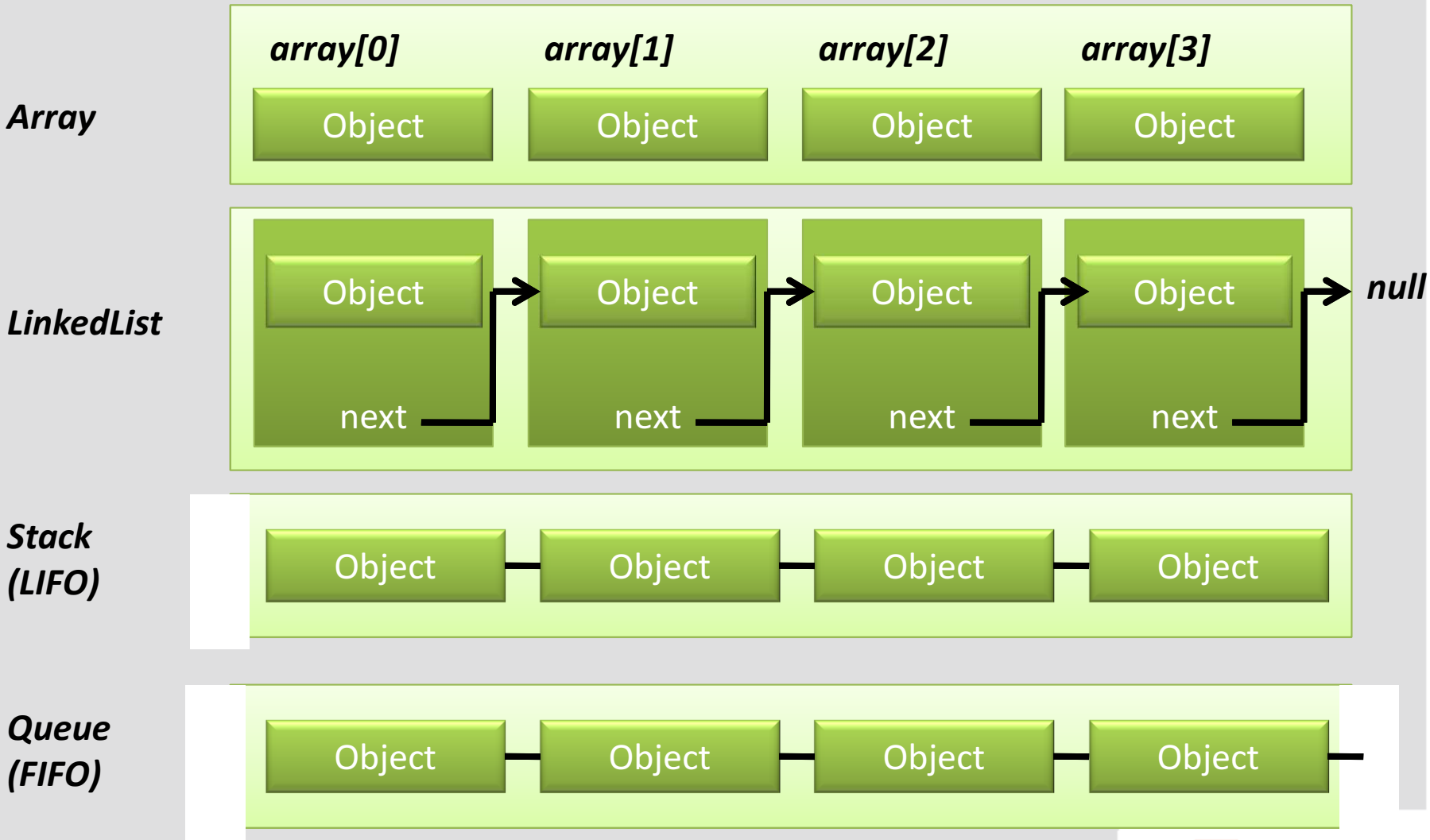
Collections and Generics Framework

Mārtiņš Leitass

lattelecom

Collections



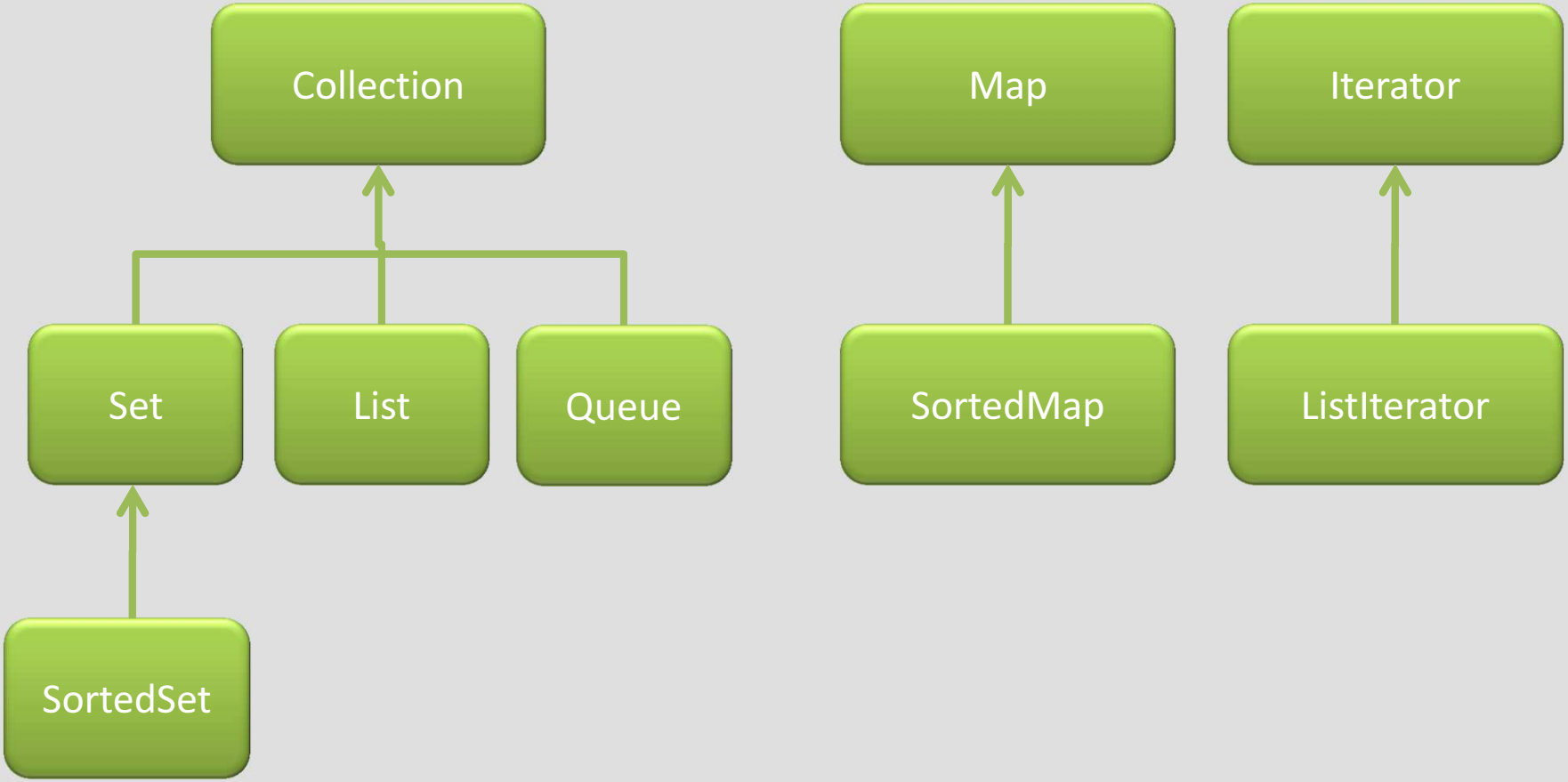
Collections in Java

- Collection classes in Java are containers of Objects which by polymorphism can hold any class that derives from Object (which is actually, any class)
- Using Generics the Collection classes can be aware of the types they store

Collections API

- a *collection* is a *container* or *object* that groups multiple objects into a single unit
- a *Collections Framework* provides a unified system for organizing and handling collections and is based on four elements:
  - Interfaces that characterize common collection types
  - Abstract Classes which can be used as a starting point for custom collections and which are extended by the JDK implementation classes
  - Classes which provide implementations of the Interfaces
  - Algorithms that provide behaviors commonly required when using collections ie search, sort, iterate, etc.
- the *Collection Framework* also provides an interface for traversing collections: Interator and it's sub interface ListIterator
- the Iterator interface should be used in preference to the earlier Enumeration interface

Java Collection Framework: core collection Interfaces



The Collections Interface

- this is the root interface for the collection hierarchy
- it is not directly implemented by an SDK class; instead they implement the subinterfaces List, Set or Queue
- it is typically used to manipulate and pass collections around in a generic manner
- classes which implement Collection or one of it's sub interfaces must provide two constructors
  - a default, no-argument constructor, which creates an empty collection
  - a constructor which takes a Collection as an argument and creates a new collection with the same elements as the specified collection

Collection: Basic operations

- int size( );
- boolean isEmpty( );
- boolean contains(Object element);
- boolean add(Object element);
- boolean remove(Object element);
- Iterator iterator( );

Collection: Bulk operations

- boolean containsAll(Collection c);
- boolean addAll(Collection c);
- boolean removeAll(Collection c);
- boolean retainAll(Collection c);
- void clear( );

- addAll, removeAll, retainAll return true if the object receiving the message was modified

Collection: Array operations

- Object[ ] toArray( );
  - creates a new array of Objects
- Object[ ] toArray(Object a[ ]);
  - Allows the caller to provide the array
- Examples:

```
Object[ ] a = c.toArray( );

String[ ] a;
a = (String[ ]) c.toArray(new String[0]);
```

General-purpose Implementations					
Interfaces	**Implementations**				
	Hash table	**Resizable array**	**Tree (sorted)**	**Linked list**	**Hash table + Linked list**
Set	HashSet		TreeSet (sorted)		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap (sorted)		LinkedHashMap

Set

- A set is a Java collections framework element which:
  - Contains items, having **no duplicates**, stored in any order
  - Has a general implementation of a HashSet (A well-known data structure for finding objects)
  - Requires that java.util package be imported before using
  - Extends the Collections interface
- implemented methods
  - add(Object obj)
    - Adds the argument to the set, if obj is not already in the set
  - size()
    - returns the number of items in the set
  - remove(Object obj)
    - removes the argument from the set, if obj is found in the set

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]

List

- a List is a collection whose elements can be accessed by an index
- the indices are zero-based
- a list has methods for inserting and removing elements
- a list can contain **duplicate** elements
- a List provides a special ListIterator which allows you to move backwards and forwards through the elements
- there are three basic ways in which a List can be modified:
  - add an element
  - remove an element
  - replace an element

List: Methods

- get(int index)
  - returns the element at the specified position
- set(int index, Object element)
  - **replaces** the element at the specified position with the given object
- add(int index, Object element)
  - **inserts** the specified element at the specified position, shifting all the elements and adds one to their index values
- remove(int index)
  - removes the element at the specified position, shifiting all the elements and subtracting one from their indices
- indexOf(Object o)
  - returns the index of the first occurence of the specified element or -1 if it is not found
- lastIndexOf(Object o)
  - returns the index of the last occurence of the specified element or -1 if it is not found
- subList(int fromIndex, int toIndex)
  - returns the portion of the list between the specified indices exclusive of the toIndex element

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  }
```

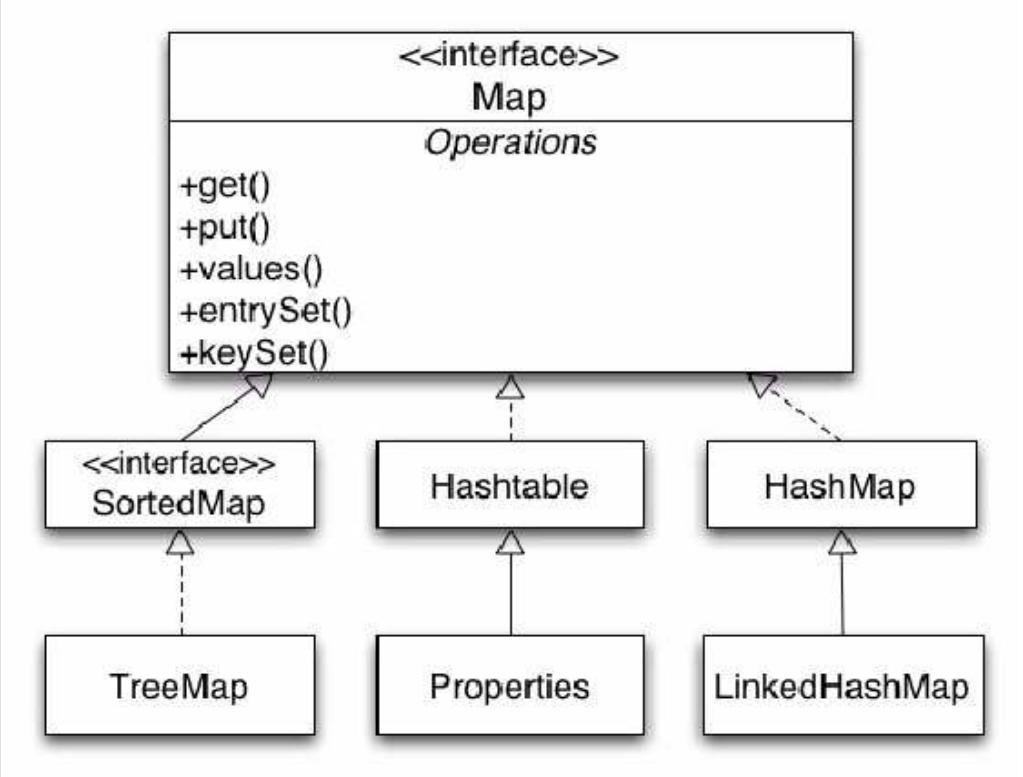
The output generated from this program is:

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

Map

- Contains keys that map to one value, in any order, but whose values can map to multiple keys
  - implemented in HashMap
- Does not extend Collection, but can be viewed as collection
- Methods:
  - put(Object key, Object val)
  - putAll(Map m)
  - get(Object key)
  - size();
  - remove(Object key)
  - clear()
  - keySet() and values()
- Subinterface SortedMap contains keys that map to one value in particular order, but whose values can map to multiple keys
  - implemented in TreeMap

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]
```

Map: suggestions

- Map implementation:
  - HashMap – best for inserting, deleting and locating elements in a Map
  - TreeMap – best if you need to traverse the keys in a sorted order
- HashMap requires that the class of key added have a well-defined hashCode() implementation

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(Object 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

- 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

```
1   import java.util.*;
2   class Student implements Comparable {
3      String firstName, lastName;
4      int studentID=0;
5      double GPA=0.0;
6      public Student(String firstName, String lastName, int studentID,
7         double GPA) {
8         if (firstName == null || lastName == null || studentID == 0
9         || GPA == 0.0) {throw new IllegalArgumentException();}
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; }
```

Example of the Comparable Interface

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

Example of the Comparable Interface

```
1   import java.util.*;
2   public class ComparableTest {
3      public static void main(String[] args) {
4        TreeSet studentSet = new TreeSet();
5        studentSet.add(new Student("Mike", "Hauffmann",101,4.0));
6        studentSet.add(new Student("John", "Lynn",102,2.8 ));
7        studentSet.add(new Student("Jim", "Max",103, 3.6));
8        studentSet.add(new Student("Kelly", "Grant",104,2.3));
9          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

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

```
1   class Student {
2      String firstName, lastName;
3      int studentID=0;
4      double GPA=0.0;
5      public Student(String firstName, String lastName,
6         int studentID, double GPA) {
7         if (firstName == null || lastName == null || studentID == 0 ||
8            GPA == 0.0) throw new NullPointerException();
9            this.firstName = firstName;
10           this.lastName = lastName;
11           this.studentID = studentID;
12           this.GPA = GPA;
13     }
14     public String firstName() { return firstName; }
15     public String lastName()   { return lastName;   }
16     public int studentID() { return studentID; }
17     public double GPA() { return GPA; }
18  }
```

Example of the Comparator Interface

```
1    import java.util.*;
2    public class NameComp implements Comparator {
3       public int compare(Object o1, Object o2) {
4          return
5             (((Student)o1).firstName.compareTo(((Student)o2).firstName));
6       }
7    }
```

```
1    import java.util.*;
2    public class GradeComp implements Comparator {
3       public int compare(Object o1, Object o2) {
4         if (((Student)o1).GPA == ((Student)o2).GPA)
5           return 0;
6         else if (((Student)o1).GPA < ((Student)o2).GPA)
7           return -1;
8         else
9           return 1;
10      }
11   }
```

Example of the Comparator Interface

```
1     import java.util.*;
2     public class ComparatorTest {
3        public static void main(String[] args) {
4           Comparator c = new NameComp();
5           TreeSet studentSet = new TreeSet(c);
6           studentSet.add(new Student("Mike", "Hauffmann",101,4.0));
7           studentSet.add(new Student("John", "Lynn",102,2.8 ));
8           studentSet.add(new Student("Jim", "Max",103, 3.6));
9           studentSet.add(new Student("Kelly", "Grant",104,2.3));
10          Object[] studentArray = studentSet.toArray();
11          Student s;
12          for(Object obj : studentArray) {
13             s = (Student) obj;
14             System.out.println("Name = %s %s ID = %d GPA = %.1f\n",
15                s.firstName(), s.lastName(), s.studentID(), s.GPA());
16          }
17    java.util.Collections
18       }
      }
```

Example of the Comparator Interface

```
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 are a way to define which types are allowed in your class or function

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 SetExample

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

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 void add(E o)
	public Object get(int index)	public E get(int index)
Variable declaration examples	ArrayList list1;	ArrayList <String> list1;
	ArrayList list2;	ArrayList <Date> list2;
Instance declaration examples	list1 = new ArrayList(10);	list1= new ArrayList<String> (10);
	list2 = new ArrayList(10);	list2= new ArrayList<Date> (10);

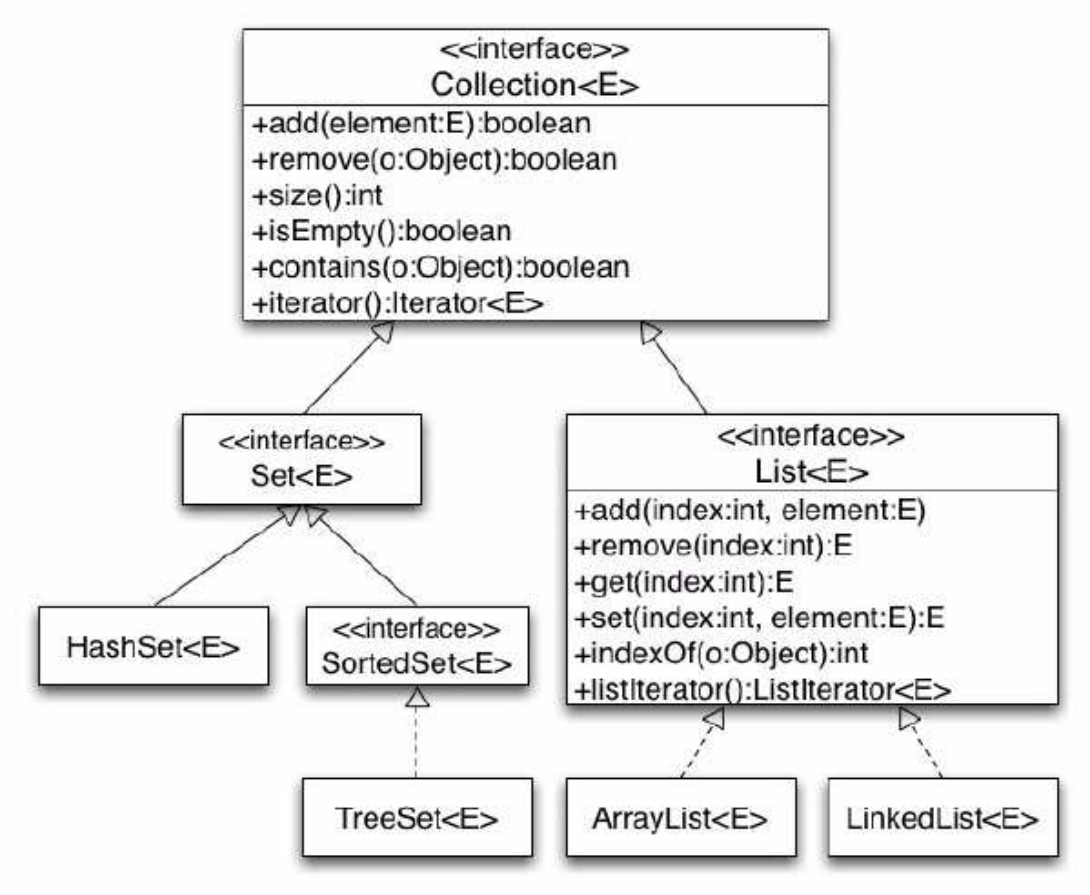
Defining (our own) Generic Types

```
public class GenericClass<T> {
    private T obj;
    public void setObj(T t) {obj = t;}
    public T getObj() {return obj;}
    public void print() {
        System.out.println(obj);
    }
}
```

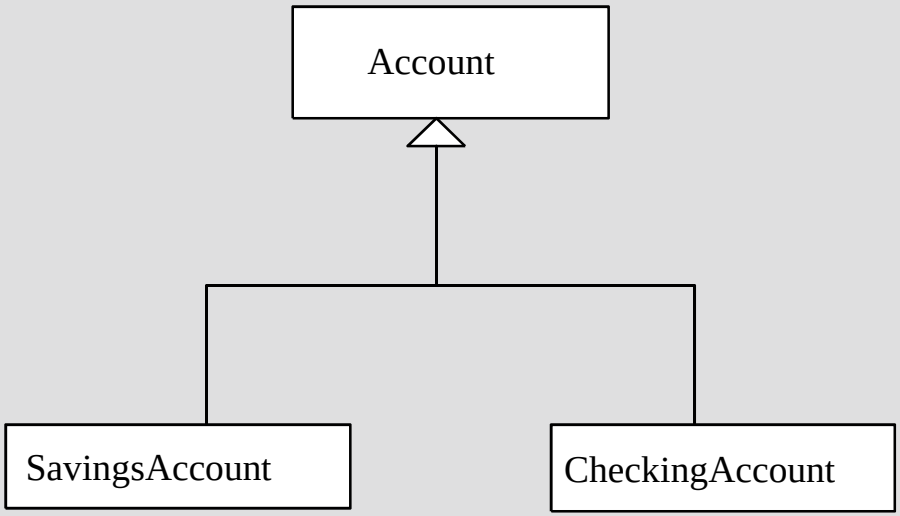
Main:

```
    GenericClass<Integer> g = new GenericClass<Integer>();
    g.setObj(5); // auto-boxing
    int i = g.getObj(); // auto-unboxing
    g.print();
```

Generic Collections API



Wild Card Type Parameters



The Type-Safety Guarantee

```
1   public class TestTypeSafety {
2
3       public static void main(String[] args) {
4           List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
5
6           lc.add(new CheckingAccount("Fred")); // OK
7           lc.add(new SavingsAccount("Fred"));    // Compile error!
8
9           // therefore...
10          CheckingAccount ca = lc.get(0);                    // Safe, no cast required
11      }
12  }
```

The Invariance Challenge

```
7       List<Account> la;
8       List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
9       List<SavingsAccount> ls = new ArrayList<SavingsAccount>();
10
11      //if the following were possible...
12      la = lc;
13      la.add(new CheckingAccount("Fred"));
14
15      //then the following must also be possible...
16      la = ls;
17      la.add(new CheckingAccount("Fred"));
18
19      //so...
20      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

```
6     public static void printNames(List <? extends Account> lea) {
7           for (int i=0; i < lea.size(); i++) {
8              System.out.println(lea.get(i).getName());
9           }
10    }
11
12    public static void main(String[] args) {
13       List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
14       List<SavingsAccount> ls = new ArrayList<SavingsAccount>();
15
16       printNames(lc);
17       printNames(ls);
18
19       //but...
20       List<? extends Object> leo = lc; //OK
21       leo.add(new CheckingAccount("Fred"));//Compile error!
22    }
23 }
```

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 unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

javac -Xlint:unchecked GenericsWarning.java

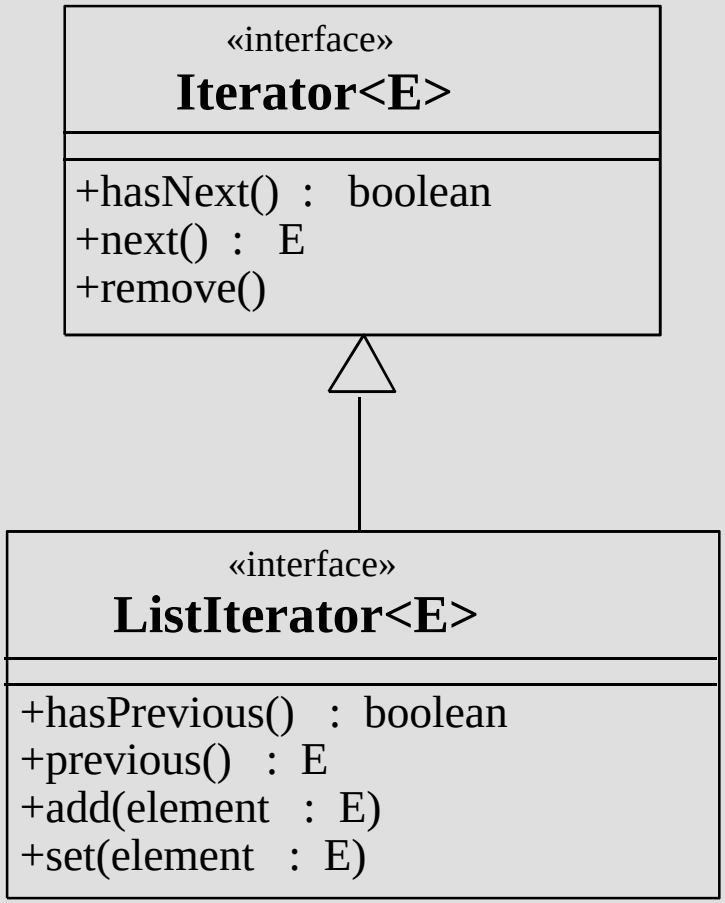
```
GenericsWarning.java:7: warning: [unchecked] unchecked call to add(int,E)
as a member of the raw type java.util.ArrayList
      list.add(0, new Integer(42));
                  ^
1 warning
```

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 Listobject 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 forLoop

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 forLoop

An enhanced forloop can look like the following:

- Using the iterator with a traditional forloop:

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

- Iterating using an enhanced forloop in collections:

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

The Enhanced forLoop

- 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   }
```

Collection implementation classes

Collection Type	Description
ArrayList	An indexed sequence that grows and shrinks dynamically
LinkedList	An ordered sequence that allows efficient insertions and removal at any location
HashSet	An unordered collection that rejects duplicates
TreeSet	A sorted set
EnumSet	A set of enumerated type values
LinkedHashSet	A set that remembers the order in which elements were inserted
PriorityQueue	A collection that allows efficient removal of the smallest element
HashMap	A data structure that stores key/value associations
TreeMap	A map in which the keys are sorted
EnumMap	A map in which the keys belong to an enumerated type
LinkedHashMap	A map that remembers the order in which entries were added
WeakHashMap	A map with values that can be reclaimed by the garbage collector if they are not used elsewhere
IdentityHashMap	A map with keys that are compared by ==, not equals

Choosing an implementation

- If you have lot of legacy code – use Hashtable, Vector, and Stack
- For List interface:
  - If you want to do many insertions and removals in the middle of a list a LinkedList is the appropriate choice
  - ArrayList is faster for other cases
  - If you work with threads – use Vector (all methods are synchronized)
- For Set interface:
  - choose HashSet by default, and change to ArraySet only in special cases
- For Map interface:
  - Iteration is faster over HashMap, data manipulation is faster with TreeMap

Futher reading

- http://docs.oracle.com/javase/1.5.0/docs/guide/language/generics.html
- http://gafter.blogspot.com/2004/09/puzzling-through-erasure-answer.html
- http://gafter.blogspot.com/2006/11/reified-generics-for-java.html
- http://docs.oracle.com/javase/6/docs/technotes/guides/collections/reference.html
- http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html

StringTokenizer

StringTokenizer class

- A **token** is a portion of a string that is separated from another portion of that string by one or more chosen characters (called **delimiters**).

- The **StringTokenizer** class contained in the java.util package can be used to break a string into separate tokens. This is particularly useful in those situations in which we want to read and process one token or one line of text file at a time; the BufferedReader class does not have a method to read one token at a time.

StringTokenizer constructors

StringTokenizer(String str)	Uses white space characters as a delimiters. The delimiters are not returned.
StringTokenizer(String str, String delimiters)	delimiters is a string that specifies the delimiters. The delimiters are not returned.
StringTokenizer(String str, String delimiters, boolean delimAsToken)	If delimAsToken is true, then each delimiter is also returned as a token; otherwise delimiters are not returned.

Some StringTokenizer methods

int countTokens( )	Using the current set of delimiters, the method returns the number of tokens **left**.
boolean hasMoreTokens( )	Returns true if one or more tokens remain in the string; otherwise it returns false.
String nextToken( ) throws NoSuchElementException	Returns the next token as a string. Throws an exception if there are no more tokens
String nextToken(String newDelimiters) throws NoSuchElementException	Returns the next token as a string and sets the delimiters to newDelimiters. Throws an exception if there are no more tokens.

example

```
StringTokenizer  tokenizer  =  new
 StringTokenizer(stringName);

 while(tokenizer.hasMoreTokens( )){
       String token =
                  tokenizer.nextToken();
    // process the token
    .   .   .
 }
```

example2

```
StringTokenizer  tokenizer  =  new
                    StringTokenizer (stringName);
 int  tokenCount  =  tokenizer.countTokens( );
 for(int k = 1; k <= tokenCount; k++){
       String  token  =  tokenizer.nextToken( );
       // process token
       .   .   .
 }
```

example 3 (1)

- Text file grades.txt contains ids and quiz grades of students

```
980000  50.0 30.0 40.0
975348  50.0 35.0
960035  80.0 70.0 60.0 75.0
950000  20.0 40.0
996245  65.0 70.0 80.0 60.0 45.0
987645  50.0 60.0
```

example 3 (2)

```
public static void main(String[] args) throws Exception {
  BufferedReader inputStream = new BufferedReader(new FileReader("grades.txt"));
  StringTokenizer tokenizer;  String inputLine, id;  int count;  float sum;

  System.out.println("ID#\tNumber of Quizzes\tAverage\n");

  while ((inputLine = inputStream.readLine()) != null) {
    tokenizer = new StringTokenizer(inputLine);
    id = tokenizer.nextToken();
    count = tokenizer.countTokens();
    sum = 0.0f;
    while (tokenizer.hasMoreTokens()){
      sum += Float.parseFloat(tokenizer.nextToken());
    }
    System.out.println(id + "\t" + count + "\t"+ sum / count);
  }
}
```