Interfaces

- One of the main objectives of Object-Oriented programming is software reuse.
- In this section we shall discuss how to make Java codes more reusable using *Interfaces* and *Generic Programming*.

Consider an example: the `DataSet` class

```
/**
   Computes information about a set of data values.
*/
public class DataSet
{
   private double sum;
   private double maximum;
   private int count;

   /**
      Constructs an empty data set.
   */
   public DataSet()
   {
      sum = 0;
      count = 0;
      maximum = 0;
   }

   /**
      Adds a data value to the data set
      @param x a data value
   */
   public void add(double x)
   {
      sum = sum + x;
      if (count == 0 || maximum < x)
         maximum = x;
      count++;
}
   /**
      Gets the average of the added data.
      @return the avg. or 0 if no data has been added
   */
```

```
    public double getAverage()
    {
       if (count == 0)
         return 0;
       return sum / count;
    }

    /**
       Gets the largest of the added data.
       @return the max. or 0 if no data has been added
    */
    public double getMaximum()
    {
       return maximum;
    }
}
```

- The `DataSet` class provides a service, namely computing the average and maximum of a set of input values.
- The class is suitable only for computing the average of a set of *numbers*.
- If we want to process bank accounts to find the bank account with the highest balance, we could not use the `DataSet` class in its current form.

Suppose the `DataSet` class is modified to suit our needs:

```
public class DataSet // Modified for BankAccount objects
{
   private double sum;
   private BankAccount maximum;
   private int count;

   ...    // Other methods not shown for brevity

   public void add(BankAccount x)
   {
     sum = sum + x.getBalance();
     if (count == 0 || maximum.getBalance() < x.getBalance())
        maximum = x;
     count++;
   }

   public BankAccount getMaximum()
   {
      return maximum;
   }
}
```

- Now, if we want to find the coin with the highest value among a set of coins. The `DataSet` class is modified again!

- Clearly, the algorithm for the data analysis service is the same in all cases, but the details of measurement differ.

- We would like to provide a single class that provides this service to any objects that can be measured. Classes could agree on a method **`getMeasure`** that obtains the measure to be used in the analysis.

In Java, an interface type is used to specify required operations (functionality):

```
public interface Measurable
{
   double getMeasure();   // abstract method
                          //  (without implementation)
}
```

A generic `DataSet` class for **Measurable** objects:

```
public class DataSet
{
   private double sum;
   private Measurable maximum;
   private int count;

   ...    // Other methods not shown for brevity

   //class of object x must implements the interface
   //Measurable
   public void add(Measurable x)
   {
      sum = sum + x.getMeasure();
      if (count == 0 ||
          maximum.getMeasure() < x.getMeasure())
         maximum = x;
      count++;
   }

   public Measurable getMaximum()
   {
      return maximum;
   }
}
```

An interface type is similar to a class, but there are several important differences:

- In Java 7 and earlier versions, all methods in an interface type are *abstract*; they don't have an implementation.
- In Java 8 an interface can have static and default methods (these methods are fully implemented).
- In Java 8 and earlier versions, all methods in an interface type are automatically **`public`**
- In Java 9, **`private`** methods are allowed in an interface (**`private`** method cannot be **`abstract`** or **`default`**)
- An interface type does not have instance variables, but it is legal to specify constants (variables in an interface are automatically **`public static final`**)

The `DataSet` class can be used to process bank accounts provided that the `BankAccount` class implements the `Measurable` interface

```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
     return balance;
   }
}
```

The `DataSet` class can also be used to process coins

```
public class Coin implements Measurable
{
   public double getMeasure()
   {
      return value;
   }
   ...
}
```

Interfaces can reduce the coupling between classes.

- Note that in the above example, `DataSet` is decoupled from `BankAccount` and `Coin`.

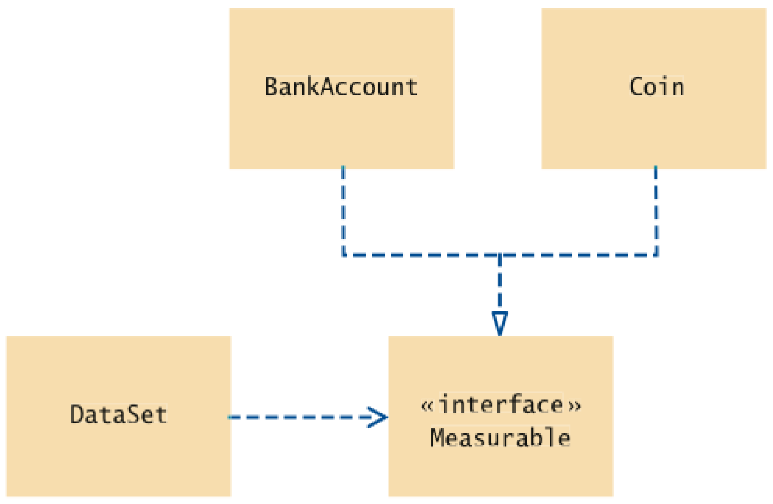


Figure 2 UML Diagram of the DataSet Class and the Classes that Implement the Measurable Interface

`BankAccount` is a `Measurable`
`Coin` is a `Measurable`

`DataSet` uses `Measurable`

Remark:

- You can define variable whose type is an interface, e.g.

```
Measurable x;
Coin c = new Coin();

x = c; //allowed if the class Coin implements Measurable

// x = new Coin();
```

- You cannot create an instance of an interface (except for anonymous class)

```
Measurable x = new Measurable();  //Syntax error
```

Generic Programming

Generic programming means to write code that can be reused for objects of different types.

```
//JDK 1.2, generic programming not supported
ArrayList files = new ArrayList(); // ArrayList stores a list
                                   // of objects (any type).

String sourceCode = "MyProgram.java";
files.add(sourceCode);
. . .

String sourceFile = (String)files.get(0); // require typecast

files.add(new Employee()); // action is allowed
```

Java 5 and later versions support generic classes

- A generic class is a class with one or more type variables.
- The type variable `E` is enclosed in angle brackets **`<E>`**.
- The type variable is used throughout the class definition to specify method return types, and the types of fields and local variables.
- The type variable `E` cannot be replaced by a primitive type (e.g. `int`, `double`, etc) in object instantiation.

```
// JDK 5.0 and later versions
// Create an array list of filenames.
// Elements in the array list must be a string

ArrayList<String> files = new ArrayList();
String sourceCode = "MyProgram.java";
files.add(sourceCode);
. . .

String sourceFile = files.get(0); // no type casting required

files.add(new Employee()); // compile time error
                           // type mismatch

// Similarly you can have an array list of Employee
ArrayList<Employee> e = new ArrayList();
```

User defined generic classes are allowed.

```
/**
  This class collects a pair of elements of different types.
*/
public class Pair<T, S>  // T and S are type variables
{
  private T first;
  private S second;

  /**
     Constructs a pair containing two given elements.
     @param firstElement the first element
     @param secondElement the second element
   */
   public Pair(T firstElement, S secondElement)
   {
     first = firstElement;
     second = secondElement;
   }

   /**
     Gets the first element of this pair.
     @return the first element
   */
   public T getFirst() {  return first;  }

   /**
     Gets the second element of this pair.
     @return the second element
   */
   public S getSecond() {  return second;  }

   public void setFirst(T element) {  first = element;  }

   public void setSecond(S element) {  second = element;  }

   public String toString() { return "(" + first + ", " + second + ")"; }
}
```

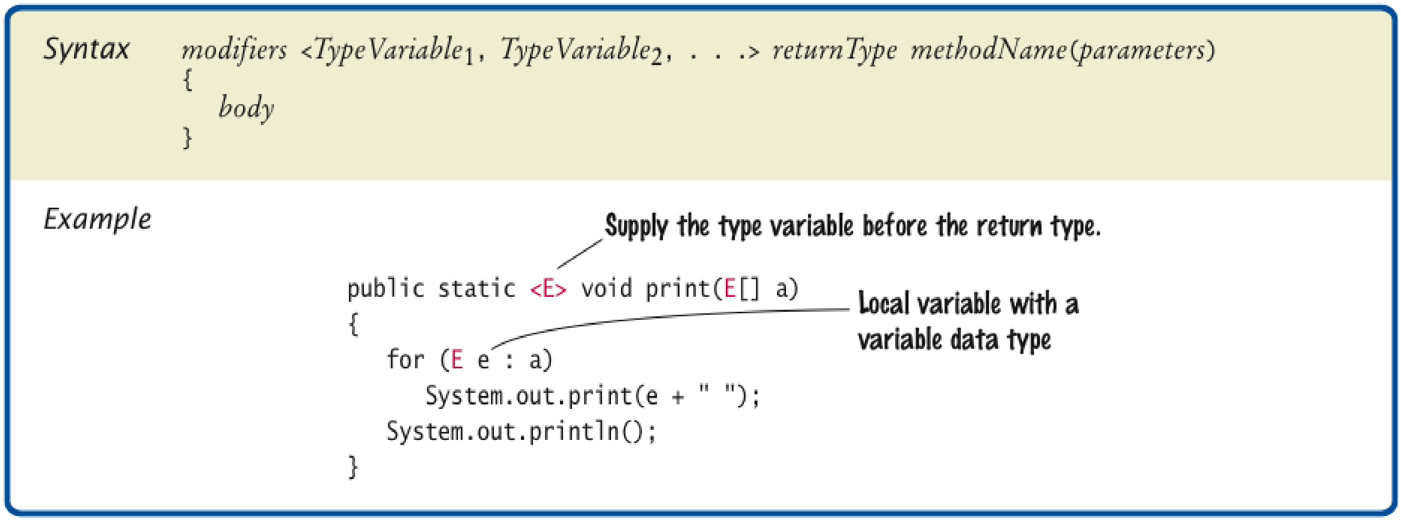
```
public class PairDemo
{
  public static void main(String[] args)
  {
    String[] names = { "Tom", "Diana", "Harry" };
    Pair<String, Integer> result = firstContaining(names, "a");
    System.out.println(result.getFirst());
    System.out.println("Expected: Diana");
    System.out.println(result.getSecond());
     System.out.println("Expected: 1");
   }

   /**
     Gets the first String containing a given string, together
     with its index.
     @param strings an array of strings
     @param sub a string
     @return a pair (strings[i], i) where strings[i] is the first
     strings[i] containing str, or a pair (null, -1) if there is no
     match.
   */
   public static Pair<String, Integer> firstContaining(
     String[] strings, String sub)
   {
     for (int i = 0; i < strings.length; i++)
     {
       if (strings[i].contains(sub))
       {
         return new Pair<String, Integer>(strings[i], i);
       }
     }
     return new Pair<String, Integer>(null, -1);
   }
}
```

Remark:

A class to model key-value pair, class Pair<K,V>, is defined in javafx.util
There is no setter method in the class.
https://openjfx.io/javadoc/14/javafx.base/javafx/util/Pair.html

Generic Methods

A generic method is a method with a type parameter.
Such a method can occur in a class that in itself is not generic.



Syntax *modifiers* <*TypeVariable*$_1$, *TypeVariable*$_2$, . . .> *returnType* *methodName*(*parameters*)

```
{
   body
}
```

Example

Supply the type variable before the return type.

```
public static <E> void print(E[] a)
{
   for (E e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Local variable with a variable data type

Example: declare a method that can print an array of any type:

```
public class ArrayUtil
{
   public static <E> void print(E[] a)
   {
      for (E e : a)  //for each element e (of type E) in a[] (a collection)
         System.out.print(e + " ");   // process element e
                                      // Remark: Cannot modify a[] within a for-each loop
      System.out.println();
   }
   . . .
}
```

When you call the generic method, you need not specify which type to use for the type parameter.

Simply call the method with appropriate parameters, and the compiler will match up the type parameters (target typing).

Example:

```
Rectangle[] rectangles = ...   //create an array of Rectangle

ArrayUtil.print(rectangles);   //The compiler deduces that E
                               //is Rectangle
```

Bounds for Type variables

Sometimes a class or a method needs to place restrictions on type variables.

Suppose we want to find the smallest element of an array

```
class ArrayUtil
{
   public static <E> E min(E[] a) //almost correct
   {
      E smallest = a[0];
      for (int i = 1; i < a.length; i++)
        if (smallest.compareTo(a[i]) > 0)
          smallest = a[i];

      return smallest;
   }
}
```

In the above example, the variable `smallest` has type `E`, which means that it could be an object of an arbitrary class.

How do we know that the class to which `E` belongs has a `compareTo` method?

The solution is to restrict `E` to a class that implements the `Comparable` interface, a standard interface with a single method, `compareTo`.

```
class ArrayUtil
{
   // correct implementation with type bound
   public static <E extends Comparable> E min(E[] a)
   {
      E smallest = a[0];
      for (int i = 1; i < a.length; i++)
        if (smallest.compareTo(a[i]) > 0)
          smallest = a[i];

      return smallest;
   }
}
```

The notation **`<E extends Comparable>`** expresses that `E` should be a subtype of the bounding type. (The actual meaning of this type bound is that the physical data type **`E`** must implement the **`Comparable`** interface).

Both `E` and the bounding type can be either a class or an interface.

The `extends` keyword was chosen because it is a reasonable approximation of the subtype concept, and the Java designers did not want to create a new keyword to the language.

A type variable can have multiple bounds, e.g.
`E extends Comparable & Serializable`

Getting back to our previous examples on the `Dataset` class

The `Dataset` class can be redesigned as a generic class.

```
public class DataSet<E extends Measurable>
{
   private double sum;
   private E maximum;
   private int count;
   ...


   public void add(E x)
   {
      sum = sum + x.getMeasure();
      if (count == 0 ||
          maximum.getMeasure() < x.getMeasure())
         maximum = x;
      count++;
   }

   public E getMaximum()
   {
      return maximum;
   }
}
```

Genericity and Inheritance

If `SavingsAccount` is a subclass of `BankAccount`,
is `ArrayList<SavingsAccount>` a subclass of `ArrayList<BankAccount>`?

The answer is: it is not.

Inheritance of type parameters does not lead to inheritance of generic classes.

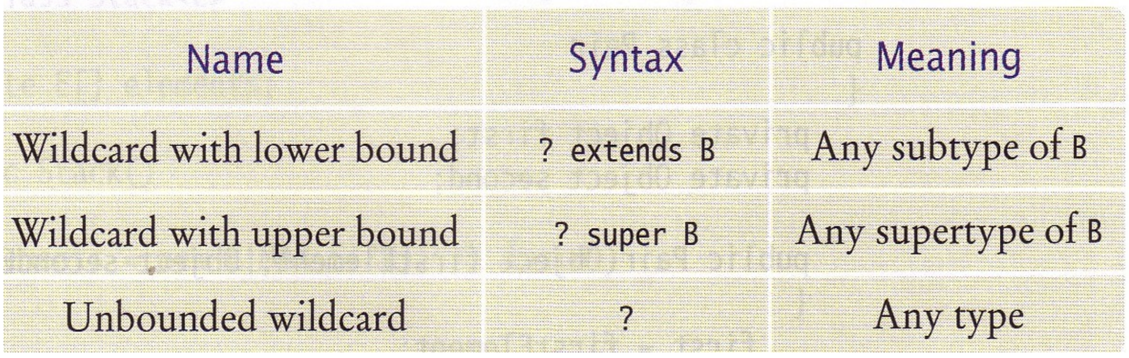
The following example codes illustrate reason behind the restriction.

```
ArrayList<SavingsAccount> sa = new ArrayList();

ArrayList<BankAccount> ba = sa; //Not legal

BankAccount barrysChecking = new CheckingAccount(); //allowed

ba.add(harrysChecking); //Adding a CheckingAccount object
                        //to an ArrayList of SavingsAccount!!
                        //Should NOT be allowed.
```

```
ArrayList<BankAccount> ba = new ArrayList();
BankAccount barrysChecking = new CheckingAccount(); //allowed
ba.add(harrysChecking);  //This action is allowed
```

Wildcard Types

It is often necessary to formulate subtle constraints of type parameters. Wildcard types are invented for this purpose.



Name	Syntax	Meaning
Wildcard with lower bound	`? extends B`	Any subtype of `B`
Wildcard with upper bound	`? super B`	Any supertype of `B`
Unbounded wildcard	`?`	Any type

type B or subclass of B

type B or superclass of B

Example:

```
public void addAll(LinkedList<? extends E> other)
{
  ListIterator<E> iter = other.listIterator();
  while (iter.hasNext())
    add(iter.next());
}
```

The `addAll` method doesn't require a specific type for the element type `other`. It allows you to use any type that is a subtype of `E`.

You can use `addAll` to add a `LinkedList<SavingsAccount>` to a `LinkedList<BankAccount>`.

Type Erasure

Generic types are a fairly recent addition to the Java language, the virtual machine that executes Java programs does not work with generic classes or methods.

Instead, type parameters are "erased", that is, they are replaced with its bound, or with `Object` if it is not bounded.

The generic class Pair<T, S> turns into the following raw class:

```
public class Pair
{
    private Object first;
    private Object second;

    public Pair(Object firstElement, Object secondElement)
    {
      first = firstElement;
      second = secondElement;
    }

    public Object getFirst() {   return first;  }
    public Object getSecond() {   return second;  }

    public void setFirst(Object element)
    {    first = element;   }

    public void setSecond(Object element)
    {    second = element;   }

}
```

Remark: type checking is only performed by the compiler.

Type erasure is also applied to generic methods:

```
   public static <E extends Comparable> E min(E[] a)
   {
      E smallest = a[0];
      for (int i = 1; i < a.length; i++)
        if (smallest.compareTo(a[i]) > 0)
          smallest = a[i];

      return smallest;
   }
```

This above generic method is type-erased to:

```
public static Comparable min(Comparable[] a)
{
    Comparable smallest = a[0];
    for (int i = 1; i < a.length; i++)
       if (smallest.compareTo(a[i]) > 0)
          smallest = a[i];

    return smallest;
}
```

Know about raw types helps you understand limitations of Java generics.

For example, trying to fill an array with copies of default objects would be wrong:

```
public static <E> void fillWithDefaults(E[] a)
{
   for (int i = 0; i < a.length; i++)
     a[i] = new E();  // Will not produce the expected effect
}
```

Type erasure yields:

```
public static void fillWithDefaults(Object[] a)
{
   for (int i = 0; i < a.length; i++)
     a[i] = new Object();  // Not useful
}
```

To solve this particular problem, you can supply a default type:

```
public static <E> void fillWithDefaults(E[] a,   E defaultValue)
{
   for (int i = 0; i < a.length; i++)
     a[i] = defaultValue;
}
```

You cannot construct an array of a generic type:

```
public class MyClass<E>
{
   private E[] elements;
   . . .
   public MyClass()
   {
      elements = new E[MAX_SIZE];  // Error
   }
}
```

Because the array construction expression new E[] would be erased to new Object[].

One remedy is to use an ArrayList<E> instead:

```
public class MyClass<E>
{
   private ArrayList<E> elements;
   . . .
   public MyClass()
   {
      elements = new ArrayList();  // Ok
   }
}
```

Sorting and Searching an array

The class `java.util.Arrays` provides a number of utility functions (static methods), such as binary search and sorting, to support the basic sorting and searching operations.

The following binary search and sorting functions require the object class implements the `Comparable` interface.

The method `compareTo()` in the `Comparable` interface defines the natural order of objects that belong to the given class.

```
static int binarySearch(Object[] a, Object key)

static void sort(Object[] a)
```

The class `Arrays` also provides generic binary search and sorting functions.

```
static <T> int binarySearch(T[] a, T key,
                            Comparator<? super T> c)

static <T> void sort(T[] a, Comparator<? super T> c)
```

To use the above generic functions, you need to provide a `Comparator` object (an instance of a class that implements the `Comparator` interface).

```
public interface Comparator<T>
{
   int compare(T o1, T o2);
   // return 0 if o1 == o2
   // return a positive value if o1 > o2
   // return a negative value if o1 < o2

   // Other static and default methods in the interface.

   // Will explain the static methods in the Comparator
   // interface later when we discuss functional programming.
}
```

Example design of the 2 versions of binary search method.

```java
// Actual class of the object implements Comparable
static int binarySearch(Object[] a, Object key)
{
   int low = 0;
   int high = a.length - 1;
   while (low <= high)
   {
      int mid = (low + high) / 2;
      int r = ((Comparable)key).compareTo(a[mid]);
      if (r == 0)
         return mid;
      if (r < 0)   // key < a[mid]
         high = mid - 1;
      else
         low = mid + 1;
   }
   return -(low + 1);   // -(insertion point + 1)
}
```

```java
static <T> int binarySearch(T[] a, T key,
                            Comparator<? super T> c)
{
   int low = 0;
   int high = a.length - 1;
   while (low <= high)
   {
      int mid = (low + high) / 2;
      int r = c.compare(key, a[mid]); // c is a functional
      if (r == 0)                     // object
         return mid;
      if (r < 0)   // key < a[mid]
         high = mid - 1;
      else
         low = mid + 1;
   }
   return -(low + 1);
}
```

Example to explain the type bound of `Comparator<? super T>`

```
// Comparator that can compare BankAccount
class BA_Comparator implements Comparator<BankAccount>
{
   @Override
   public int compare(BankAccount a1, BankAccount a2)
   {
      // compare BankAccount objects by balance
      if (a1.getBalance() < a2.getBalance())
         return -1;
      else if (a1.getBalance() > a2.getBalance())
         return 1;
      else
         return 0;
   }
}
```

```
int n = 100;
SavingsAccount[] sa_array = new SavingsAccount[n];
...
// statements to create and initialize the array


// Sort the array of SavingsAccount by balance.

Comparator cmp = new BA_Comparator();
Arrays.sort(sa_array, cmp);

// In this generic method invocation,
// type parameter T = SavingsAccount
// cmp is an instance of Comparator<BankAccount>
// BankAccount is a superclass of SavingsAccount
```

Example : basic coding style

```
class Emp_Comparator implements Comparator<Employee>
{
   @Override
   public int compare(Employee e1, Employee e2)
   {
      // compare Employee by name
      return e1.getName().compareTo(e2.getName());
   }
}
```

```
int n = 100;
Employee[] emp_array = new Employee[n];
...
// statements to create and initialize the Employee array

Comparator cmp = new Emp_Comparator();
Arrays.sort(emp_array, cmp);
```

Example: coding style using anonymous class

```
Comparator cmp = new Comparator<Employee>() {
               public int compare(Employee e1, Employee e2)
               {
                  return  e1.getName().compareTo(e2.getName());
               }
            };
Arrays.sort(emp_array, cmp);
```

```
// Another coding style using anonymous object
Arrays.sort(emp_array,
            new Comparator<Employee>() {
               public int compare(Employee e1, Employee e2)
               {
                  return  e1.getName().compareTo(e2.getName());
               }
            });
```

Defining a functional object using Lambda expression

Lambda expression is introduced in Java 8.

Lambda expression is used to define an implementation of a Functional interface (an interface with only 1 abstract method).

Basic syntax of Lambda expression

```
(arg1, arg2) -> single statement;

(Type arg1, Type arg2) -> { multiple statements };
```

The number of arguments depends on the abstract method defined in the interface.

Arguments are enclosed in parentheses and separated by commas.
The arguments correspond to the inputs required by the (abstract) method of the Functional interface.

The expression or statement block corresponds to the body of the method.

The data type of the arguments can be explicitly declared or it can be inferred from the context (i.e. the compiler will determine the target data type)

When there is a single argument, if its type is inferred, it is not mandatory to use parentheses, e.g. `(a) -> statement;` is the same as `a -> statement;`

Code snippet to sort an array of Employee using Lambda expression:

```
Arrays.sort(emp_array,
            (e1, e2) -> e1.getName().compareTo(e2.getName()));
```

Collection

A collection (called a container in C/C++) is simply an object that groups multiple elements into a single unit.

Collections are used to store, retrieve, manipulate, and communicate aggregate data.

A collection typically represents data items that form a natural group, such as a mail folder (a collection of letters), or a telephone directory (a mapping from names to phone numbers).

Typical data structures used to implement a collection include array, array list, linked list, stack, queue, tree, hash table, and etc.

A collection framework is a unified architecture for representing and manipulating collections.

Interfaces of the collection frameworks:

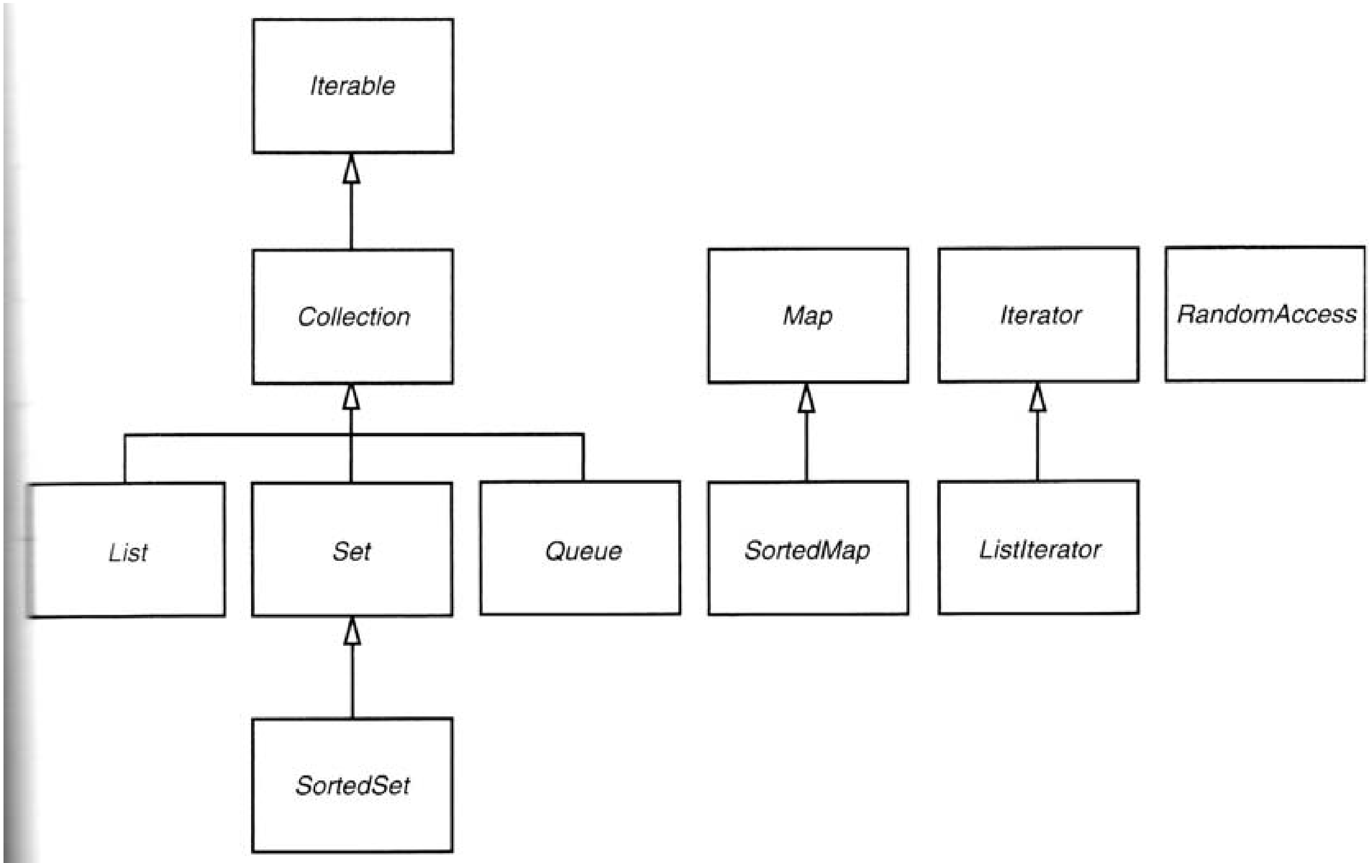


Figure 2–10: The interfaces of the collections framework

`Iterable` means that there is a mechanism to allow the user to visit each member in the collection one by one in a specific order.

The object that facilitates the iterated processing is called an `Iterator`.

All Methods Instance Methods Abstract Methods Default Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(**Collection**<? extends **E**> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(**Object** o) Returns true if this collection contains the specified element.
boolean	**containsAll**(**Collection**<?> c) Returns true if this collection contains all of the elements in the specified collection.
boolean	**equals**(**Object** o) Compares the specified object with this collection for equality.
int	**hashCode**() Returns the hash code value for this collection.
boolean	**isEmpty**() Returns true if this collection contains no elements.
Iterator<**E**>	**iterator**() Returns an iterator over the elements in this collection.
default **Stream**<**E**>	**parallelStream**() Returns a possibly parallel Stream with this collection as its source.
boolean	**remove**(**Object** o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll**(**Collection**<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
default boolean	**removeIf**(**Predicate**<? super **E**> filter) Removes all of the elements of this collection that satisfy the given predicate.
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this collection.
default **Spliterator**<**E**>	**spliterator**() Creates a **Spliterator** over the elements in this collection.
default **Stream**<**E**>	**stream**() Returns a sequential Stream with this collection as its source.
Object[]	**toArray**() Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

<u>Remark</u>: The methods `contains()`, `containsAll()`, `indexOf()`, `remove()`, `removeAll()`, `retainAll()` should be used with care when you are dealing with user defined object classes !!

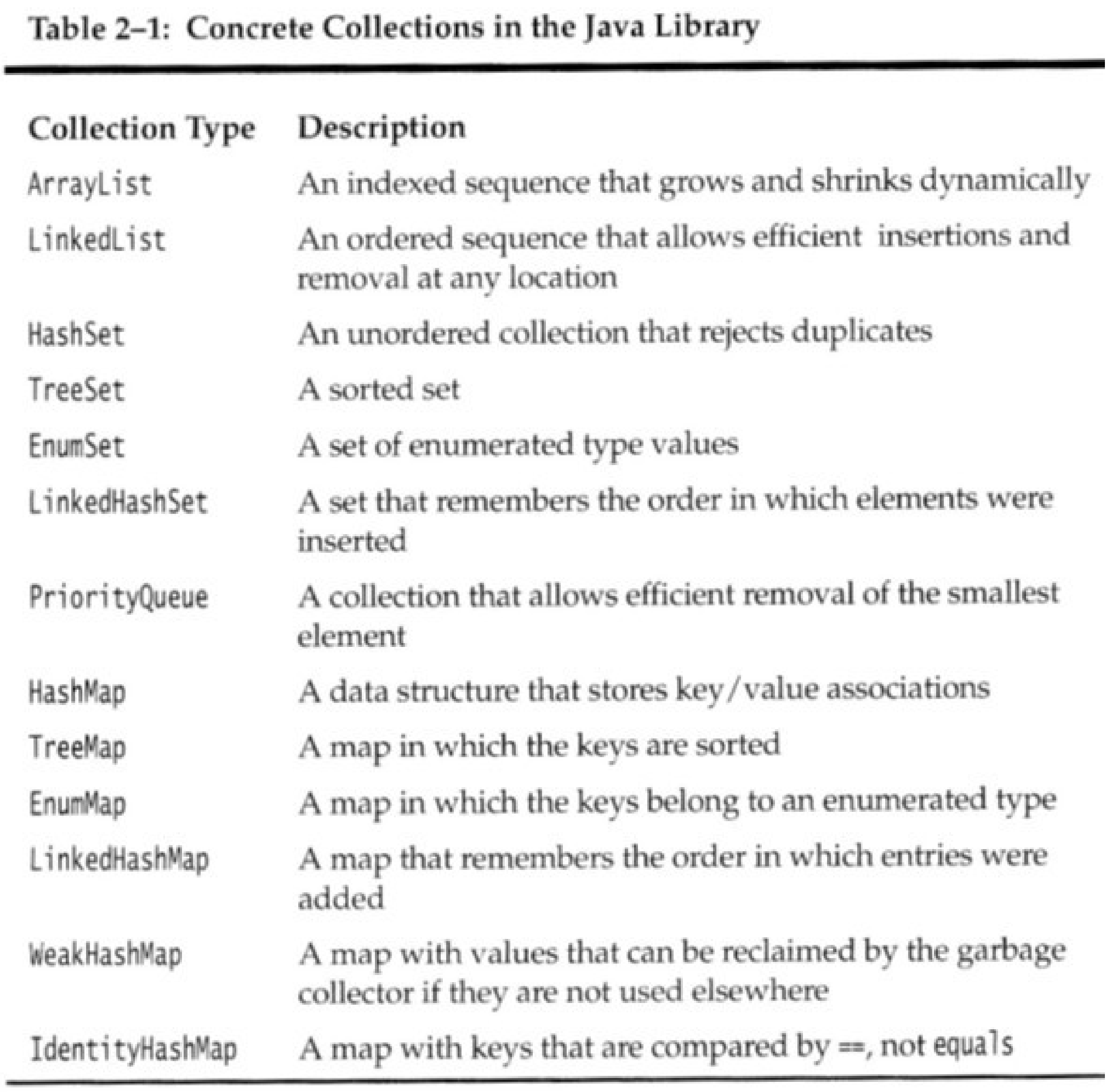


Table 2–1: Concrete Collections in the Java Library

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`

Methods defined in the class ArrayList

Modifier and Type	**Method and Description**
boolean	**add**(**E** e) Appends the specified element to the end of this list.
void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(**Collection**<? extends **E**> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this ArrayList instance.
boolean	**contains**(**Object** o) Returns true if this list contains the specified element.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
void	**forEach**(**Consumer**<? super **E**> action) Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(**Object** o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(**Object** o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(**Object** o)

	Removes the first occurrence of the specified element from this list, if it is present.
boolean	**removeAll**(**Collection**<?> c) Removes from this list all of its elements that are contained in the specified collection.
boolean	**removeIf**(**Predicate**<? super **E**> filter) Removes all of the elements of this collection that satisfy the given predicate.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
void	**replaceAll**(**UnaryOperator**<**E**> operator) Replaces each element of this list with the result of applying the operator to that element.
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this list that are contained in the specified collection.
E	**set**(int index, **E** element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
void	**sort**(**Comparator**<? super **E**> c) Sorts this list according to the order induced by the specified **Comparator**.
Spliterator<**E**>	**spliterator**() Creates a ***late-binding*** and *fail-fast* **Spliterator** over the elements in this list.
List<**E**>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
void	**trimToSize**() Trims the capacity of this ArrayList instance to be the list's current size.

ArrayList

- `ArrayList` class manages a sequence of objects
- Support random access, and can grow and shrink as needed.
- `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements
- `ArrayList<T>` is a **generic class**:

```
ArrayList<String> names = new ArrayList();
names.add("Emily");
names.add("Bob");
names.add("Cindy"); //new item added at the end of the list
```

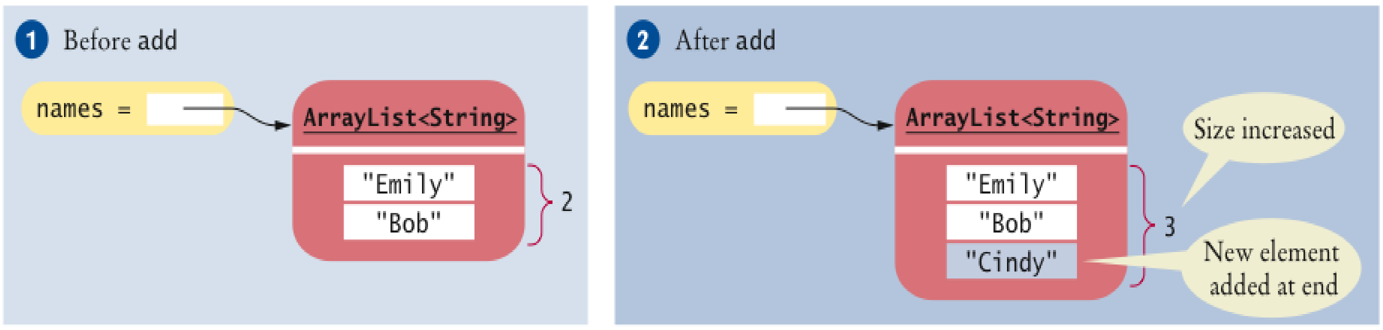


Figure 5 Adding an Element with add

- `size()` method yields number of elements
- To obtain the value of an element at an index, use the `get()` method
- Index starts at 0

```
String name = names.get(2);
// gets the third element of the array list
```

- Bounds error if index is out of range
- Most common bounds error:

```
int i = names.size();
name = names.get(i); // Error
// legal index values are 0 ... i-1
```

- To set an element to a new value, use the `set()` method:

```
names.set(2, "Carolyn");
```

- To remove an element at an index, use the `remove()` method:

```
names.remove(1);
```

Example:

```
names.add("Emily");
names.add("Bob");
names.add("Cindy");
names.set(2, "Carolyn"); //(1)
names.add(1, "Ann");   //(2), add new element at a given loc
                       //     loc <= names.size()
names.remove(1); //(3)
```

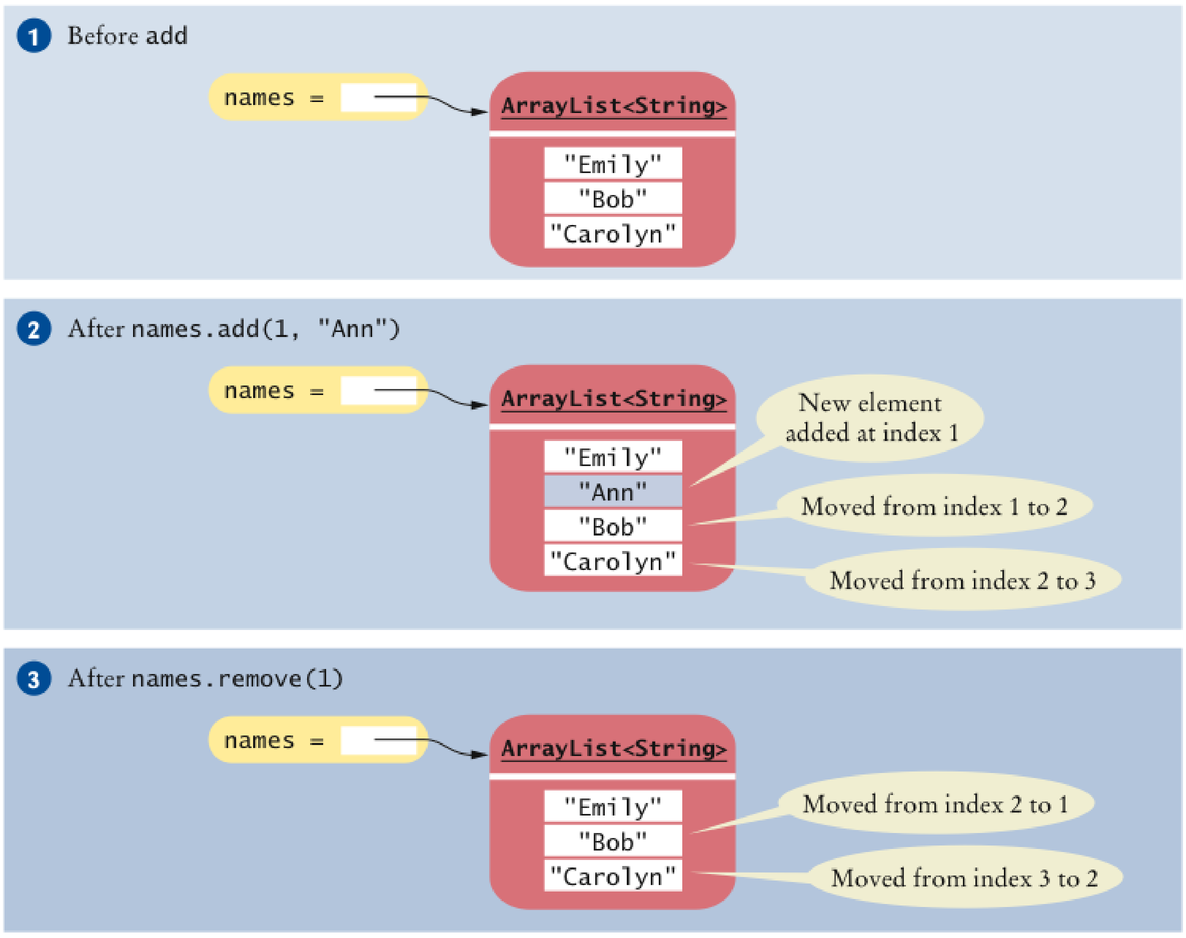


Figure 6 Adding and Removing Elements in the Middle of an Array List

`ArrayList` of numbers

- To collect numbers in an `ArrayList`, use the wrapper type as the type parameter, and then rely on auto-boxing:

```
ArrayList<Double> values = new ArrayList();
values.add(29.95);
double x = values.get(0);
```

- Storing wrapped numbers is quite inefficient
  - Acceptable if you only collect a few numbers
  - Use arrays for long sequences of numbers or characters

Remarks:

- `ArrayList` provides certain conveniences to the programmer
- Automatic grow and shrink, number of elements is equal to the size of the `ArrayList`.
- Support generic programming.
- Insertion and removal of elements at specific locations.
  - Remove the last element (or append a new element at the end) requires $O(1)$ time.
  - Remove the first element (or insert a new element at the front) requires $O(n)$ time.
- Auto-boxing of primitive types.
- Major disadvantage: slow

Processing elements in a collection one by one using an iterator

The `Iterator` interface has 3 methods

`boolean` **`hasNext()`**

- returns true if there is another element to visit.

`E` **`next()`**

- returns the next object to visit. Throws a `NoSuchElementException` if the end of the collection has been reached.

`void` **`remove()`**

- removes the last visited object. This method must immediately follow an element visit. If the collection has been modified since the last element visit, then the method throws an `IllegalStateException`.

To visit elements in the collection one by one:

```
Collection<String> c = ...;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
   String element = iter.next();
   //do something with element
}
```

As of JDK 5.0, you can use the "**for each**" loop to visit all elements in the collection:

```
Collection<String> c = ...;

for (String element : c) //for each element in c
{
   //do something with element
   //but you cannot modify the contents of the collection
   //when using the for-each loop
}
```

Example codes on the uses of the method `contains()`

```
ArrayList<String> list_s = new ArrayList();
String s1 = "ABC";
String s2 = "ABC123";

list_s.add(s1);

String s3 = s2.substring(0, 3); // s3 = "ABC"

/*
   Question: list_s.contains(s3) returns true or false ?
*/


ArrayList<StudentRecord> list_r = new ArrayList();
StudentRecord r1 = new StudentRecord("Bill", 12345, "MSMIT");

list_r.add(r1);

StudentRecord r2 = new StudentRecord(r1.getName(),
                       r1.getSid(), r1.getProgCode());

/*
   Question: list_r.contains(r2) returns true or false ?
*/
```

```
public class StudentRecord
{
   private String name;
   private int sid;
   private String progCode;
   ...

   @Override
   public boolean equals(Object other)
   {
      if (other instanceof StudentRecord)
      {
         StudentRecord r = (StudentRecord)other;
         return name.equals(r.name) && sid == r.sid &&
                progCode.equals(r.progCode);
      }
      else
         return false;
   }
}
```

```
class ArrayList<E> {
   public boolean contains(Object obj)
   {
      for (E e : this)
         if (e.equals(obj))
            return true;
      return false;
   }
}
```

```
Remark: if you override the method equals, you may also
        need to override the method hashCode.
```

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.
The `hashCode` method is used to support the implementation of hash table and hash map.

Linked List

In Java all linked lists are actually doubly linked; i.e. each node also stores a reference to its predecessor.

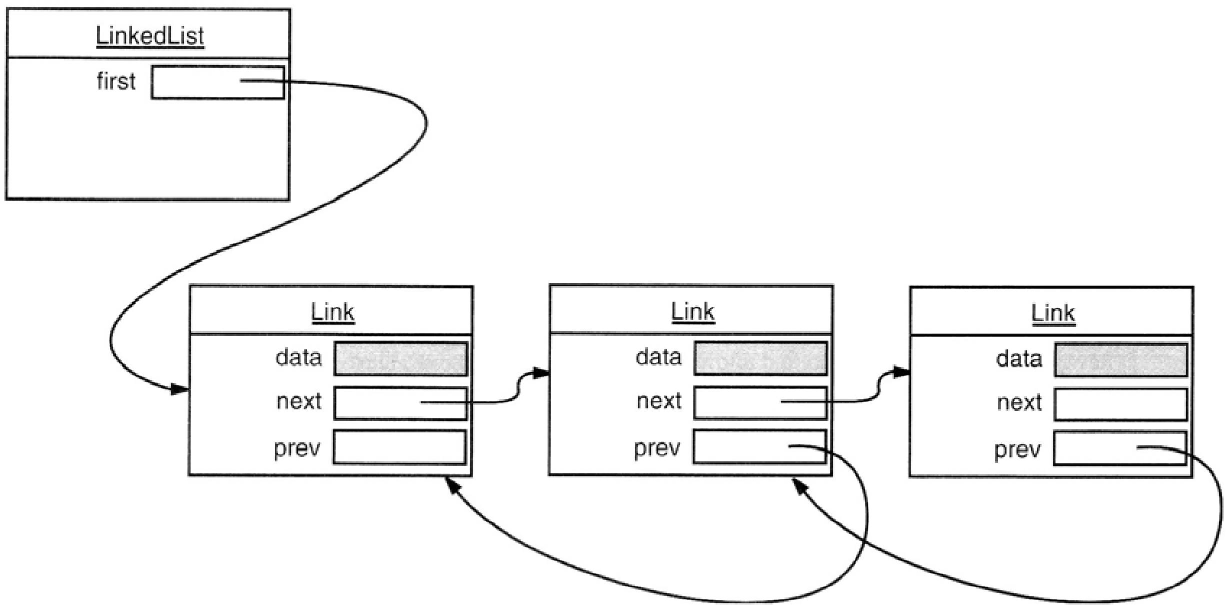


Figure 2–5: A doubly linked list

A linked list is an ordered collection in which the position of the objects matters. The `LinkedList.add` method adds the object to the end of the list.

There is no `add` method in the `Iterator` interface. Instead, the collection library provides a subinterface `ListIterator` that contains an `add` method and some other methods.

The `ListIterator.add` method allows you to add an element at specific location in the list.

```
interface ListIterator<E> extends Iterator<E>
{
   void add(E newElement);
   void set(E newElement);
   boolean hasPrevious();
   E previous();
   int nextIndex();
   int previousIndex();
}
```

You can think of Java iterators as being *between elements*.

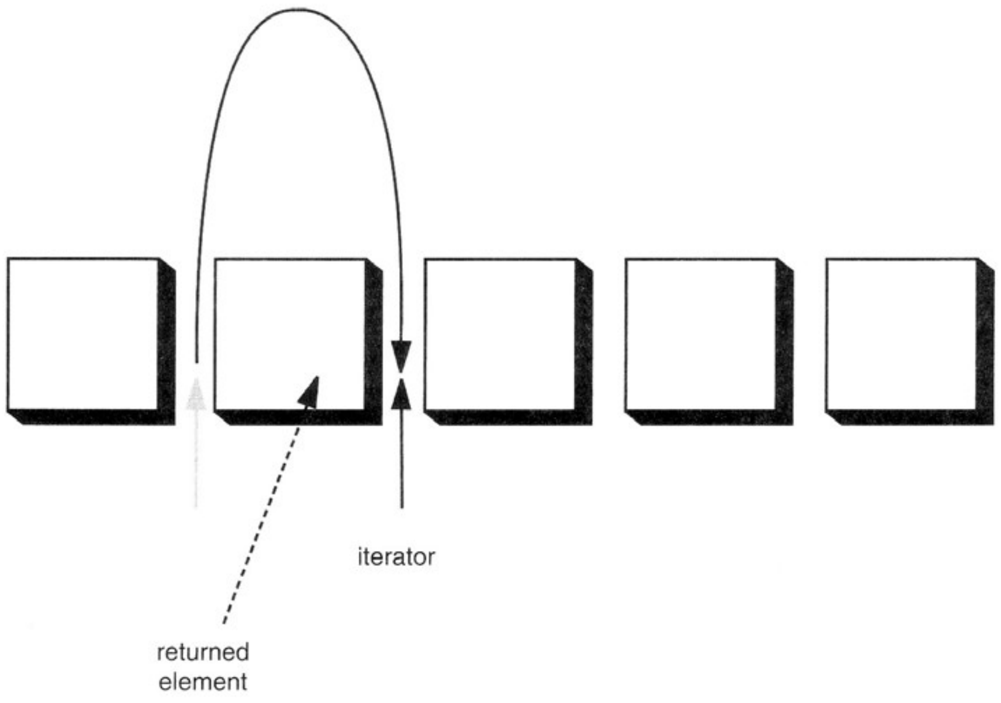


Figure 2–3: Advancing an iterator

When you call `next()`, the `iterator` jumps over the next element, and it returns a reference to the element that it just passed.

The `remove()` method of the `Iterator` interface removes the element that was returned by the last call to `next()`.

Example: to remove the first 2 elements

```
LinkedList<String> c = ...;
Iterator<String> iter = c.iterator();
iter.next();
iter.remove(); //remove the 1st element
iter.next();
iter.remove(); //remove the 2nd element
```

To add a new element after the element just visited (before the `iterator`).

```
List<String> staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next(); // skip past first element
iter.add("Juliet");
```

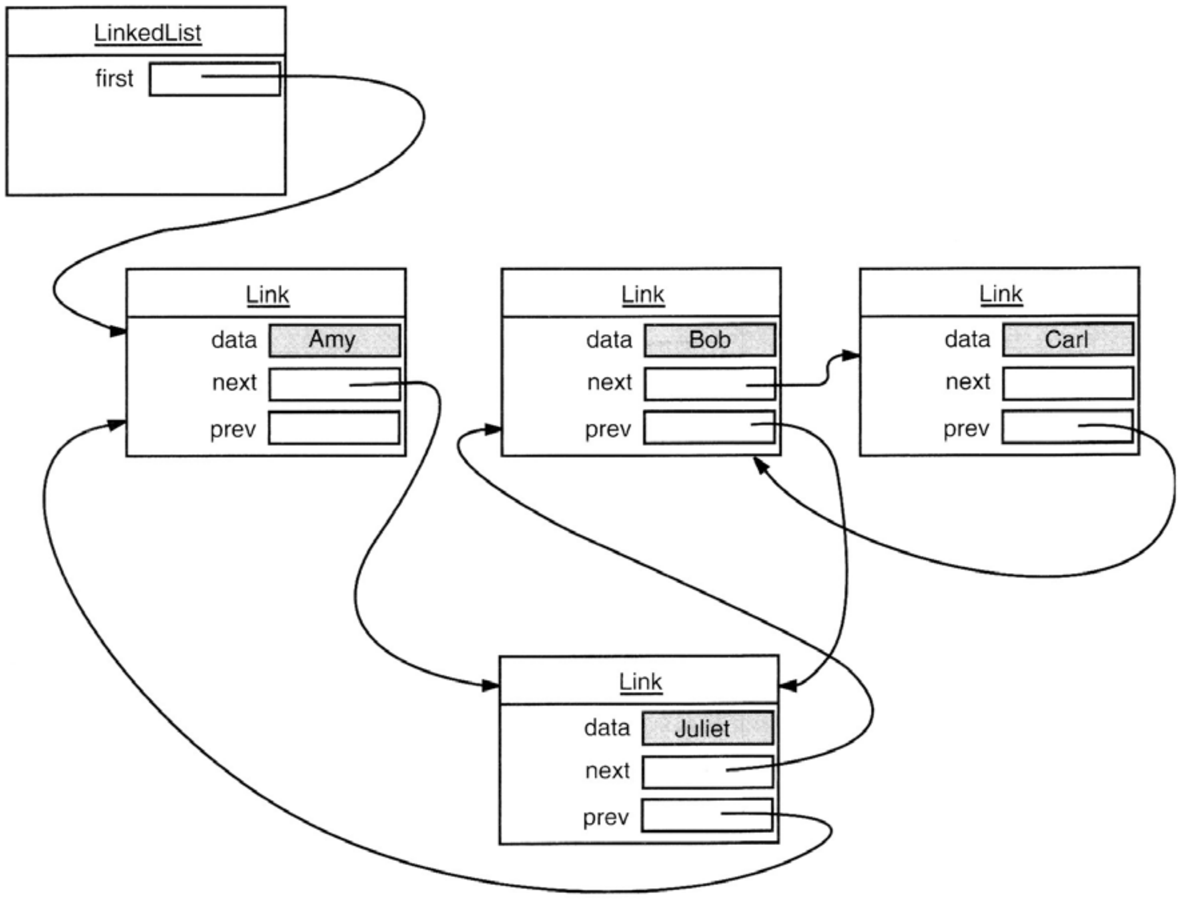


Figure 2–7: Adding an element to a linked list

More than one `iterator` can be attached to a collection.

If an `iterator` traverses a collection while another `iterator` is modifying the collection, confusing situations can occur.

```
List<String> list = ...;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove(); // remove first element
iter2.next();   // throws ConcurrentModificationException
```

General rules:

1. You can attach as many iterators to a collection as you like, provided that all of them are only readers; or
2. You can attach a single iterator that can both read and write.

Example 2–1: LinkedListTest.java

```
import java.util.*;

/**
   This program demonstrates operations on linked lists.
*/
public class LinkedListTest
{
   public static void main(String[] args)
   {
      List<String> a = new LinkedList<String>();
      a.add("Amy");
      a.add("Carl");
      a.add("Erica");

      List<String> b = new LinkedList<String>();
      b.add("Bob");
      b.add("Doug");
      b.add("Frances");
      b.add("Gloria");

      // merge the words from b into a

      ListIterator<String> aIter = a.listIterator();
      Iterator<String> bIter = b.iterator();

      while (bIter.hasNext())
      {
         if (aIter.hasNext()) aIter.next();
         aIter.add(bIter.next());
      }

      System.out.println(a);

      // remove every second word from b

      bIter = b.iterator();
      while (bIter.hasNext())
      {
         bIter.next(); // skip one element
         if (bIter.hasNext())
         {
            bIter.next(); // skip next element
            bIter.remove(); // remove that element
         }
      }

      System.out.println(b);

      // bulk operation: remove all words in b from a

      a.removeAll(b);

      System.out.println(a);
   }
}
```

Tree Sets

A tree set is a *sorted collection*.

You insert elements into a tree set in any order. When you iterate through the collection, the values are automatically presented in sorted order.

In the current version of Java collection library, tree set is implemented as a *red-black tree* (a form of balanced search tree).

Example:
```
public class TreeSetTest
{
   public static void main(String[] args)
   {
      TreeSet<String> sorter = new TreeSet();
      sorter.add("Bob");
      sorter.add("Amy");
      sorter.add("Carl");

      for (String s : sorter)
         System.out.println(s);
         //output: Amy Bob Carl
   }
}
```

By default, the tree set assumes that you insert elements that implement the `Comparable` interface, i.e. elements can be ordered using the `compareTo` method.

What if the object class does not implement the `Comparable` interface, or you want to sort the items using another attribute?

For example, the `compareTo` method of the `Student` class is based on the `ID` field, and you want to sort the collection of students using the `name` field.

In that case, you tell the tree set to use a different comparison method, by passing a `Comparator` object into the `TreeSet` constructor.

The `Comparator` object must implement the `Comparator` interface (which contains a generic method `compare`)

```
class StudentComparator implements Comparator<Student>
{
   public int compare(Student a, Student b)
   {
      return a.getName().compareTo(B.getName());
   }
}
```

You then pass a `Comparator` object to the tree set constructor

```
Comparator comp = new StudentComparator();
TreeSet<Student> sortByName = new TreeSet(comp);
```

```
interface Map<K, V>

class Hashtable<K, V> implements Map<K, V>,
                                 Serializable, Cloneable

class HashMap<K, V> implements Map<K, V>,
                               Serializable, Cloneable

class TreeMap<K, V> implements Map<K, V>,
                               Serializable, Cloneable
```

In many applications, we need to process (key, value) pairs.

Basic operations supported by a `Map<K, V>` object

```
V put(K key, V value)
V get(Object key)
V remove(Object key)
V replace(K key, V value)
int size()
boolean isEmpty()
boolean containsKey(K key)
boolean containsValue(V value)
```

The `Hashtable<K, V>` class implements the `Map<K ,V>` interface.
An instance of `Hashtable` has 2 parameters that affect its performance:

- initial capacity : number of buckets in the hash table
- load factor : $f = n / C$,
  n = number of keys, C = capacity of the hash table
- Keys in the map are distinct
- Keys and values cannot be null

The `HashMap<K, V>` class is roughly equivalent to `Hashtable`, except that it is unsynchronized and permits null values and null key.

- Unsynchronized : have safety concern in multi-thread program

`TreeMap<K, V>`

- Implementation based on search tree
- The map is sorted
- Guaranteed log(n) time for searching, insertion, and deletion