Programming in Java

Collections and Generics Framework

Hua Huang, Ph.D.

Spring 2019

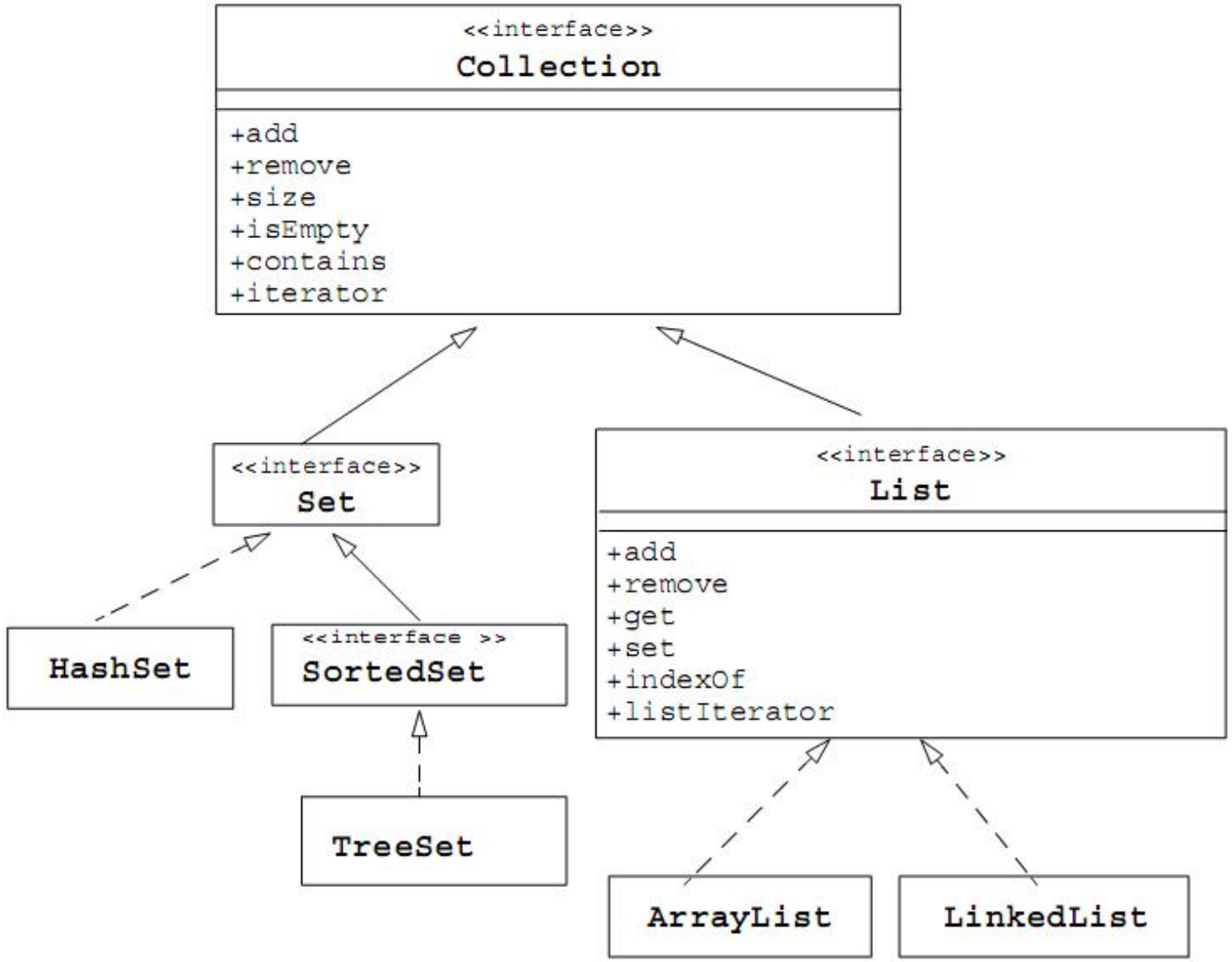
Objectives

- Describe the **Collections**
- Describe the general purpose implementations of the **core interfaces** in the **Collections** framework
- Examine the **Map** interface
- Examine the **legacy collection classes**
- Create **natural** and **custom** ordering by implementing the `Comparable` and `Comparator` interfaces
- Use **`generic collections`**
- Use **type parameters** in generic classes
- **Refactor** existing non-generic code
- Write a program to **iterate over a collection**
- Examine the enhanced for loop again

The Collections API

- 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

```
01 import java.util.*;
02 public class SetExample {
03   public static void main(String[] args) {
04     Set set = new HashSet();
05     set.add("one");
06     set.add("second");
07     set.add("3rd");
08     set.add(new Integer(4));
09     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

```
01 import java.util.*
02 public class ListExample {
03   public static void main(String[] args) {
04     List list = new ArrayList();
05     list.add("one");
06     list.add("second");
07     list.add("3rd");
08     list.add(new Integer(4));
09     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:

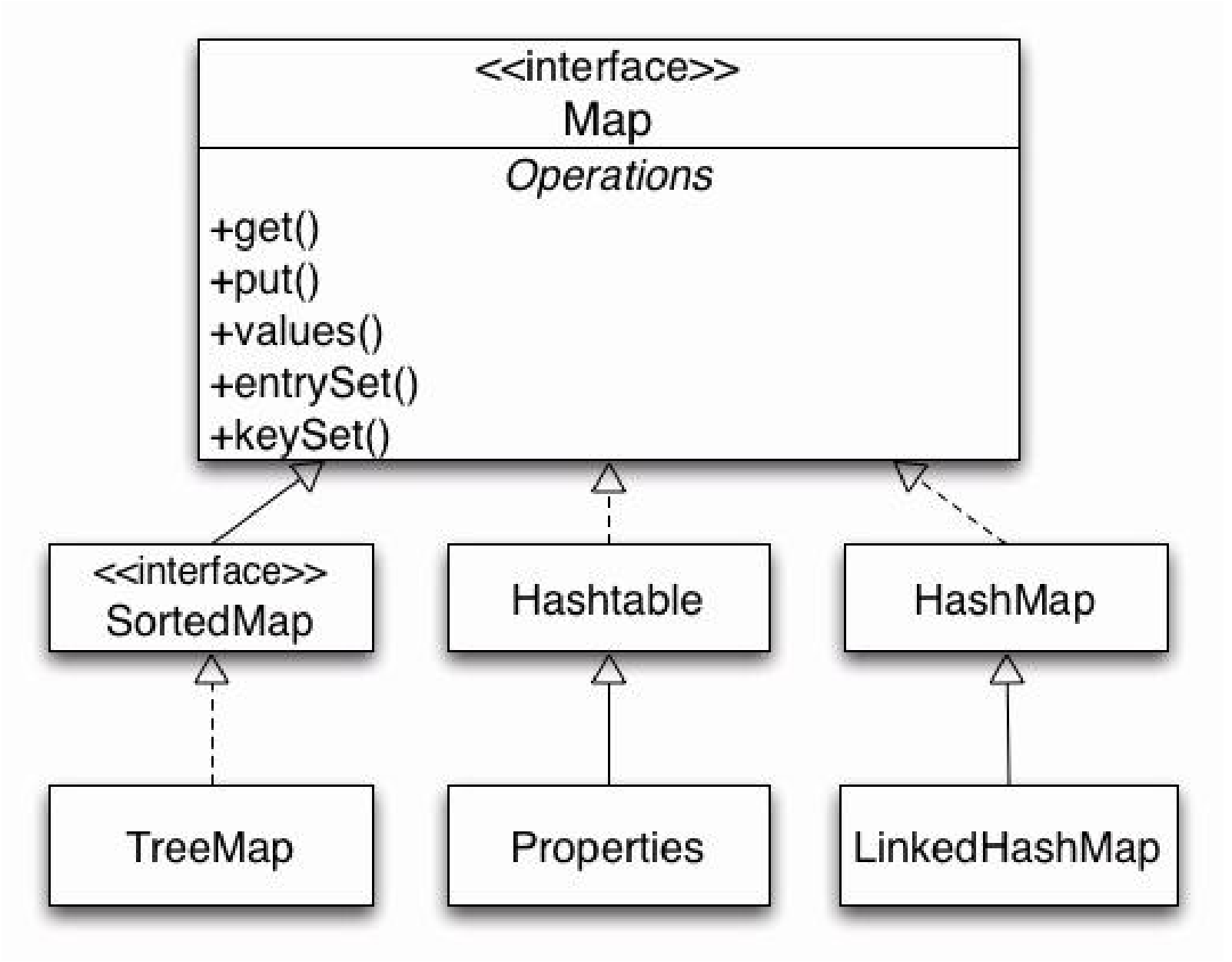
```
[one, second, 3rd, 4, 5.0, second, 4]
```

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

```
01 import java.util.*;
02 public class MapExample {
03   public static void main(String args[]) {
04     Map map = new HashMap();
05     map.put("one","1st");
06     map.put("second", new Integer(2));
07     map.put("third","3rd");
08     // Overwrites the previous assignment
09     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(Cont.)

- 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.

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(T o) method`
  - 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(Cont.)

- 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

Example of the `Comparable` Interface

```
01  import java.util.*;
02  class Student implements Comparable {
03    String firstName, lastName;
04    int studentID=0;
05    double GPA=0.0;
06    public Student(String firstName, String lastName,
07                   int studentID, double GPA) {
08      if (firstName == null || lastName == null || studentID == 0
09          || GPA == 0.0) {
10        throw new IllegalArgumentException();
11      }
12      this.firstName = firstName;
13      this.lastName = lastName;
14      this.studentID = studentID;
15      this.GPA = GPA;
16    }
```

Example of the `Comparable` Interface(Cont.)

```
17    public String firstName() { return firstName; }
18    public String lastName()      { return lastName; }
19    public int studentID() { return studentID; }
20    public double GPA() { return GPA; }
21    // Implement compareTo method.
22    public int compareTo(Object o) {
23      double f = GPA-((Student)o).GPA;
24      if (f == 0.0)                          ????
25        return 0;      // 0 signifies equals
26      else if (f<0.0)
27        return -1;     // negative value signifies less than or before
28      else
29        return 1;      // positive value signifies more than or after
30    }
31  }
```

Example of the Comparable Interface(Cont.)

```
01  import java.util.*;
02  public class ComparableTest {
03    public static void main(String[] args) {
04      TreeSet studentSet = new TreeSet();
05      studentSet.add(new Student("Mike", "Hauffmann",101,4.0));
06      studentSet.add(new Student("John", "Lynn",102,2.8 ));
07      studentSet.add(new Student("Jim", "Max",103, 3.6));
08      studentSet.add(new Student("Kelly", "Grant",104,2.3));
09      Object[] studentArray = studentSet.toArray();
10      Student s;
11      for(Object obj : studentArray) {
12        s = (Student) obj;
13        System.out.printf("Name = %s %s ID = %d GPA = %.1f\n",
14          s.firstName(), s.lastName(), s.studentID(), s.GPA());
15      }
16    }
17  }
```

Example of the `Comparable` Interface(Cont.)

- Generated Output:

```
Name = Kelly Grant ID = 104 GPA = 2.3
Name = John Lynn ID = 102 GPA = 2.8
Name = Jim Max ID = 103 GPA = 3.6
Name = Mike Hauffmann ID = 101 GPA = 4.0
```

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

Example of the `Comparator` Interface

```
01  class Student1 {
02    private String firstName;
03    private String lastName;
04    private int studentID=0;
05    private double GPA=0.0;
06    public Student1(String firstName, String lastName,
07                    int studentID, double GPA) {
08      if (firstName == null || lastName == null || studentID==0||
09          GPA == 0.0) throw new NullPointerException();
10      this.firstName = firstName;
11      this.lastName = lastName;
12      this.studentID = studentID;
13      this.GPA = GPA;
14    }
15    public String firstName() { return firstName; }
16    public String lastName() { return lastName; }
17    public int studentID() { return studentID; }
18    public double GPA() { return GPA; }
19  }
```

Example of the `Comparator` Interface(Cont.)

```
1  import java.util.*;
2  public class NameComp implements Comparator<Student1> {
3    public int compare(Student1 s1, Student1 s2) {
4      return s1.firstName().compareTo(s2.firstName());
5    }
6  }
```

```
01  import java.util.*;
02  public class GradeComp implements Comparator<Student1> {
03    public int compare(Student1 s1, Student1 s2) {
04      if (s1.GPA() == s2.GPA())
05        return 0;
06      else if (s1.GPA() < s2.GPA())
07        return -1;
08      else
09        return 1;
10    }
11  }
```

Example of the Comparator Interface(Cont.)

```
01  import java.util.*;
02  public class ComparatorTest {
03    public static void main(String[] args) {
04      Comparator<Student1> c = new NameComp();
05      TreeSet<Student1> studentSet = new TreeSet<>(c);
06      studentSet.add(new Student1("Mike", "Hauffmann",101,4.0));
07      studentSet.add(new Student1("John", "Lynn",102,2.8 ));
08      studentSet.add(new Student1("Jim", "Max",103, 3.6));
09      studentSet.add(new Student1("Kelly", "Grant",104,2.3));
10      Student1[] studentArray = studentSet.toArray(new
Student1[studentSet.size()]);
11
12      for(Student1 s : studentArray) {
13        System.out.printf("Name = %s %s ID = %d GPA = %.1f\n",
14          s.firstName(), s.lastName(), s.studentID(), s.GPA());
15      }
16    }
17  }
```

Example of the `Comparator` Interface(Cont.)

```
Name = Jim Max ID = 0 GPA = 3.6
Name = John Lynn ID = 0 GPA = 2.8
Name = Kelly Grant ID = 0 GPA = 2.3
Name = Mike Hauffmann ID = 0 GPA = 4.0
```

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(Cont.)

- 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

```
01  import java.util.*;
02  public class GenSetExample {
03    public static void main(String[] args) {
04      Set<String> set = new HashSet<String>();
05      set.add("one");
06      set.add("second");
07      set.add("3rd");
08      // This line generates compile error
09      set.add(new Integer(4));
10      set.add("second");
11      // Duplicate, not added
12      System.out.println(set);
13    }
14  }
```

Generic Map Example

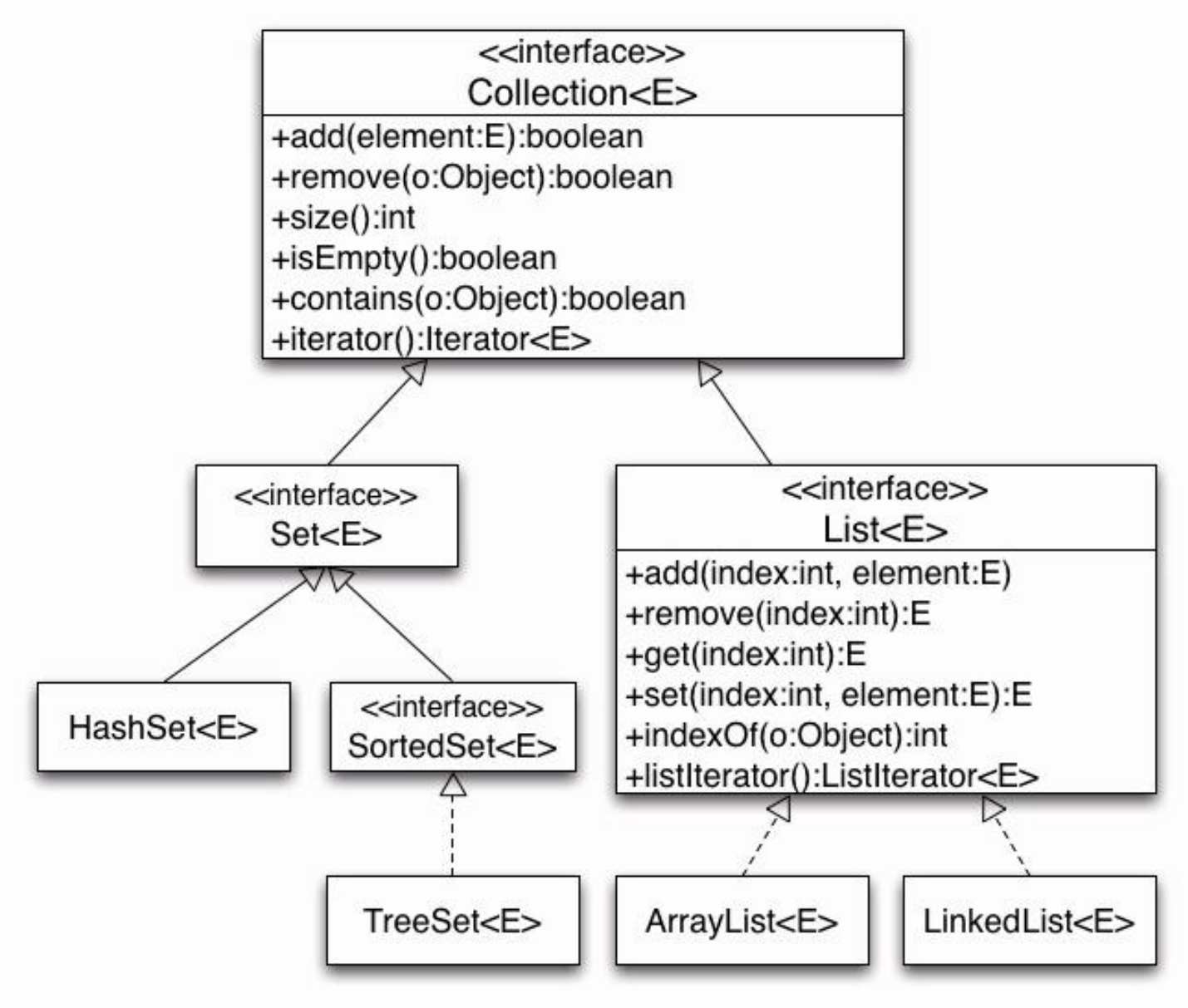
```
01  import java.util.*;
02  public class MapPlayerRepository {
03    Map<String, String> players;
04    public MapPlayerRepository() {
05      players = new HashMap<String, String> ();//HashMap<>
06    }
07    public String get(String position) {
08      String player = players.get(position);
09      return player;
10    }
11    public void put(String position, String name) {
12      players.put(position, name);
13    }
14  }
```

Generics: Examining Type Parameters

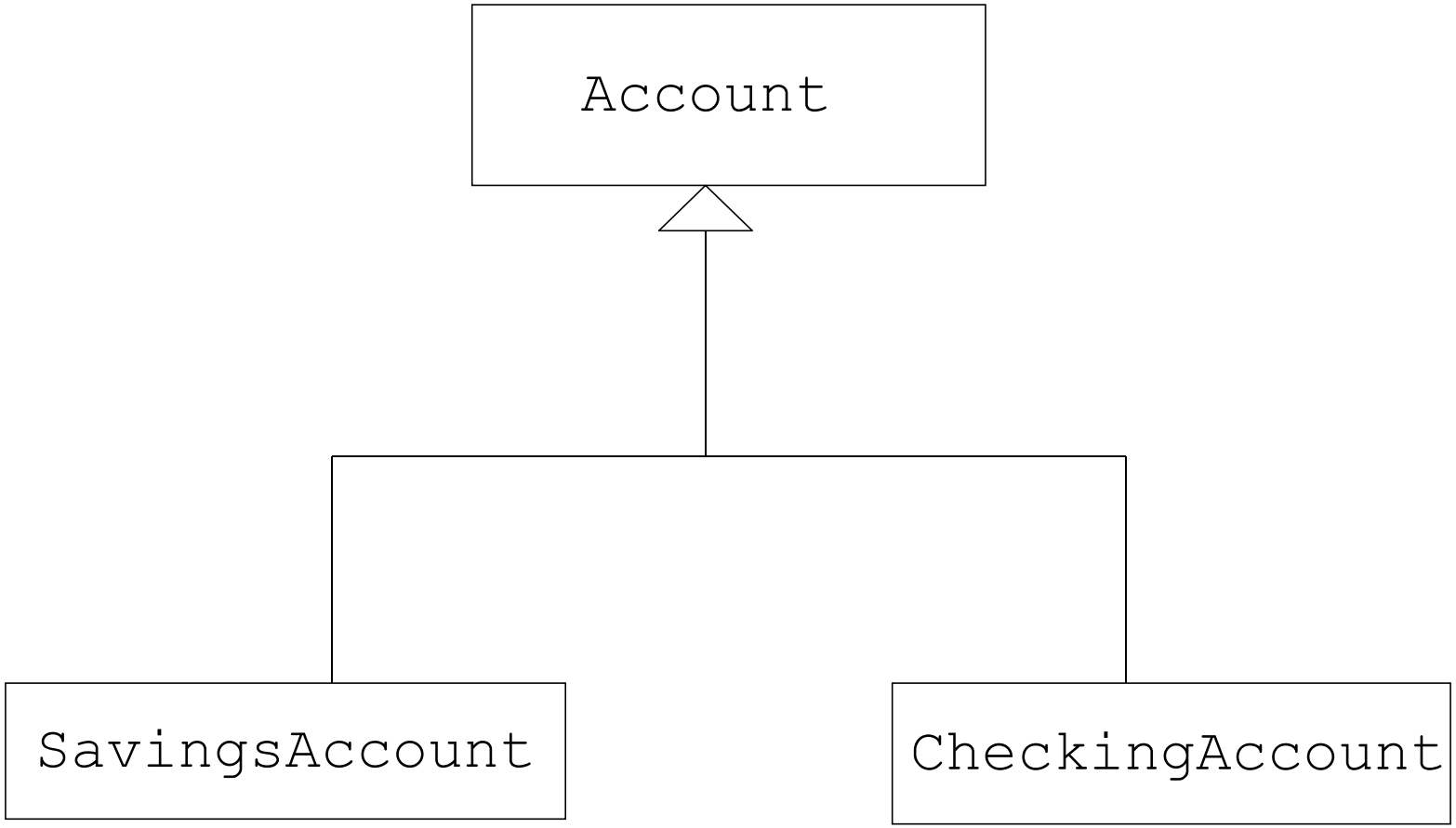
- Shows how to use type parameters

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



Wild Card Type Parameters



The Type-Safety Guarantee

```
01  public class TestTypeSafety {
02    public static void main(String[] args) {
03      List<CheckingAccount> lc =
04        new ArrayList<CheckingAccount>();
05
06      lc.add(new CheckingAccount("Fred")); // OK
07      lc.add(new SavingsAccount("Fred"));
08      // Compile error!
09
10      // therefore...
11      CheckingAccount ca = lc.get(0);
12      // Safe, no cast required
13    }
14  }
```

The Invariance Challenge

```
01  List<Account> la;
02  List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
03  List<SavingsAccount> ls = new ArrayList<SavingsAccount>();
04  //if the following were possible...
05  la = lc;
06  la.add(new CheckingAccount("Fred"));
07  //then the following must also be possible...
08  la = ls;
09  la.add(new CheckingAccount("Fred"));
10  //so...
11  SavingsAccount sa = ls.get(0); //aarrgghh!!
```

- In fact, `la=lc;` is illegal, so even though a `CheckingAccount` is an `Account`, an `ArrayList<CheckingAccount>` is not an `ArrayList<Account>`.

The Covariance Response

```
01  public static void printNames(List <? extends Account> lea) {
02    for (int i=0; i < lea.size(); i++) {
03      System.out.println(lea.get(i).getName());
04    }
05  }
06
07  public static void main(String[] args) {
08    List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
09    List<SavingsAccount> ls = new ArrayList<SavingsAccount>();
10
11    printNames(lc);
12    printNames(ls);
13
14    //but...
15    List<? extends Object> leo = lc; //OK
16    leo.add(new CheckingAccount("Fred"));//Compile error!
17  }
```

Generics: 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 or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: GenericsWarning.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

javac -Xlint:unchecked GenericsWarning.java

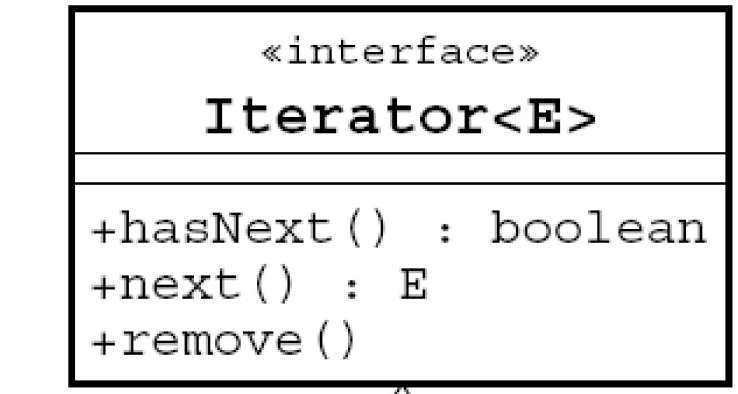
```
GenericsWarning.java:5: warning: [unchecked] unchecked call to add(int,E) as a member of the raw type List
   list.add(0, new Integer(42));
        ^
  where E is a type-variable:
    E extends Object declared in interface List
Note: GenericsWarning.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
1 warning
```

Iterators

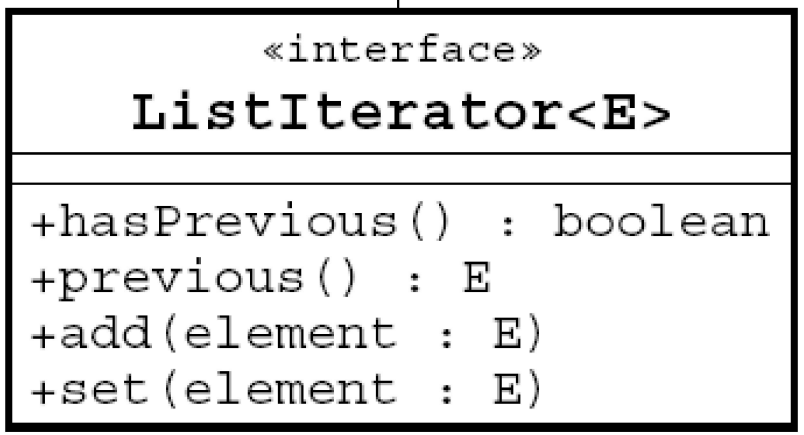
- Iteration is the process of **retrieving every element** in a collection.
- The basic `Iterator` interface allows you to **scan forward** through any collection.
- A `List` object supports the `ListIterator`, which allows you to **scan the list backwards** and **insert** or **modify** elements.

```
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



UnsupportedOperationException



UnsupportedOperationException

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.

The Enhanced `for` Loop(Cont.)

- 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(Cont.)

- **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  }
```

Summary

- **Core interfaces** in the Collections framework
- Legacy collection classes
- Creating natural and custom ordering by implementing the `Comparable` and `Comparator` interfaces
- Using **generic collections**
- Using **type parameters** in generic classes
- **Iteration** over a collection and the **enhanced for loops**

Questions or Comments?