Module 9

Collections and Generics Framework

Objectives

- Describe the core interfaces in the **Collectionsframework**
- *Set*,*List* interface
- *Map* interface
- Generic collections, Use type parameters in generic classes
- System *Properties*
- Create natural and custom ordering by implementing the *Comparable* and *Comparator* interfaces
- Refactor existing non-generic code
- Iterate over a collection

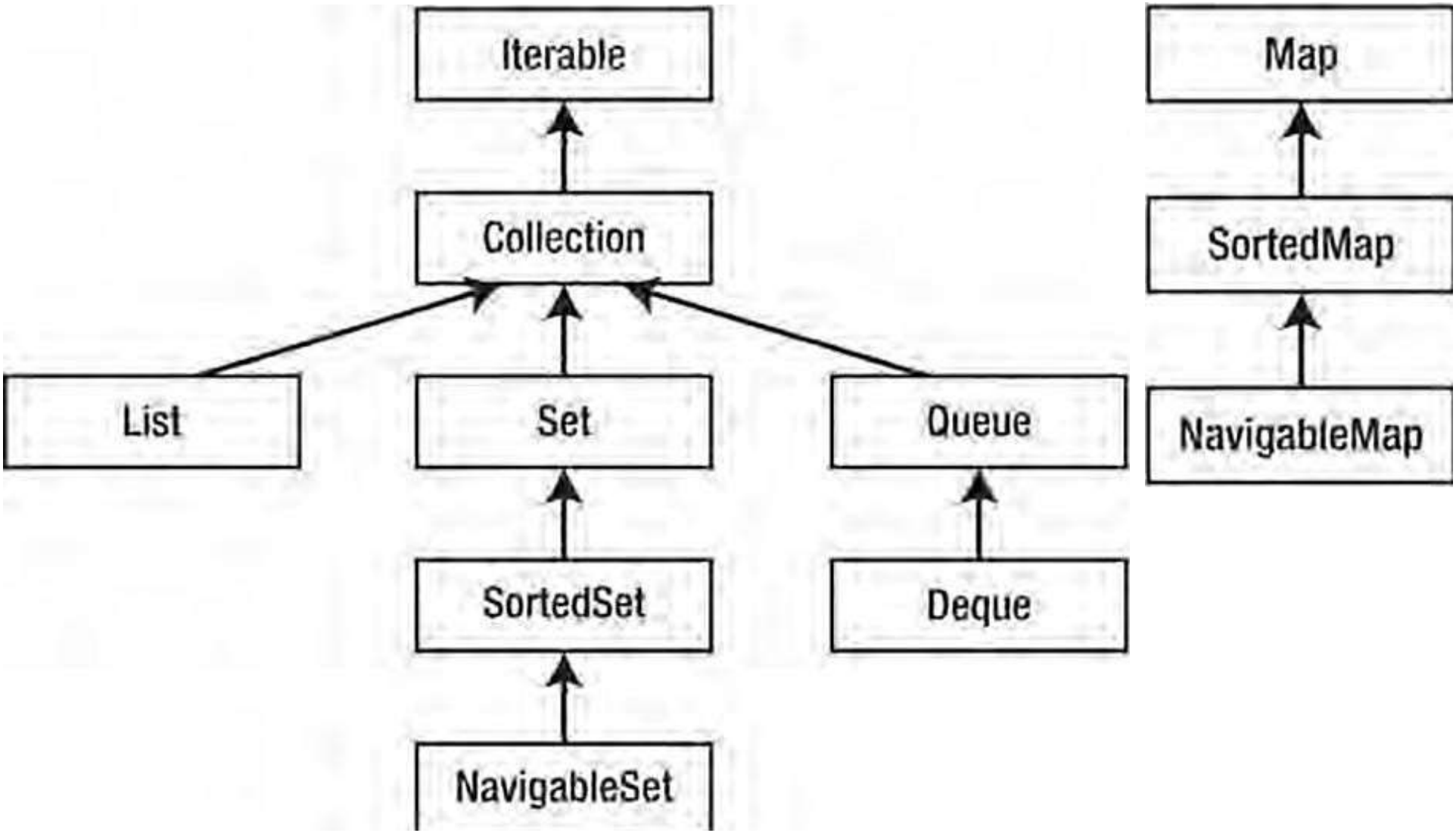
The Collections Framework

Collection Framework's architecture

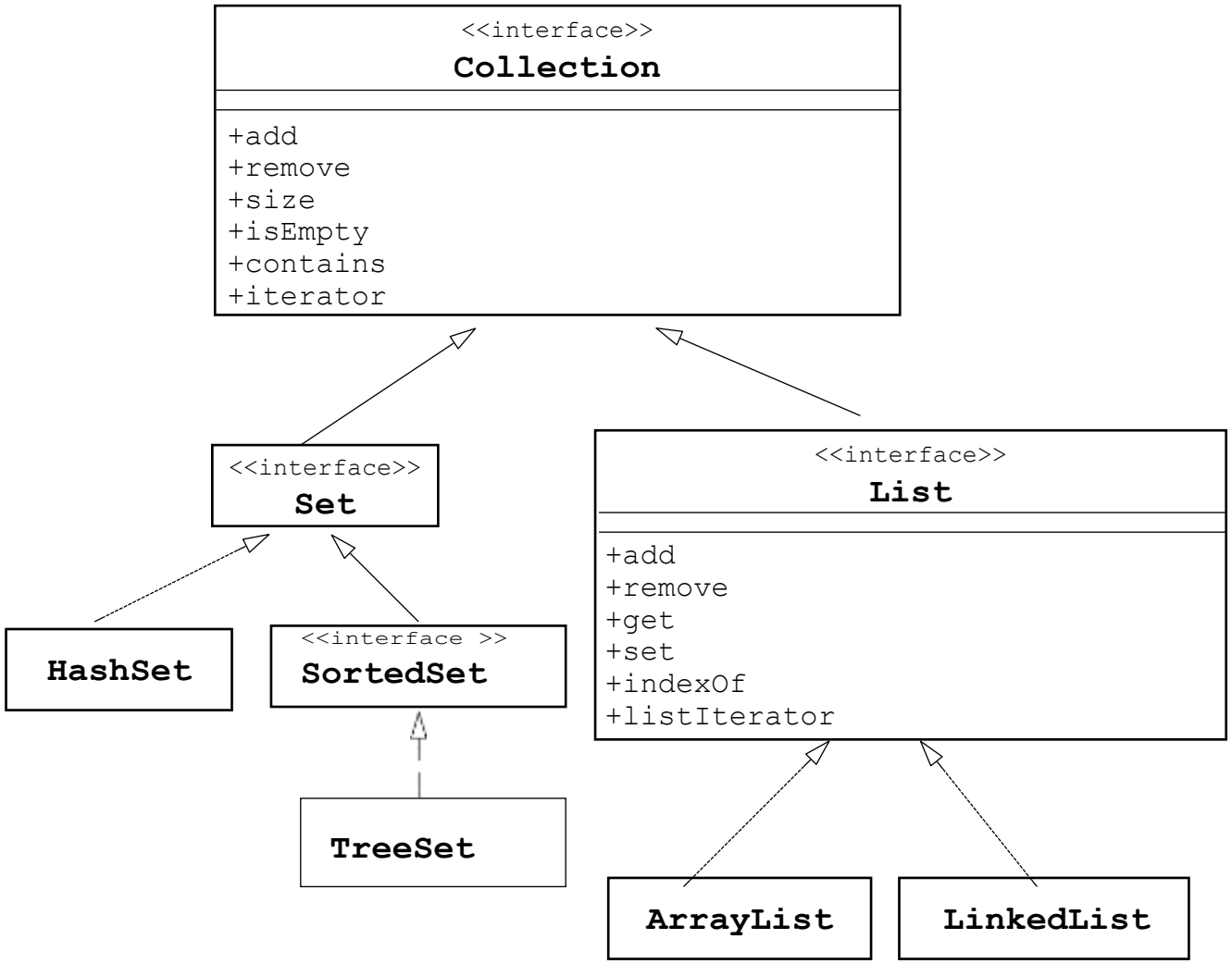
- Collections Framework is a standard architecture for representing and manipulating collections
- The Collection Framework's architecture is divided into three sections:
  - *Core interfaces*: for manipulating collections.
  - *Implementation classes*: provide different core interface implementations
  - *Utility classes*: utility classes methods let you sort arrays, obtain synchronized collections, and perform other operations

The Collections Framework

- A *collection* is a single object managing a group of objects known as its elements.
- The Collections API contains interfaces that group objects as one of the following:
  - `Collection` – A group of objects called elements; implementations determine whether there is specific ordering and whether duplicates are permitted.
  - `Set` – An unordered collection; no duplicates are permitted.
  - `List` – An ordered collection; duplicates are permitted.



The Collections API



Collection Implementations

There are several general purpose implementations of the core interfaces (`Set`, `List`, `Deque` and `Map`)

	Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**	**Hash Table + Linked List**
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

A `Set` Example

```
1   import java.util.*;
2   public class SetExample {
3     public static void main(String[] args)
4       { Set set = new HashSet();
5       set.add("one");
6       set.add("second");
7       set.add("3rd");
8       set.add(new Integer(4));
9       set.add(new Float(5.0F));
10      set.add("second");          // duplicate, not added
11      set.add(new Integer(4));    // duplicate, not added
12      System.out.println(set);
13    }
14  }
```

The output generated from this program is:

```
[one, second, 5.0, 3rd, 4]
```

A `List` Example

```
1   import java.util.*
2   public class ListExample {
3     public static void main(String[] args)
4       { List list = new ArrayList();
5       list.add("one");
6       list.add("second");
7       list.add("3rd");
8       list.add(new Integer(4));
9       list.add(new Float(5.0F));
10      list.add("second");          // duplicate, is added
11      list.add(new Integer(4));    // duplicate, is added
12      System.out.println(list);
13    }
14  }
```

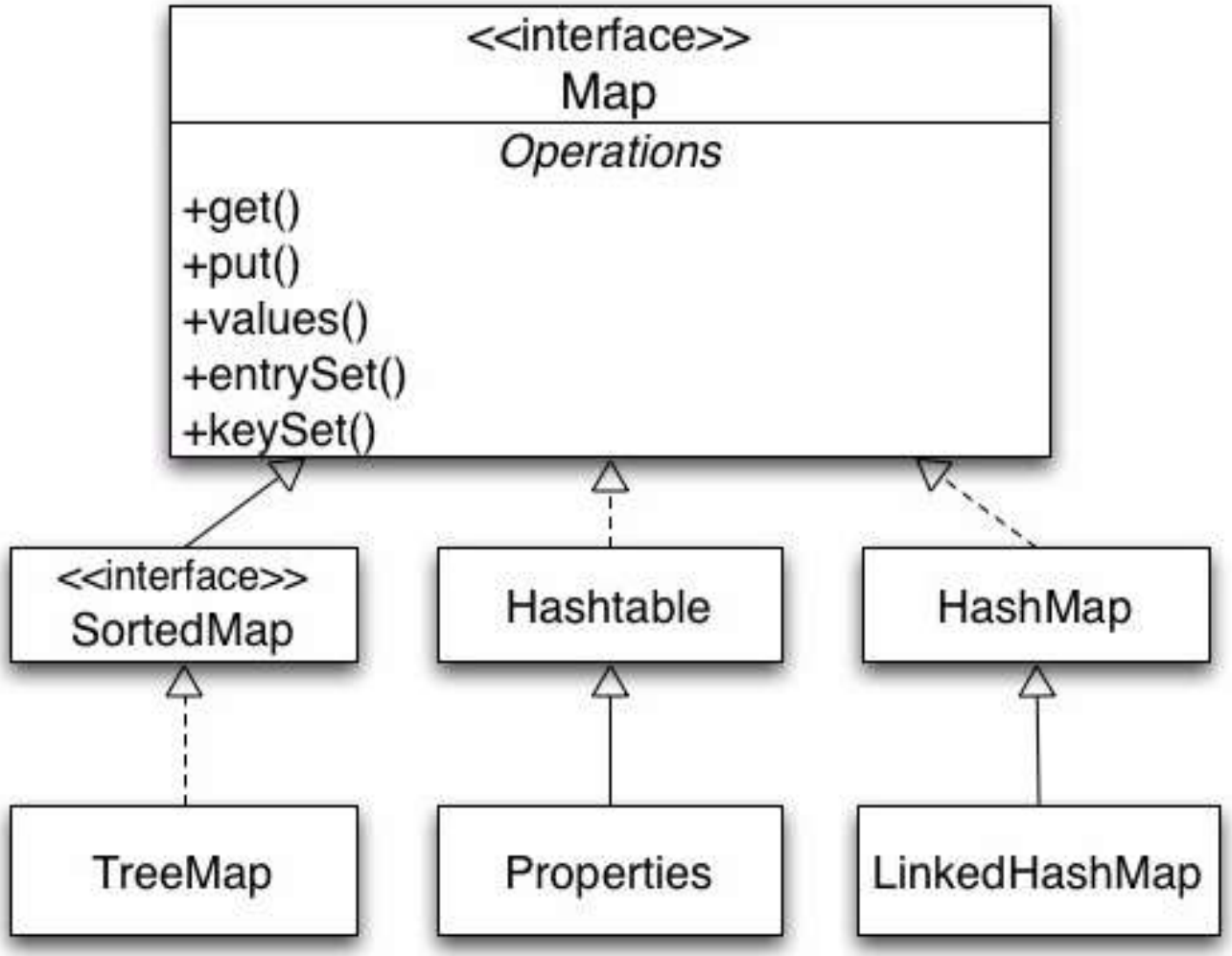
The output generated from this program is:

```
[one, second, 3rd, 4, 5.0, second, 4]
```

The `Map` Interface

- Maps are sometimes called associative arrays
- A `Map` object describes mappings from keys to values:
  - Duplicate keys are not allowed
  - One-to-many mappings from keys to values is not permitted
- The contents of the `Map` interface can be viewed and manipulated as collections
  - `entrySet` – Returns a `Set` of all the key-value pairs.
  - `keySet` – Returns a `Set` of all the keys in the map.
  - `values` – Returns a `Collection` of all values in the map.

The `Map` Interface API



A `Map` Example

```
1   import java.util.*;
2   public class MapExample {
3     public static void main(String args[])
4       { Map map = new HashMap();
5       map.put("one","1st");
6       map.put("second", new Integer(2));
7       map.put("third","3rd");
8       // Overwrites the previous assignment
9       map.put("third","III");
10      // Returns set view of keys
11      Set set1 = map.keySet();
12      // Returns Collection view of values
13      Collection collection = map.values();
14      // Returns set view of key value mappings
15      Set set2 = map.entrySet();
16      System.out.println(set1 + "\n" + collection + "\n" + set2);
17    }
18  }
```

A `Map` Example

Output generated from the `MapExample` program:

```
[second, one, third]
[2, 1st, III]
[second=2, one=1st, third=III]
```

Legacy Collection Classes

Collections in the JDK include:

- The `Vector` class, which implements the `List` interface.
- The `Stack` class, which is a subclass of the `Vector` class and supports the `push`, `pop`, and `peek` methods.
- The `Hashtable` class, which implements the `Map` interface.
- The `Properties` class is an extension of `Hashtable` that only uses `Strings` for keys and values.
- Each of these collections has an `elements` method that returns an `Enumeration` object. The `Enumeration` interface is incompatible with, the `Iterator` interface.

Generics

Generics

Generics are described as follows:

- Provide compile-time type safety
- Eliminate the need for casts
- Provide the ability to create compiler-checked homogeneous collections

Generics

Using non-generic collections:

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

Using generic collections:

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

Generic `Set` Example

```
1   import java.util.*;
2   public class GenSetExample {
3     public static void main(String[] args)
4       { Set<String> set = new HashSet<String>();
5       set.add("one");
6       set.add("second");
7       set.add("3rd");
8   // This line generates compile error
9       set.add(new Integer(4));
10      set.add("second");
11  // Duplicate, not added
12      System.out.println(set);
13    }
14  }
```

Generic Map Example

```
1   import java.util.*;
2
3   public class MapPlayerRepository
4     { HashMap<String, String>
5
6     public MapPlayerRepository() {
7       players = new HashMap<String, String> ();
8     }
9
10    public String get(String position)
11      { String player =
12      players.get(position); return player;
13    }
14
15    public void put(String position, String name) {
16      players.put(position, name);
17    }
```

Generics: Examining TypeParameters

Shows how to use type parameters

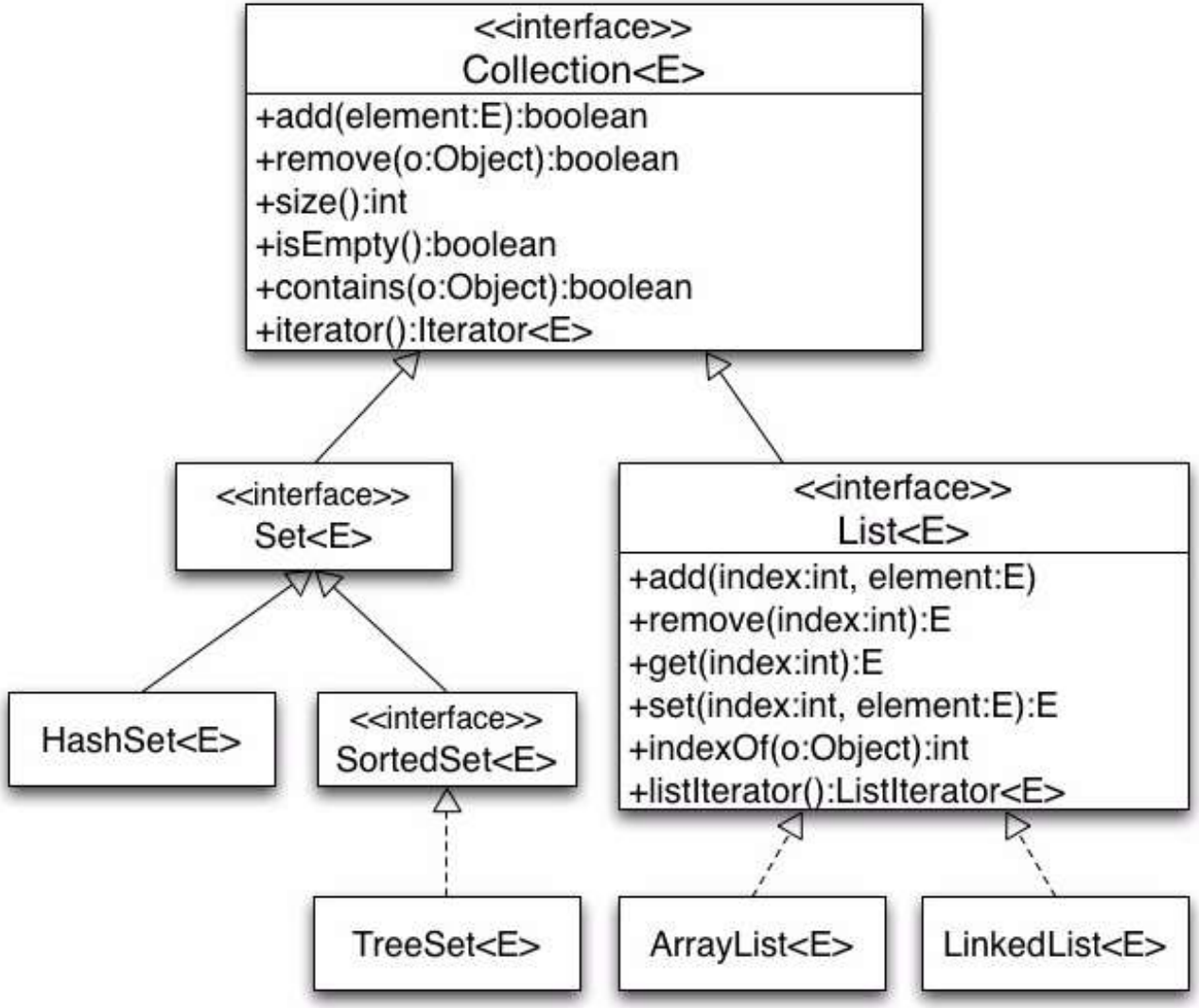
Category	Non Generic Class	Generic Class
Class declaration	`public class ArrayList extends AbstractList implements List`	`public class ArrayList<E> extends AbstractList<E> implements List <E>`
Constructor declaration	`public ArrayList (int capacity);`	`public ArrayList<E> (int capacity);`

Method declaration	`public void add((Object o)`
	
	`public Object get(int index)`
		`public void add(E o)`
		
		`public E get(int index)`

Variable declaration examples	`ArrayList list1;`
	
	`ArrayList list2;`
		`ArrayList <String> list1;`
		
		`ArrayList <Date> list2;`

Instance declaration examples	`list1 = new ArrayList(10);`
	
	`list2 = new ArrayList(10);`
		`list1= new ArrayList<String> (10);`
		
		`list2= new ArrayList<Date> (10);`

Generic Collections API



Define My Generic Class

```
class Box<T>

    { private T

    data;

    public Box(T data)
        { this.data =
        data;
    }

    public T getData()
        { return data;
```

```
public class MyGenericTest {

    public static void main(String[] args)

        { Box<String> name =
            new Box<String>("corn");
        System.out.println("name:" +
            name.getData());
    }

}
```

Refactoring Existing Non-Generic Code

```
1   import java.util.*;
2   public class GenericsWarning {
3     public static void main(String[] args)
4       List list = { new ArrayList();
5       list.add(0, new Integer(42));
6       int total = ((Integer)list.get(0).intValue();
7     }
8   }
```

javac GenericsWarning.java

```
Note: GenericsWarning.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

javac -Xlint:unchecked GenericsWarning.java

```
GenericsWarning.java:7: warning: [unchecked] unchecked call to add(int,E)
as a member of the raw type java.util.ArrayList
    list.add(0, new Integer(42));
             ^
1 warning
```

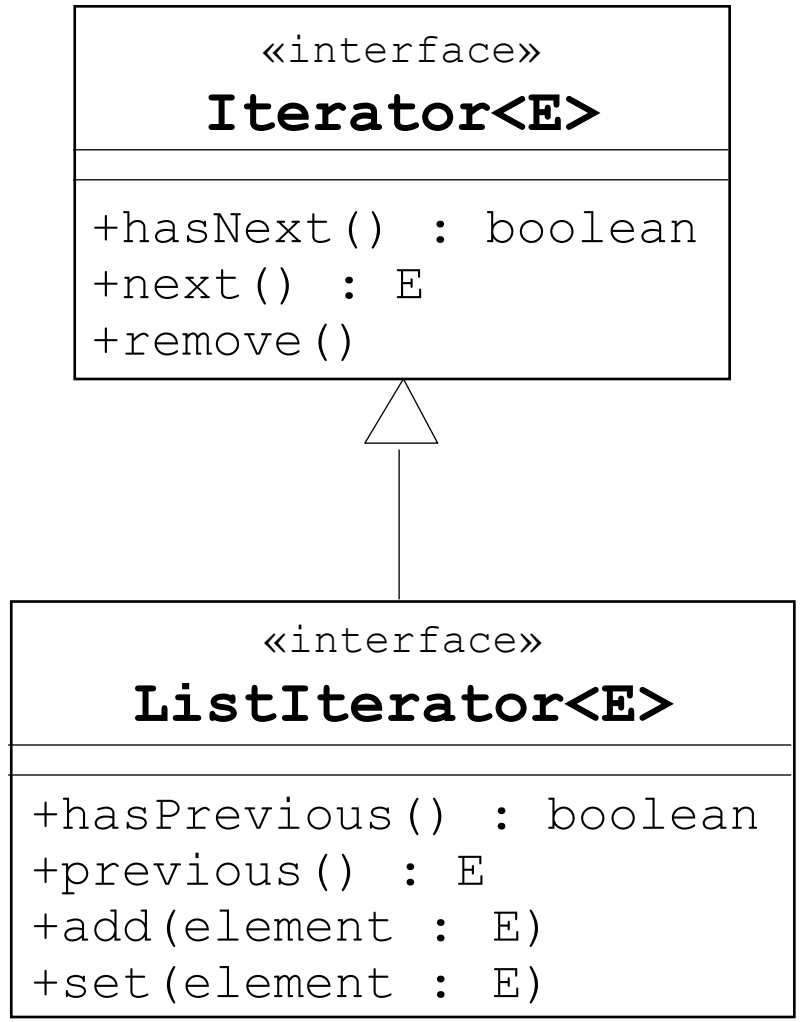
Collection Iteration : *Iterator* or Enhanced *for* Statement

Iterators

- Iteration is the process of retrieving every element in a collection.
- The basic `Iterator` interface allows you to scan forward through any collection.
- A `List` object supports the `ListIterator`, which allows you to scan the list backwards and insert or modify elements.

```
1   List<Student> list = new ArrayList<Student>();
2   // add some elements
3   Iterator<Student> elements = list.iterator();
4   while (elements.hasNext()) {
5     System.out.println(elements.next());
6   }
```

Generic Iterator Interfaces



The Enhanced `for` Loop

An enhanced `for` loop can look like the following:

- Using the iterator with a traditional `for` loop:

```
public void deleteAll(Collection<NameList> c){
  for ( Iterator<NameList> i = c.iterator() ; i.hasNext() ; ){
    NameList nl = i.next();
    nl.deleteItem();
  }
}
```

- Iterating using an enhanced `for` loop in collections:

```
public void deleteAll(Collection<NameList>
  c){ for ( NameList nl : c ){
    nl.deleteItem();
  }
}
```

The Enhanced `for` Loop

- Nested enhanced `for` loops:

```
1   List<Subject> subjects=...;
2   List<Teacher> teachers=...;
3   List<Course> courseList = ArrayList<Course)();
4   for (Subject subj: subjects) {
5     for (Teacher tchr: teachers) {
6        courseList.add(new Course(subj, tchr));
7     }
8   }
```

The Enhanced `for` Loop

The enhanced `for` loop has the following characteristics:

- Simplified iteration over collections
- Much shorter, clearer, and safer
- Effective for arrays
- Simpler when using nested loops
- Iterator disadvantages removed

Iterators are error prone:

- Iterator variables occur three times per loop.
- This provides the opportunity for code to go wrong.

System Properties

System Properties

- System properties are a feature that replaces the concept of *environment variables* (which are plat-form-specific).
- The `System.getProperties`() method returns a `Properties` object.
- The `getProperty` () method returns a `String` representing the value of the named property.
- Use the `-D` option on the command line to include a new property.

The `Properties` Class

- The `Properties` class implements a mapping of names to values (a `String`-to-`String` map).
- The `propertyNames` method returns an `Enumeration` of all property names.
- The `getProperty` method returns a `String` representing the value of the named property.
- You can also read and write a properties collection into a file using `load` and `store`.

The `Properties` Class

- Using method of Properties:
  - ShowProperties.java

- The following is an example test run of this program:
  java -DmyProp=myValue ShowProperties

- Setup new properties from file myProperties.txt:
  - PropertiesTest.java

The *Comparable* and *Comparator* interfaces

Ordering Collections

The `Comparable` and `Comparator` interfaces are useful for ordering collections:

- The `Comparable` interface imparts natural ordering to classes that implement it.
- The `Comparator` interface specifies order relation. It can also be used to override natural ordering.
- Both interfaces are useful for sorting collections.

The `Comparable` Interface

Imparts natural ordering to classes that implement it:

- Used for sorting
- The `compareTo` method should be implemented to make any class comparable:
  - `int compareTo(Object o)` method
  - `int compareTo(T o)` of Comparable<T>
- The `String`, `Date`, and `Integer` classes implement the `Comparable` interface
- You can sort the `List` elements containing objects that implement the `Comparable` interface

The `Comparable` Interface

- While sorting, the `List` elements follow the natural ordering of the element types
  - `String` elements – Alphabetical order
  - `Date` elements – Chronological order
  - `Integer` elements – Numerical order

- E.g. Comparable/
  - ComparableTest.java
  - Student.java

The `Comparator` Interface

- Represents an order relation
- Used for sorting
- Enables sorting in an order different from the natural order
- Used for objects that do not implement the `Comparable` interface
- Can be passed to a sort method

You need the `compare` method to implement the `Comparator` interface:

- `int compare(Object o1, Object o2)` method
- int compare(T o1, T o2) method of *Comparator*<*T*>

Example of the `Comparator` Interface

- Comparator/*.java
  - Student.java
  - GradeComp.java
  - NameComp.java
  - TestComparator.java

Polymorphic Algorithms

Polymorphic Algorithms

- come from the Collections class, and all take the form of static methods whose first argument is the collection on which the operation is to be performed
  - Sorting
  - Searching
  - Shuffling
    - public static void **shuffle**(List<?> list)
  - **Routine Data Manipulation**
    - reverse, fill , copy , swap , addAll , etc.

Collections.sort()

```
import java.util.*;
public class Sort {
    public static void main(String[] args)
        { List<String> list =
        Arrays.asList(args);
        Collections.sort(list);
        System.out.println(list);
    }
```

Collections.sort()

- public static <*T extends Comparable<? super T>>* void **sort**(List<*T*> **list**)
- public static <*T*> void sort(List<*T*> **list**, Comparator<? super *T*> **c**)

- **TestCollectionsSort.java**

Collections.binarySearch()

- publicstatic <*T*> intbinarySearch (List<?extends Comparable<?super*T*>> **list**,*T* **key**)
- publicstatic<*T*>intbinarySearch(List<?extends*T*>**list**, T**key**, Comparator<?superT> c)

- **TestCollectionsSearch.java**

jar And *javadoc* Tools

Documentation Tags

- Comments starting with `/**` are parsed by the `javadoc` tool
- These comments should immediately precede the declaration they reflect
- DocExample.java

Tag	Purpose	Class/ Interface	Constructor	Method	Attribute
`@see`	To create a link to another declaration (or any other HTML page)	✓	✓	✓	✓
`@deprecated`	Documents that the declaration has been deprecated in this re-lease	✓	✓	✓	✓
`@author`	The author of the class or interface	✓			
`@param`	Documents a parameter		✓	✓	
`@throws` `@exception`	Documents why an exception might be thrown		✓	✓	
`@return`	DocumentYy return Value/ Type			✓	

Using the `javadoc` Tool

- This Java 2 SDK tool generates HTML documentation pages
- Usage: `javadoc [options] [packages|files]`

  This example generates the API documentation for the complete Banking project:

```
javadoc -d ../doc/api banking banking.domain /
   banking.reports
```

Option	Value	Description
`-d`	output path	The directory in which the generated HTML files should be placed.
`-sourcepath`	directory path	The root directory where the source file package tree.
`-public`		Specifies that only public declarations be included (*default*).
`-private`		Specifies that all declarations be included.

Public Documentation

> **`javadoc -d doc/api/public DocExample.java`**

程序包 类 树 已过时 索引 帮助

概要: 嵌套 | 字段 | 构造器 | 方法 详细资料: 字段 | 构造器 | 方法 SEARCH: Search

程序包 mypack

类 DocExample

java.lang.Object
mypack.DocExample

```
public class DocExample
extends java.lang.Object
```

文档注释 This class contains a bunch of documentation tags.

字段概要

字段

修饰符和类型	字段	说明
java.lang.String	test	A public attribute tag.

构造器概要

截图(Alt + A)

构造器

构造器	说明
DocExample()	This is the default constructor
DocExample(int x_value)	This constructor initializes the x attribute.

Private Documentation

- **`javadoc -private -d doc/api/private DocExample.java`**

程序包 类 树 已过时 索引 帮助

概要: 嵌套 | 字段 | 构造器 | 方法 详细资料: 字段 | 构造器 | 方法 SEARCH: Search

java.lang.Object
mypack.DocExample
mypack.ChenTest

class **ChenTest**
extends DocExample

This class is a test for another class definition.

版本:
0.1(beta)

作者:
CHEN Xudong

字段概要

字段

修饰符和类型	字段	说明
private java.lang.String	firstName	
private java.lang.String	lastName	
private java.lang.String	name	*** @deprecated : replaced by firstName and lastName

Using the `jar` Tool

- This JDK tool generates a compressed archive of .class and media files
  Usage: *jar* [options] [archive_file] [files]
- This generates an archive for the Banking project:

  jar cvf bank.jar bank/domain/*.class bank/reports/*.class

  This extracts an archive for the Banking project:

  jar xvf bank.jar

Option	Value	Description
`c`		This option creates a new archive.
`f`	filepath	specifies the filepath of the Java archive (JAR) file.
`x`		This option extracts an archive to the current directory.
`v`		This option specifies verbose output from the `jar` tool.

Summary

- The **Collections** Framework
- Collections API Interfaces - **collection**, **set**, and **map**
- Generics set, and Generics map
- Retrieving a Collection - Iterator, Enhanced for
- System Properties
- The **Comparable** and **Comparator** Interfaces
- Polymorphic Algorithms
- ***jar*** And ***javadoc*** Tools