

Increases performance by providing high-performance implementations of useful data structures and algorithms. Since the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations.

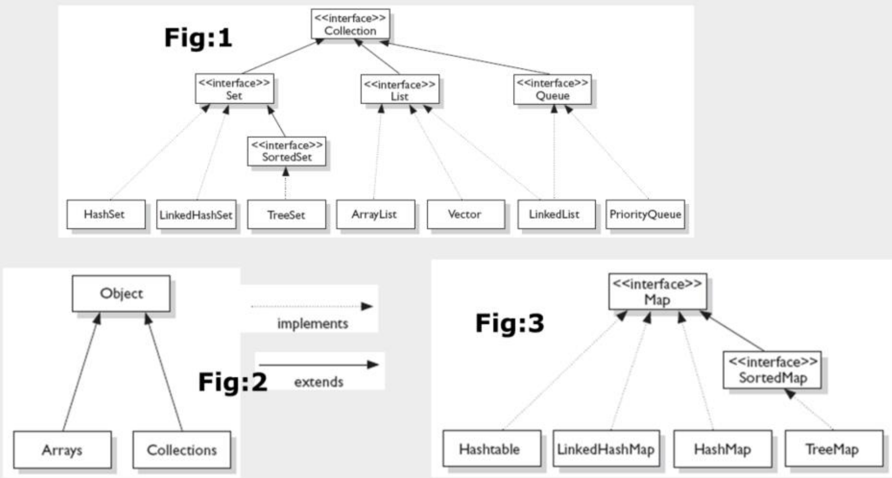
Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.

Reduces the effort required to learn APIs by eliminating the need to learn multiple ad hoc collection APIs.

Reduces the effort required to design and implement APIs by eliminating the need to produce ad hoc collections APIs.

Fosters software reuse by providing a standard interface for collections and algorithms to manipulate them.

Concept of Interfaces and Implementation



Interfaces and Implementation:
The core collection interfaces (shown in figure above) are the interfaces used to manipulate collections, and to pass them from one method to another. The basic purpose of these interfaces is to allow collections to be manipulated independently of the details of their representation.
Not all collections in the Collections Framework actually implement the Collection interface. Specifically, none of the Map-related classes and interfaces extend from Collection. So while SortedMap, Hashtable, HashMap, TreeMap, and LinkedHashMap are all thought of as collections, none are actually extended from Collection.
Note: Collections is a class, with static utility methods, while Collection is an interface with declarations of the methods common to most collections including add(), remove(), contains(), size(), and iterator().

Collection Interfaces

Let us discuss some of the collection interfaces:

Interfaces	Description
Collection	A basic interface that defines the operations that all the classes that maintain collections of objects typically implement.
Set	Extends the Collection interface for sets that maintain unique element.
SortedSet	Augments the Set interface or Sets that maintain their elements in sorted order.
List	Collections that require position-oriented operations should be created as lists. Duplicates are allowed.
Queue	Things arranged by the order in which they are to be processed.
Map	A basic interface that defines operations that classes that represent mapping of keys to values typically implement.
SortedMap	Extends the Map interface for maps that maintain their mappings in the key order.

Interfaces and Implementation:
Collection Interfaces:
Following are the four major interfaces:

Set Interface: holds only unique values and rejects duplicates.

List Interface: represents an ordered list of objects, meaning the elements of a List can be accessed in a specific order, and by an index too. List can hold duplicates.

Queue Interface: represents an ordered list of objects just like a List. However, a queue is designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue. Just like a queue in a supermarket!

Map Interface: represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. A Map cannot contain duplicate keys; each key can map to at most one value. The Map implementations let you do things like search for a value based on the key, ask for a collection of just the values, or ask for a collection of just the keys.

SortedSet Interface: is a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering.

SortedMap Interface: is a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

Collection Implementations

Collection Implementations:

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Collection Implementations:
The Java Collections Framework provides several general-purpose implementations of the Set, List, and Map interfaces. The general purpose implementations are summarized in the table above.

HashSet: is an unsorted, unordered Set. It uses the hashcode of the object being inserted, so the more efficient your hashCode() implementation is, the better access performance you will get. Use this class when you want a collection with no duplicates and you do not care about order when you iterate through it. Implements the Set interface.

LinkedHashSet: differs from HashSet by guaranteeing that the order of the elements during iteration is the same as the order they were inserted into the LinkedHashSet.

TreeSet: implements the SortedSet interface. Like LinkedHashSet, TreeSet also guarantees the order of the elements when iterated, but the order is the sorting order of the elements. This order is determined either by their natural order (if they implement Comparable), or by a specific Comparator implementation.

ArrayList: Think of this as a growable array. It gives you fast iteration and fast random access. It is an ordered collection (by index). However, it is not sorted. ArrayList now implements the new RandomAccess interface — a marker interface (meaning it has no methods) that says, “this list supports fast (generally constant time) random access.” Choose this over a LinkedList when you need fast iteration but are not as likely to be doing a lot of insertion and deletion.

LinkedList: A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another.

Collection Interface methods

Method	Description
`int size();`	Returns number of elements in collection.
`boolean isEmpty();`	Returns true if invoking collection is empty.
`boolean contains(Object element);`	Returns true if element is an element of invoking collection.
`boolean add(Object element);`	Adds element to invoking collection.
`boolean remove(Object element);`	Removes one instance of element from invoking collection
`Iterator iterator();`	Returns an iterator fro the invoking collection
`boolean containsAll(Collection c);`	Returns true if invoking collection contains all elements of *c*; false otherwise.
`boolean addAll(Collection c);`	Adds all elements of *c* to the invoking collection.
`boolean removeAll(Collection c);`	Removes all elements of *c* from the invoking collection
`boolean retainAll(Collection c);`	Removes all elements from the invoking collection except those in *c*.
`void clear();`	Removes all elements from the invoking collection
`Object[] toArray();`	Returns an array that contains all elements stored in the invoking collection
`Object[] toArray(Object a[]);`	Returns an array that contains only those collection elements whose type matches that of *a*.

Collection Interface Methods:
The Collection Interface is the foundation on which the collection framework is built. It declares the core methods that all collections will have. Some of these methods are summarized in the table given in the above slide.
The bulk operations perform some operation on an entire Collection in a single shot. They are done through the following methods, namely: containsAll(), addAll(), removeAll(), retainAll(), clear().

AutoBoxing with Collections

Boxing conversion converts primitive values to objects of corresponding wrapper types.

```
int intVal = 14;
Integer iReference = new Integer(i); // prior to Java 5,
explicit Boxing
iReference = intVal;                 // In Java
5,Automatic Boxing
```

Unboxing conversion converts objects of wrapper types to values of corresponding primitive types.

```
int intVal = iReference.intValue();   // prior to Java5,
explicit unboxing
intVal = iReference;                   // In Java 5,
Automatic Unboxing
```

AutoBoxing with Collections:
J2SE 5 adds to the Java language autoboxing and auto-unboxing.
Primitive types and their corresponding wrapper classes can now be used interchangeably. For example: The following lines of code are legitimate in Java 5:

```
int intval1 = 0;
Integer intval2 = intval1;
int intval3 = new Integer(intval2);
```

This is often referred to as automatic boxing or unboxing.
If an int is passed where an Integer is expected, then the compiler will automatically insert a call to the Integer constructor. Conversely, if an Integer is provided where an int is required, then there will be an automatic call to the IntValue method.
Autoboxing is the process by which a primitive type is automatically encapsulated into its equivalent type wrapper whenever an object of that type is needed.
Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from type wrapper when its value is needed.

Iterating through a collection

Iterator is an object that enables you to traverse through a collection.

It can be used to remove elements from the collection selectively, if desired.

```
public interface Iterator<E>
 {
 boolean hasNext();
 E next();
void remove();
 }
```

Iterable is an superinterface of Collection interface, allows to iterate the elements using foreach method

```
Collection.forEach(Consumer<? super T> action)
```

Iterators:
Java provides two interfaces that define the methods by which you can access each element of a collection: Enumeration and Iterator.

- Enumeration is a legacy interface and is considered obsolete for new code. It is now superseded by the iterator interface.
- The iterator() method of every collection returns an iterator to a collection. It is similar to an Enumeration, but differs in two respects:
  - Iterator allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
  - Method names have been improved.

There is no safe way to remove elements from a collection while traversing it with an Enumeration.
In the example in the above slide, the following parameters are used:

- boolean hasNext() : returns true if there are more elementsObject next() : It returns next element. It throws NoSuchElementException if there is no next element.
- void remove() : It removes current element. Throws IllegalStateException if an attempt is made to call remove() that is not preceded by a call to next()

Note 1: The hasNext() method is identical in function to Enumeration.hasMoreElements(), and the next() method is identical in function to Enumeration.nextElement().
Note 2: Iterator.remove() is the only safe way to modify a collection during iteration. The behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

Enhanced for loop

Iterating over collections looks cluttered:

```
void printAll(Collection<Emp> employees) {
    for (Iterator<Emp> iterator = employees.iterator();
                                   iterator.hasNext(); )
        System.out.println(iterator.next()); } }
```

Using enhanced for loop, we can do the same thing as:

```
void printAll(Collection<Emp> employees) {
    for (Emp empObj : employees) )
        System.out.println( empObj ); }}
```

- When you see the colon (:) read it as "in."
- The loop above reads as "for each emp 't' in collection 'e'."

Enhanced for loop:
The enhanced for loop can be used for both Arrays and Collections:

```
class Enhancedforloop {
  static void printArray(int intArr[]) {
        for (int arrayindex : intArr )
            System.out.println(arrayindex);
  }
  static void printCollection(ArrayList arrList) {
      for (Object object : arrList)
          System.out.println(object);
  }

public static void main(String arg[]) {
    int intArr[] = { 1, 2, 3, 4, 5 };
    printArray(intArr);
    ArrayList arraylist = new ArrayList();
    arraylist.add(10);
    arraylist.add(30);
    arraylist.add(20);
    printCollection(arraylist);
  }
}
```

Demo :Concept of Iterators

Execute:

- MailList.java
- ItTest.java program

ArrayList Class

An ArrayList Class can grow dynamically.

It provides more powerful insertion and search mechanisms than arrays.

It gives faster Iteration and fast random access.

It uses Ordered Collection (by index), but not Sorted.

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

ArrayList Class:
Let us check the power of ArrayList with an example:

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

In many ways, ArrayList<String> is similar to a String[] in that it declares a container that can hold only Strings. However, it is more powerful than a String[]. Let us look at some of the capabilities that an ArrayList has:

```
import java.util.*;
    public class ArrayListTest {
      public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        String str = "hi";
        list.add("string");
        list.add(str);
        list.add(str + str);
        System.out.println(list.size());
        System.out.println(list.contains(42));
        System.out.println(list.contains("hihi"));
        list.remove("hi");
        System.out.println(list.size());
      }
    }
```

```
output :
3
false
true
2
```

Vector Class

Vector implements a dynamic array. It is similar to ArrayList, but with two differences –

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

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 over the lifetime of a program.

ArrayList Class:
Let us check the power of ArrayList with an example:

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

In many ways, ArrayList<String> is similar to a String[] in that it declares a container that can hold only Strings. However, it is more powerful than a String[]. Let us look at some of the capabilities that an ArrayList has:

```
import java.util.*;
    public class ArrayListTest {
      public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        String str = "hi";
        list.add("string");
        list.add(str);
        list.add(str + str);
        System.out.println(list.size());
        System.out.println(list.contains(42));
        System.out.println(list.contains("hihi"));
        list.remove("hi");
        System.out.println(list.size());
      }
    }
```

```
output :
3
false
true
2
```

Demo: Vector Class

Execute the VectorDemo.java program

HashSet Class

HashSet Class does not allow duplicates.

A HashSet is an unsorted, unordered Set.

It can be used when you want a collection with no duplicates and you do not care about the order when you iterate through it.

HashSet Class:
Remember that Sets are used when you do not want any duplicates in your collection. If you attempt to add an element to a set that already exists in the set, then the duplicate element will not be added, and the add() method will return false. Remember, HashSets tend to be very fast because they use hashcodes.

```
import java.util.*;
    class SetTest {
      public static void main(String[] args) {
        boolean[] boolArr = new boolean[5];
        Set<Integer> set = new HashSet<Integer>();
        boolArr[0] = set.add(1);
        boolArr[1] = set.add(2);
        boolArr[2] = set.add(3);
        boolArr[3] = set.add(4);
        boolArr[4] = set.add(5);
        for (Integer index : set)
            System.out.print(index + " ");
        }
    }
```

O/P: 2 4 1 3 5

Note: The order of the objects printed are not predictable

Demo: Hash Set Class

Execute the HashSetDemo.java program


```
import java.util.*;
class HashSetDemo {
  public static void main(String args[]) {
  // create a hash set
     HashSet hs = new HashSet();
     // add elements to the hash set
        hs.add("B");
        hs.add("A");
        hs.add("D");
        hs.add("E");
        hs.add("C");
        hs.add("F");
       System.out.println(hs);
    }
}
```

Output :
[D, A, F, C, B, E]

TreeSet class

TreeSet does not allow duplicates.

It iterates in sorted order.

Sorted Collection:

- By default elements will be in ascending order.

Not synchronized:

- If more than one thread wants to access it at the same time, then it must be synchronized externally.

TreeSet:
TreeSet implements the Set interface, backed by a TreeMap instance. This class guarantees that the sorted set will be in ascending element order, sorted according to the natural order of the elements, or by the comparator provided at set creation time, depending on which constructor is used.

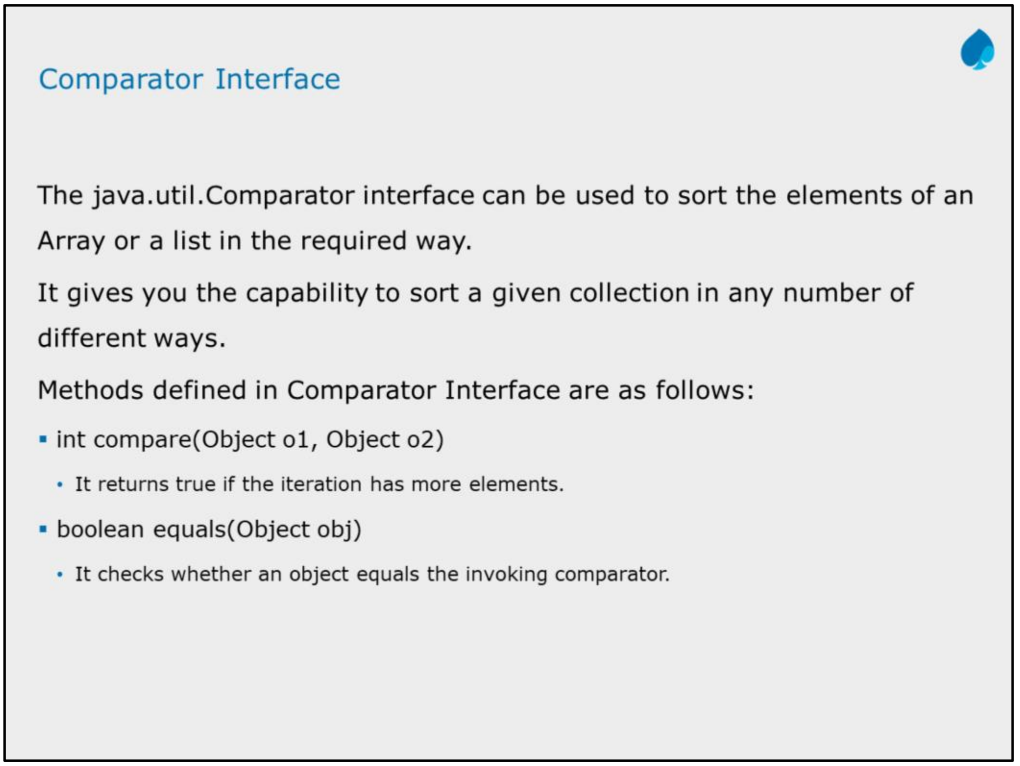
```
class TreeSetDemo {
 public static void main(String args[]) {
   TreeSet<String> treeSet = new TreeSet<String>();
   treeSet.add("One");
   treeSet.add("Two");
   treeSet.add("Three");
   treeSet.add("Four");
   treeSet.add("Five");
         System.out.println("Contents of treeset");
   Iterator iterator = treeSet.iterator(); // obtaining iterator object
   while (iterator.hasNext()) { // to iterate thru collection.
        Object object = iterator.next();
         System.out.print(object + "\t");
    }
  }}
```

O/P: Five Four One Three Two

Demo: Tree Set class

Execute the Treeset.java program

You can also refer to the



Comparator Interface:

The Comparator interface defines two methods: compare( ) and equals( ).

The compare( ) method, shown here, compares two elements for order:

int compare(Object obj1, Object obj2)

obj1 and obj2 are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned. The method can throw a ClassCastException if the types of the objects are not compatible for comparison.

By overriding compare( ), you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

The equals( ) method, shown here, tests whether an object equals the invoking comparator:

boolean equals(Object obj)

obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false. Overriding equals( ) is unnecessary, and most simple comparators will not do so.

Comparable Interface

Java.util.Comparable interface imposes a total ordering on the objects of each class that implements it.

This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.

Methods defined in Comparable Interface are as follows:

- public int compareTo(Object o)
- Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Comparable Interface:
This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

public int compareTo(Object o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Comparable Interface Example

```
class Emp implements Comparable {
    int empID;
    String empName;
    double empSal;
     public Emp(String ename, double sal) { ...  }
     public String toString() {  ...  }

      public int compareTo(Object o) {
          if (this.empSal == ((Emp) o).empSal)     return 0;
            else if (this.empSal > ((Emp) o).empSal)    return 1;
               else   return -1;
    }}
```

Comparable Interface Example (ctnd…)

```
class Comparable Demo {
      public static void main(String[] args) {
        TreeSet tset = new TreeSet();
        tset.add(new Emp("harry", 40000.00));
        tset.add(new Emp("Mary", 20000.00));
        tset.add(new Emp("Peter", 50000.00));

        Iterator iterator = tset.iterator();
        while (iterator.hasNext()) {
          Object empObj = iterator.next();
          System.out.println(empObj + "\n");
        }
}}
```

Output:

```
Ename : Mary    Sal : 20000.0
Ename : harry   Sal : 40000.0
Ename : Peter   Sal : 50000.0
```

Demo : Concept of Comparator & Comparable Interface

Execute:

- ComparatorExample.java
- ComparableDemo.java

HashMap Class

HashMap uses the hashcode value of an object to determine how the object should be stored in the collection.

Hashcode is used again to help locate the object in the collection.

HashMap gives you an unsorted and unordered Map.

It allows one null key and multiple null values in a collection.

HashMap Class:
Map is an object that stores key/value pairs. Given a key, you can find its value. Keys must be unique, values may be duplicated. The HashMap class implements the map interface. The HashMap class uses a hash table to implement Map interface.
The following example maps names to account balances.

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
HashMap<String,Double> hm = new HashMap<String,Double>();
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
Set set = hm.entrySet();     // Get a set of the entries
Iterator i = set.iterator();     // Get an iterator
    while(i.hasNext()) {          // Display elements
        Map.Entry me = (Map.Entry)i.next();
        System.out.println(me.getKey() + ": "+ me.getValue());
    }
// Deposit 1000 into John Doe's account
double balance =  ((Double)hm.get("John Doe")).doubleValue();
hm.put("John Doe", new Double(balance + 1000));
System.out.println("John Doe's new balance: " + hm.get("John Doe")); } }
```

Internal Working of HashMap

HashMap works on the principal of hashing.

HashMap uses the hashCode() method to calculate a hash value. Hash value is calculated using the key object. This hash value is used to find the correct bucket where Entry object will be stored.

HashMap uses the equals() method to find the correct key whose value is to be retrieved in case of get() and to find if that key already exists or not in case of put().

Hashing collision means more than one key having the same hash value, in that case Entry objects are stored as a linked-list with in a same bucket.

With in a bucket values are stored as Entry objects which contain both key and value.

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
HashMap<String,Double> hm = new HashMap<String,Double>();
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
Set set = hm.entrySet();     // Get a set of the entries
Iterator i = set.iterator();     // Get an iterator
    while(i.hasNext()) {           // Display elements
        Map.Entry me = (Map.Entry)i.next();
        System.out.println(me.getKey() + ": "+ me.getValue());
    }
// Deposit 1000 into John Doe's account
double balance =  ((Double)hm.get("John Doe")).doubleValue();
hm.put("John Doe", new Double(balance + 1000));
System.out.println("John Doe's new balance: " + hm.get("John Doe")); } }
```

Internal Working of HashMap

```
Class Key {
int index ;
String name;
Key(int index,String name){
this.index=index;
this.name=name;
}
@Override
public int hashCode(){
return 5;
}
@Override
public boolean equals (Object obj){
return true;
}
}
```

```
public class Demo{
public static void main(String args[]){
 Map <Key, String> cityMap = new HashMap<Key, String>();
      cityMap.put(new Key(1, "NY"),"New York City" );
      cityMap.put(new Key(2, "ND"), "New Delhi");
      cityMap.put(new Key(3, "NW"), "Newark");
      cityMap.put(new Key(4, "NP"), "Newport");
      System.out.println("size :"+ cityMap.size());
      Iterator <Key> itr = cityMap.keySet().iterator();
      while (itr.hasNext()){
         System.out.println(cityMap.get(itr.next()));
      }
      System.out.println("size after iteration " + cityMap.size());
}
}
```

Output :
Size before iteration : 1
Newport
Size after iteration : 1

Demo: HashMap Class

Execute the HashMapDemo.java program

The example is provided on previous page.
The output of the program is as follows:
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0
John Doe’s new balance: 4434.34
The above program first populates the HashMap object. Then the contents of the map are displayed using a set-view, obtained by calling entrySet(). The keys and values are displayed by calling getKey() and getValue() methods of the Map.Entry interface.

Note: TreeMap instead of HashMap will have given a sorted output.

Hashtable Class

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary

Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.

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.

Hashtable Class:
The Hashtable was a part of the original java.util package.
Hashtable is synchronized, and stores a key/value pair using the hashing technique. While 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. Subsequently, the resulting hash code is used as the index at which the value is stored within the table. The Hashtable class only stores objects that override the hashCode() and equals() methods that are defined by Object.

Demo: Hash table Class

Execute the HashTableDemo.java program


```
import java.util.*;
class HashTableDemo {
   public static void main(String args[]) {
        Hashtable<String,Double> balance = new
                                                  Hashtable<String,Double>();
        Enumeration names;
        String str;
        double bal;
        balance.put("Arun", new Double(3434.34));
        balance.put("Radha", new Double(123.22));
        balance.put("Ram", new Double(99.22));
        // Show all balances in hash table.
       names = balance.keys();
        while(names.hasMoreElements()) {
             str = (String) names.nextElement();
             System.out.println(str + ": " +
             balance.get(str));
        }
      // Deposit 1,000 into Zara's account
          bal = ((Double)balance.get("Ram")).doubleValue();
          balance.put("Ram", new Double(bal+1000));
          System.out.println("Ram's new balance: " +
          balance.get("Ram"));
   }
}
```

TreeMap Class

The TreeMap class implements the Map interface by using a tree. A TreeMap provides an efficient means of storing key/value pairs in sorted order, and allows rapid retrieval.

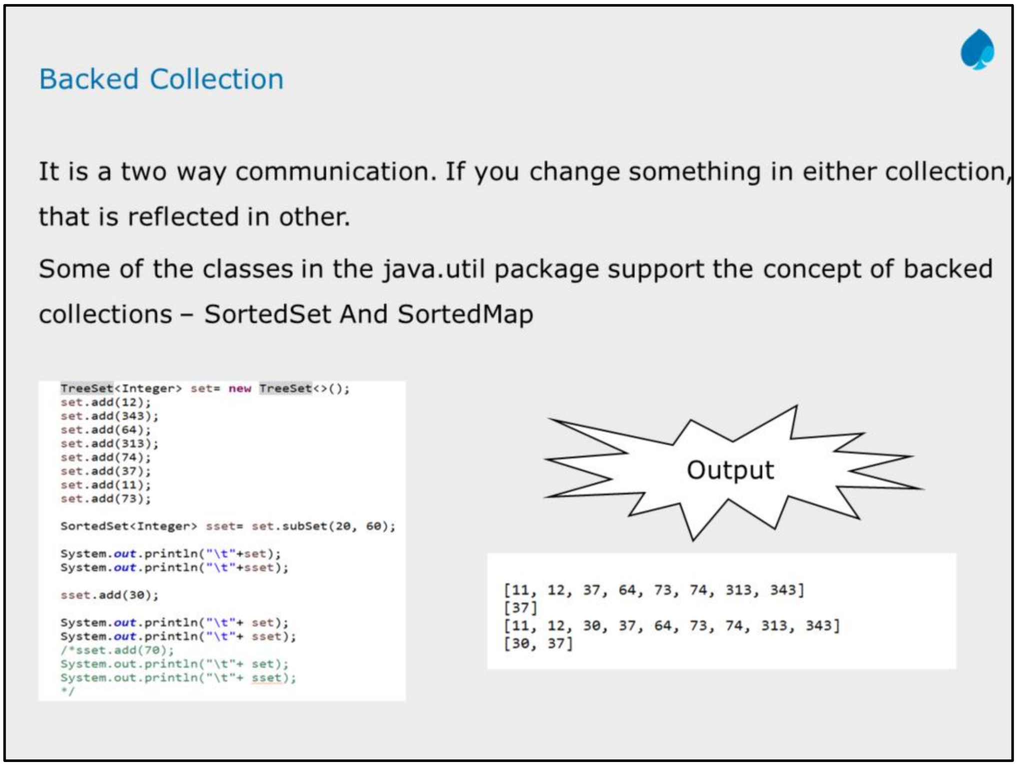
You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in an ascending key order.

HashMap Class:
Map is an object that stores key/value pairs. Given a key, you can find its value. Keys must be unique, values may be duplicated. The HashMap class implements the map interface. The HashMap class uses a hash table to implement Map interface.
The following example maps names to account balances.

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
HashMap<String,Double> hm = new HashMap<String,Double>();
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
Set set = hm.entrySet();     // Get a set of the entries
Iterator i = set.iterator();     // Get an iterator
    while(i.hasNext()) {          // Display elements
        Map.Entry me = (Map.Entry)i.next();
        System.out.println(me.getKey() + ": "+ me.getValue());
    }
// Deposit 1000 into John Doe's account
double balance =  ((Double)hm.get("John Doe")).doubleValue();
hm.put("John Doe", new Double(balance + 1000));
System.out.println("John Doe's new balance: " + hm.get("John Doe")); } }
```

Demo: TreeMap Class

Execute the TreeMapDemo.java program



The important method in this code is the TreeMap.subMap() method. It's easy to guess (and it's correct), that the subMap() method is making a copy of a portion of the TreeMap named map. The first line of output verifies the conclusions we've just drawn.

What happens next is powerful and a little bit unexpected. When we add key-value pairs to either the original TreeSet or the partial-copy SortedSet, the new entries were automatically added to the other collection—sometimes. When subset was created, we provided a value range for the new collection. This range defines not only what should be included when the partial copy is created, but also defines the range of values that can be added to the copy. As we can verify by looking at the second line of output, we can add new entries to either collection within the range of the copy, and the new entries will show up in both collections. In addition, we can add a new entry to the original collection, even if it's outside the range of the copy. In this case, the new entry will show up only in the original—it won't be added to the copy because it's outside the copy's range. Notice that we commented out line If you attempt to add an out-of-range entry to the copied collection an exception will be thrown.

we can see a lot of parallels between the TreeSet and the TreeMap methods. The headSet() / headMap() methods create a subset that starts at the beginning of the original collection and ends at the point specified by the method's argument. The tailSet() / tailMap() methods create a subset that starts at the point specified by the method's argument and goes to the end of the original collection. Finally, the subSet() / subMap() methods allow you to specify both the start and end points for the subset collection you're creating.

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
HashMap<String,Double> hm = new HashMap<String,Double>();
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
Set set = hm.entrySet();     // Get a set of the entries
Iterator i = set.iterator();     // Get an iterator
   while(i.hasNext()) {          // Display elements
     Map.Entry me = (Map.Entry)i.next();
     System.out.println(me.getKey() + ": " + me.getValue());
   }
// Deposit 1000 into John Doe's account
double balance =  ((Double)hm.get("John Doe")).doubleValue();
hm.put("John Doe", new Double(balance + 1000));
System.out.println("John Doe's new balance: " + hm.get("John Doe")); } }
```

Generics

Generics is a mechanism by which a single piece of code can manipulate many different data types without explicitly having a separate entity for each data type.

What and Why of Generics:
JDK 1.5 introduces several extensions to the Java programming language. One of these is generics. Generics allow you to abstract over types. The most common examples are container types, such as those in the Collection hierarchy.
Here is a typical usage of that sort:

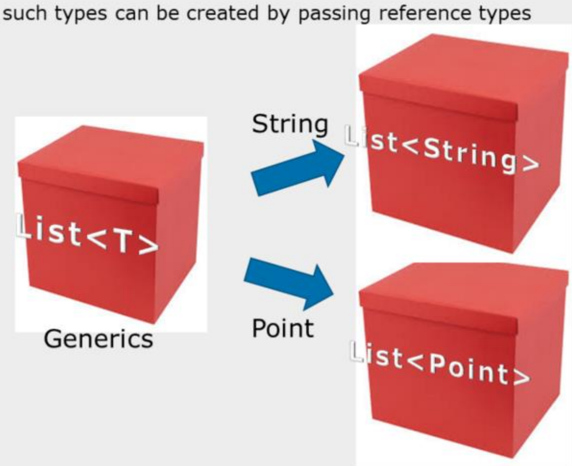
```
List myIntegerList = new LinkedList(); // 1
myIntegerList.add(new Integer(0)); // 2
Integer intObj = (Integer) myIntegerList.iterator().next(); // 3
```

The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an Object will be returned by the iterator. To ensure the assignment to a variable of type Integer is type safe, the cast is required. Of course, the cast not only introduces clutter, it also introduces the possibility of a run time error, since the programmer might be mistaken. What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics.

Generics

Generics allows programmer to create parameterized types

Instances of such types can be created by passing reference types



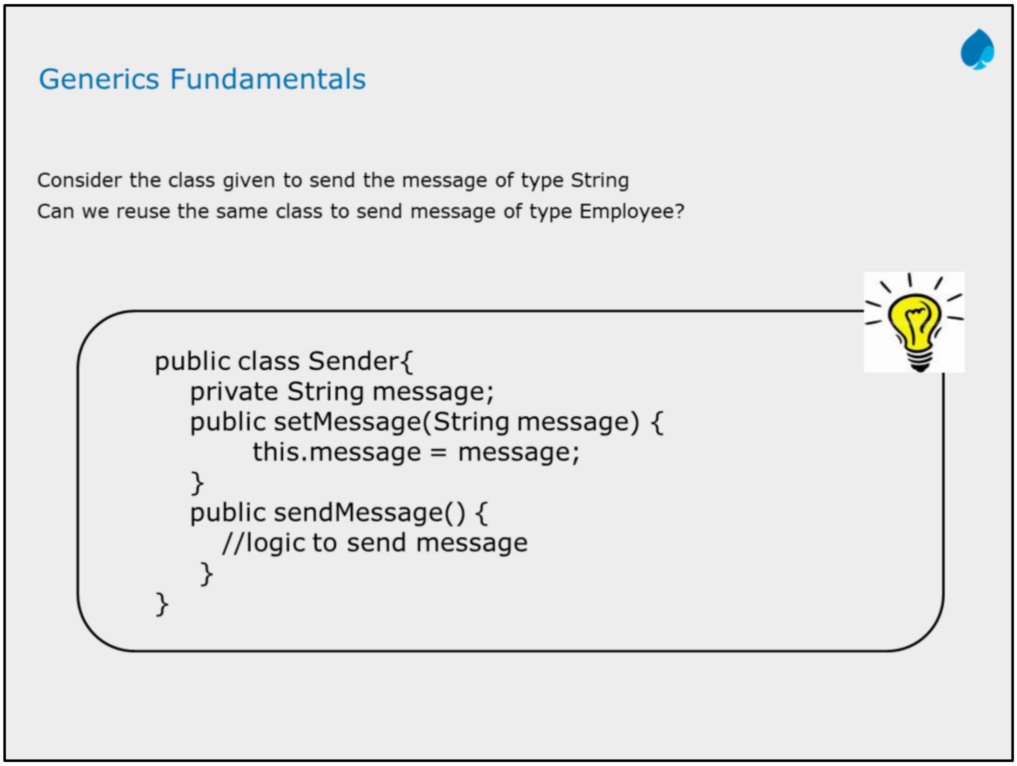
What is Generics?:

Generics allows to write Parameterized type like List<T> and allows us to pass references to create instance of that type. Like when we pass the reference of String to List<T> as a reference type it creates List of strings.

Why generics?

Use of generics enables stricter compiler check. It means when List is created of type string, compiler only allows to add string elements to the list.

No need to perform casting. In case of generics enabled collections, no need to perform explicit casting.



Generics Fundamentals:

Consider the example given in the slide, if you want to reuse the same class to send employee object as a message, then we need to create one more additional class as shown below:

```
public class Sender{
  private Employee message;
  public setMessage(Employee message) {
      this.message = message;
  }
  public sendMessage() {
          //logic to send message
   }
}
```

Is it possible to create a class with generic type of message? Yes. Generics enables us to create a parameterized class and later we can instantiate it as per our required reference type.

Writing Generic Types

How to create a sender class to send generic type of message?

```
public class Sender<T>{
    private T message;
    public setMessage(T message) {
        this.message = message;
    }
    public sendMessage() {
      //logic to send message
     }
}
```

```
Sender<String> stringSender = new Sender<String>();
Sender<Employee> empSender = new Sender<Employee>();
```

Generics Fundamentals:

As shown in the slide example, The sender class is declared as parameterized generic type of one parameter as “T”. The “T” in diamond operator refers as generic type of message.

In case of String message sender, the class instance would be initialized as:

```
Sender<String> stringSender = new Sender<String>() ;
```

In the above instance creation, the type T is replaced by String reference type. The same generic sender can be used to send message of type employee as shown below:

```
Sender<Employee> stringSender = new Sender<Employee>() ;
```

Generics Terminology

Below listed are different conventions used in generics

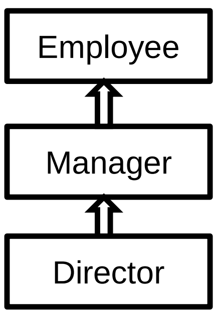
Syntax	Meaning
<T>	T denotes instance of any reference type
<?>	? denotes object of any type
<? super T>	? denotes lower bound object of type T
<? extends T>	? denotes upper bound object of type T (class)
<K, V>	K and V denotes instance of any type (same as T)

Generics Terminology:

In generics, different conventions are used to indicate the applicable reference type. For example, class Sender<T> indicates, the allowed reference type to create instance of Sender are:
Any reference type T
Subclass of T

The wildcard ? is used to indicate any type. There are two variations in using wildcard.
? super T: indicates lower bound meaning, any reference types which are superclass of T are allowed.
? extends T: indicated upper bound meaning, any reference types which are subclass of T are allowed.

Consider the given example for inheritance relationship.



List<? super Manager> means, list can be created of Manager, Employee etc. That is all superclass's of Manager.

List<? extends Manager> means, list can be created of Manager, Director etc. That is all subclasses of Manager.

Using Generics with Collections

- Before Generics:

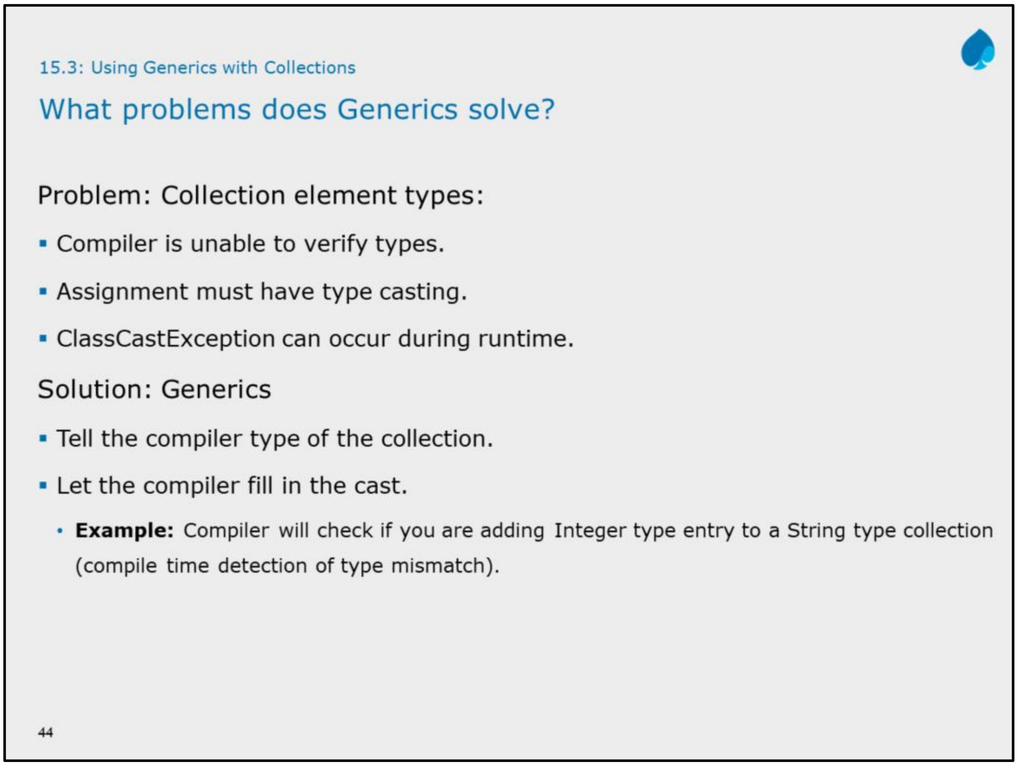
```
List myIntegerList = new LinkedList(); // 1
myIntegerList.add(new Integer(0)); // 2
Integer intObj = (Integer) myIntegerList.iterator().next(); // 3
```

Note: Line no 3 if not properly typecasted will throw runtime exception

- After Generics:

```
List<Integer> myIntegerList = new LinkedList<Integer>(); // 1
myIntegerList.add(new Integer(0)); //2
Integer intObj = myIntegerList.iterator().next(); // 3
```

What and Why of Generics:
Observe the program fragment given above (in box #2) using generics:
Notice the type declaration for the variable myIntegerList. It specifies that this is not just an arbitrary List, but a List of Integer, written as List<Integer>. We say that List is a generic interface that takes a type parameter - in this case, Integer. We also specify a type parameter while creating the list object. Notice that the cast is gone from line 3. It may seem that all that's accomplished is just moving the clutter around. Instead of a cast to Integer on line 3, we have Integer as a type parameter on line 1.
However, there is a very big difference here. The compiler can now check the type correctness of the program at compile-time. When we say that myIntegerList is declared with type List<Integer>, this tells us something about the variable myIntegerList, which holds true wherever and whenever it is used, and the compiler will guarantee it. In contrast, the cast tells us something the programmer thinks is true at a single point in the code. The net effect, especially in large programs, is improved readability and robustness.



What and Why of Generics:
What problems does Generics solve?
Type cast not being checked at compile-time leads to a major problem occurring at application’s runtime.
Biggest achievement of the Generics is to avoid the runtime exceptions.

15.3: Using Generics with Collections

Using Generic Classes: 1

You can instantiate a generic class to create type specific object.

In J2SE 5.0, all collection classes are rewritten to be generic classes.

- Example:

```
Vector<String> vector = new Vector<String>();
vector.add(new Integer(5)); // Compile error!
vector.add(new String("hello"));
String string = vector.get(0); // No casting needed
```

45

Usage of Generics:
Usage of Generic classes is pertaining to the type that is required.
Once the Generic class is used for specific type, compile-time and runtime errors can be avoided.

15.3: Using Generics with Collections

Using Generic Classes: 2

Generic class can have multiple type parameters.

Type argument can be a custom type.

- Example:

```
HashMap<String, Mammal> map =
                    new HashMap<String, Mammal>();
map.put("wombat", new Mammal("wombat"));
Mammal mammal = map.get("wombat");
```

46

Usage of Generics:
Generic classes in use can have multiple arguments. Arguments can be standard as well as custom types.

15.3: Using Generics with Collections

Generics

Using generics, you can do this:

Object object = new Integer(5);

You can even do this:

Object[] objArr = new Integer[5];

So you would expect to be able to do this: ArrayList<Object> arraylist = new ArrayList<Integer>();

But you can't do it!!

- This is counter-intuitive at the first glance.

47

Note: Generic classes cannot be assigned according to the super or subclass hierarchy of them.

15.3: Using Generics with Collections

Generics

Why does this compile error occur?

- It is because if it is allowed, ClassCastException can occur during runtime – this is not type-safe.

```
ArrayList<Integer> ai = new ArrayList<Integer>();
ArrayList<Object> ao = ai;   // If it is allowed at compile time,
ao.add(new Object());Integer i = ao.get(0);   // will result in runtime
 ClassCastException
```

There is no inheritance relationship between type arguments of a generic class.

48

Note: Generic classes do not support inheritance relationship between type arguments.

15.3: Using Generics with Collections

Generics

The following code works:

```
ArrayList<Integer> ai = new ArrayList<Integer>();
List<Integer> li = new ArrayList<Integer>();
Collection<Integer> ci = new ArrayList<Integer>();
Collection<String> cs = new Vector<String>(4);
```

Inheritance relationship between Generic classes themselves still exists.

49

Note: Although the inheritance relationship between the type arguments of the generic classes does not exist, Inheritance relationship between Generic classes themselves still exist.

15.3: Using Generics with Collections

Generics

The following code works:

```
ArrayList<Number> an = new ArrayList<Number>();
an.add(new Integer(5));
an.add(new Long(1000L));
an.add(new String("hello")); // compile error
```

The entries maintain inheritance relationship.

50

Lab

Lab Arrays and Collections

14.8: Common Best Practices on Collections

Best Practices

Let us discuss some of the best practices on Collections:

- Use for-each liberally.
- Presize collection objects.
- Note that Vector and HashTable is costly.
- Note that LinkedList is the worst performer.

Common Best Practices on Collections:
Use for-each liberally : When there is a choice, the for-each loop should be preferred over the for loop, since it increases legibility.
Presize collection objects.

This is necessary because whenever the collection size has reached the maximum, internally whole array is copied to a new array with new increased size. This takes considerable time.

Try to presize any collection object to be as big as it will need to be. It is better for the object to be slightly bigger than necessary than to be smaller. This recommendation really applies to collections that implement size increases in such a way that objects are discarded.

For example: Vector grows by creating a new larger internal array object, copying all the elements from and discarding the old array. Most collection implementations work similarly, so presizing a collection to its largest potential size reduces the number of objects discarded.

Vector and HashTable is costly.

Usage of vector is very costly especially in code which heavily uses Vector to store lots of elements. Avoid using that if the elements in it are of same type. This is because elements are stored as Object so while accessing them one has to cast them into relevant classes which is very costly. Use ArrayList instead.

HashTable has the same reason as in the case of Vector. Moreover, the problem is compounded because of the use of Key and Value. Use HashMap class instead.

Never use linked List while accessing the objects : Sequentially access the elements.

14.8: Common Best Practices on Collections

Best Practices

- Choose the right Collection.
- Note that adding objects at the beginning of the collections is considerably slower than adding at the end.
- Encapsulate collections.
- Use thread safe collections when needed.

Common Best Practices on Collections:

Choosing the right Collection:

We can select the appropriate collection based on the different implementation of the collection interfaces. As we know, there are different collection classes available such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, and so on.

Principal features of non-primary implementations:

HashMap has slightly better performance than LinkedHashMap. However, its iteration order is undefined.

HashSet has slightly better performance than LinkedHashSet. However its iteration order is undefined.

TreeSet is ordered and sorted, but slow.

TreeMap is ordered and sorted, but slow.

LinkedList has fast adding to the start of the list, and fast deletion from the interior via iteration.

Summary

The various Collection classes and Interfaces

Generics

Best practices in Collections

Review Questions

Question 1: Consider the following code:

```
TreeSet map = new TreeSet();
   map.add("one");
   map.add("two");
   map.add("three");
   map.add("one");
   map.add("four");
   Iterator it = map.iterator();
   while (it.hasNext() )
       System.out.print( it.next() + " " );
```

- **Option 1:** Compilation fails
- **Option 2:** four three two one
- **Option 3:** one two three four
- **Option 4:** four one three two

Review Questions

Question 2: Which of the following statements are true for the given code?

```
public static void before() {
  Set set = new TreeSet();
  set.add("2");
  set.add(3);
  set.add("1");
  Iterator it = set.iterator();
    while (it.hasNext())
        System.out.print(it.next() + " ");
}
```

- **Option 1:** The before() method will print 1 2
- **Option 2:** The before() method will print 1 2 3
- **Option 3:** The before() method will not compile.
- **Option 4:** The before() method will throw an exception at runtime.