SFWR ENG/COMP SCI 2S03 Principles of Programming

Dr. Ridha Khedri

Department of Computing and Software, McMaster University
Canada L8S 4L7, Hamilton, Ontario

Acknowledgments: Material based on *Java actually: A Comprehensive Primer in Programming* (Chapter 15)

Topics Covered

program = algorithms + data structures

- The development of data structures for organizing data is as important as the algorithm
- We have already seen the data structures strings and arrays
- We discuss data structures that expand and shrink as data is inserted and retrieved
- They are called dynamic data structures

Definition

A collection is a structure for storing data.

- Collections have
  - different properties (depending on how the data is organized)
  - a set of operations that allow data to be added to or retrieved
- The operations comprise a contract that a client can use without regard to how the collection is implemented
- Such a collection is often called an abstract data type (ADT)

- Java provides ready-made classes that implement many dynamic data structures
  - StringBuilder
  - ArrayList
  - HashSet
  - HashMap

- Different data structures organize data in different ways
- In a normal list, the objects will be ordered in the sequence they were inserted (insertion order)
- A sorted list will require that the objects maintained in natural order
- In a normal set the order of the elements is irrelevant
- $\rightsquigarrow$ The most important operations on a dynamic data structure are to put data in it and retrieve data from it

- We look at dynamic data structures that store objects
- We handle objects with references
- There are variants that can store values of primitive data types
- We use generic types that allow us to customize collections with objects of particular types
- Data structures discussed are created in computer memory

Learning Objectives

- What an abstract data type (ADT) is
- Using a StringBuilder rather than a String to handle a character sequence
- What generic types are and how to use them
- How objects can be organized into collections
- How to use a list and a set
- What collection to use in a given situation
- How and when to use a map
- Implementing, calling, and using generic methods

Character sequences: StringBuilder

- The String class is not suitable if the characters in a string are modified frequently
- Each operation that modifies the contents of a String object actually returns a new String object
- The StringBuilder class can be used to represent a character sequence that can be modified/expands/shrinks dynamically
- We use the name string builder for such a character sequence
- A string builder is an object of the class StringBuilder

- A string builder keeps track
  - Its size: number of characters in its character sequence
  - Its capacity: the total number of characters that can be inserted (prior to expansion)
- Default capacity is 16 characters
- Other initial capacities can be specified:

```
StringBuilder buffer1 = new StringBuilder();    // length 0, capacity 16
StringBuilder buffer2 = new StringBuilder(20); //length 0, capacity 20
```

- A buffer is a storage place for keeping a limited amount of data in memory
- Objects of the class StringBuilder can be regarded as buffers
- String objects contain characters $\rightsquigarrow$ we can create a string builder from a String object

```
// String builder of length 4, capacity 20 (4+16), character sequence "mama":
  StringBuilder buffer3 = new StringBuilder("mama");
  String str1 = new String("mia");
// String builder of length 3, capacity 19 (3+16), character sequence "mia":
  StringBuilder buffer4 = new StringBuilder(str1);
```

- A string builder's length can be determined by calling the method length( )

```
int length = buffer4.length();
```

- A string builder's capacity can be determined by calling the method capacity( )

```
int capacity = buffer4.capacity();
```

- toString() method converts a string builder to a string

```
String str2 = buffer3.toString(); // "mama"
```

- To compare two string builders for equality, we must first convert them to strings

```
boolean flag = buffer3.toString().equals(buffer4.toString());// false
```

- Inserting characters in a string builder automatically results in adjustment to the capacity of the string builder (if necessary)
- The class provides positional access to the characters
- Method charAt() uses an index to read the character at that index
- Method setCharAt() uses an index to modify the character at that index
- Method deleteCharAt() uses an index to delete the character at that index

Modifying StringBuilder

```
// Assume that buffer4 contains the character sequence "mia"
char ch = buffer4.charAt (1);          // 'i'
buffer4.setCharAt(0,'P');              // "Pia"
buffer4.deleteCharAt(2);               // "Pi"
```

- The overloaded method append() can be used to append characters at the end of a string builder
- Values of primitive type, strings, character arrays and objects can be appended

```
buffer3.append(" mia ");               // "mama mia "
buffer3. append ('R');                 // "mama mia R"
buffer3.append(4);                     // "mama mia R4"
buffer3.append('U');                   // "mama mia R4U"
buffer3.append(new Integer(2));        // "mama mia R4U2"
```

- The overloaded method insert() can be used to insert characters at a particular index
- Any characters after the insertion index are shifted towards the end
- The method insert() can be used to insert string representations of different types of values

```
// Assume that buffer1 is empty
  buffer1.insert( 0, "Saturday");       // "Saturday"
  buffer1.insert(8, " March ");         // "Saturday March "
  buffer1.insert( 9, 1);                // "Saturday 1 March "
  buffer1.insert( buffer1.length(),
                  new Integer(2008));   // "Saturday 1 March 2008"
  buffer1.insert ( 10, '.');            //"Saturday 1. March 2008"
```

Method	Description
`int length()`	Returns the number of characters in the character sequence, i.e. the *length* of the character sequence.
`int capacity()`	Returns the current capacity of the string builder, i.e. the total number of characters that can be inserted currently, before the string builder is expanded.
`char charAt(int index)`	Returns the character at the `index` in the character sequence. The first character is at *index* 0.

Method	Description
`void setCharAt(int index, char ch)`	Replace character at the specified `index` with the character `ch`.

`void deleteCharAt(int index)`
`void delete(int startIndex,`
`int endIndex)`
	The first method deletes the character at the specified `index`. The second method deletes the characters from the specified `startIndex`, inclusive, to the specified `endIndex`, exclusive.

`int lastIndexOf(String subString)`
`int lastIndexOf(String subString,`
`int startIndex)`
	Returns the index of the start of the *last* occurrence of the specified `subString` in the character sequence, otherwise returns -1. Argument `startIndex` can be used to start the search from a particular index, otherwise the search starts at index 0.

`String substring(int startIndex)` `String substring(int startIndex, int endIndex)`	Returns a string containing the sequence from `startIndex` to `(endIndex-1)`. The returned string has length `(endIndex-startIndex)`. If no `endIndex` is specified, the sequence extends to the end of the character sequence.

`StringBuilder append(Object obj)`
`StringBuilder append(String str)`
`StringBuilder append(char t)`
`StringBuilder append(boolean b)`
`StringBuilder append(int i)`
`StringBuilder append(long l)`
`StringBuilder append(float f)`
`StringBuilder append(double d)`
	This overloaded method adds characters to the *end* of the character sequence. The number of characters appended depends on the parameter value which is first converted to a string representation, if necessary. The methods return the reference value of the updated `StringBuilder`.

`StringBuilder insert(int index, Object obj)`
`StringBuilder insert(int index, String str)`
`StringBuilder insert(int index, char t)`
`StringBuilder insert(int index, boolean b)`
`StringBuilder insert(int index, int i)`
`StringBuilder insert(int index, long l)`
`StringBuilder insert(int index, float f)`
`StringBuilder insert(int index, double d)`
	This overloaded method inserts characters at the specified `index`. The number of characters inserted depends on the parameter value, which is first converted to a string representation if necessary. The methods return the reference value of the updated `StringBuilder`.

`StringBuilder reverse()`	Replaces the character sequence with the result from reversing its characters and returns the reference of the object.
`String toString()`	Returns a `String` object that contains the current character sequence.
`void setLength(int newLength)`	Sets the length of the current string builder to the value of the parameter. To clear a string builder, pass the value 0.

- ADTs where we can replace the reference types are called generic types
- We can use the same definition of an ADT on different types of objects
- **Example:** A generic type that can be used to create pairs of values

```
1 class PairInt {
    private int first;
3   private int second;
    PairInt(int first, int second) {
5     this.first = first;
      this.second = second;
7   }
    // other methods
9 }
```

```
1  // Legacy class
   public class PairObj {
3    private Object first;
     private Object second;
5    PairObj () { }
     PairObj (Object first, Object second) {
7      this.first = first;
       this.second = second;
9    }
     public Object getFirst() { return first; }
11   public Object getSecond() { return second; }
     public void setFirst(Object firstOne) { first = firstOne; }
13   public void setSecond(Object secondOne) { second = secondOne; }

15 }
```

continues from previous slide

- Make sure that the object referred to by the Object reference is of the right type
- The programmer ensures that the objects being paired are of the same type

```
  public static void main(String[] args) {
    PairObj firstPair = new PairObj("Adam", "Eve");
    PairObj anotherPair = new PairObj("17. May", 1905);
    Object obj = firstPair.getFirst();
    if (obj instanceof String) {// Is the object of the right type?
      String str = (String) obj;// Type conversion to the subclass String.
      System.out.println(str.toLowerCase()); // Specific method in String.
    }
  }
}
```

Program output:

adam

- A generic class specifies one or more formal type parameters
- In defining a generic class, we can take a class where the Object type is utilized to generalize the use of the class

```
1  class Pair<T> {                          // (1)
     private T first;
3    private T second;
     Pair () { }
5    Pair (T first, T second) {
       this.first = first;
7      this.second = second;
     }
9    public T getFirst() { return first; }
     public T getSecond() { return second; }
11   public void setFirst(T firstOne) { first = firstOne; }
     public void setSecond(T secondOne) { second = secondOne; }
13   public String toString() {
       return "(" + first.toString() + "," + second.toString() + ")";    //
           (2)
15   }
   }
```

- Example: In the generic class Pair<T>, T is used in all the places where the type Object was used in the definition of the class PairObj
- Several formal type parameters are specified as a comma-separated list $< T_1, T_2 \cdots T_n >$
- **Note:** formal type parameters are not specified after the class name in the constructor declaration
- **Recommendation:** Use one-letter names for formal type parameters (e.g., T)

```
class Pair<F, S> {                              // (1)
  private F first;
  private S second;
  Pair () { }
  Pair (F first, S second) {
    this.first = first;
    this.second = second;
  }
  public F getFirst() { return first; }
  public S getSecond() { return second; }
  public void setFirst(F firstOne) { first = firstOne; }
  public void setSecond(S secondOne) { second = secondOne; }
  public String toString() {
    return "(" + first.toString() + "," + second.toString() + ")";    //
        (2)
  }
}
```

- Reference declaration, method calls, and object creation are done in the same way as for non-generic classes
- When we use a generic class, we specify the actual type parameters (e.g., Pair<String>, Pair<Integer>)
- The compiler verifies that parameterized types are used correctly in the source code

```
2 public class ParameterizedTypes {

4   public static void main(String[] args) {
      Pair<String> strPair = new Pair<String>("Adam", "Eve");   // (1)
6 //  Pair<String> mixPair = new Pair<String>("17. May", 1905);// (2)
      Error!
      Pair<Integer> intPair = new Pair<Integer>(2005, 2010);    // (3)
8 //  strPair = intPair;                    // (4) Compile-time error!
      Pair<String> tempPair = strPair;  // (5) OK
10
      strPair.setFirst("Ole");     // (6) OK. Only String accepted.
12 //  intPair.setSecond("Maria"); // (7) Compile-time error!
      String name = strPair.getSecond().toLowerCase();  // (8) "eve"
14    System.out.println(name);
    }
16 }
```

- Only one implementation of a generic class + values of primitive types have different sizes
  $\rightsquigarrow$ primitive data types are not allowed as actual type parameters

```
2 public class ParameterizedTypesPrb {

4   public static void main(String[] args) {
      //Pair<int> intPair = new Pair<int>(3, 7);  // ERROR: CANNOT use
          primitive data types
6     Pair<Integer> intPair = new Pair<Integer>(2005, 2010);   // (3)
//      int index = intPair.getSecond();  // ERROR
8       Integer index = intPair.getSecond();
      System.out.println(index);
10  }
}
```

- A contract specifies the behaviour of a class
  - the parameter values accepted
  - the values returned by the methods
  - special conditions in the contract of each method: preconditions, and postconditions
- In the documentation comment of a method, we use special Javadoc tags, @pre and @post, to specify the contract
- These conditions are tested using assertions at relevant places in the code

```
/**
 * Calculates the weekly salary for the employee.
 * @param numHours Number of hours worked in current week.
 * @return Weekly salary in GBP for the current week.
 *
 * @pre numHours >= 0.0
 * @post salary >= 0.0
 */
double calculateSalary(double numHours) {
  assert numHours >= 0.0 : "Number of hours must be >= 0.0";    // @pre
  double salary = ...
  ...
  assert salary >= 0.0 : "Weekly salary must be >= 0.0";        // @post
}
```

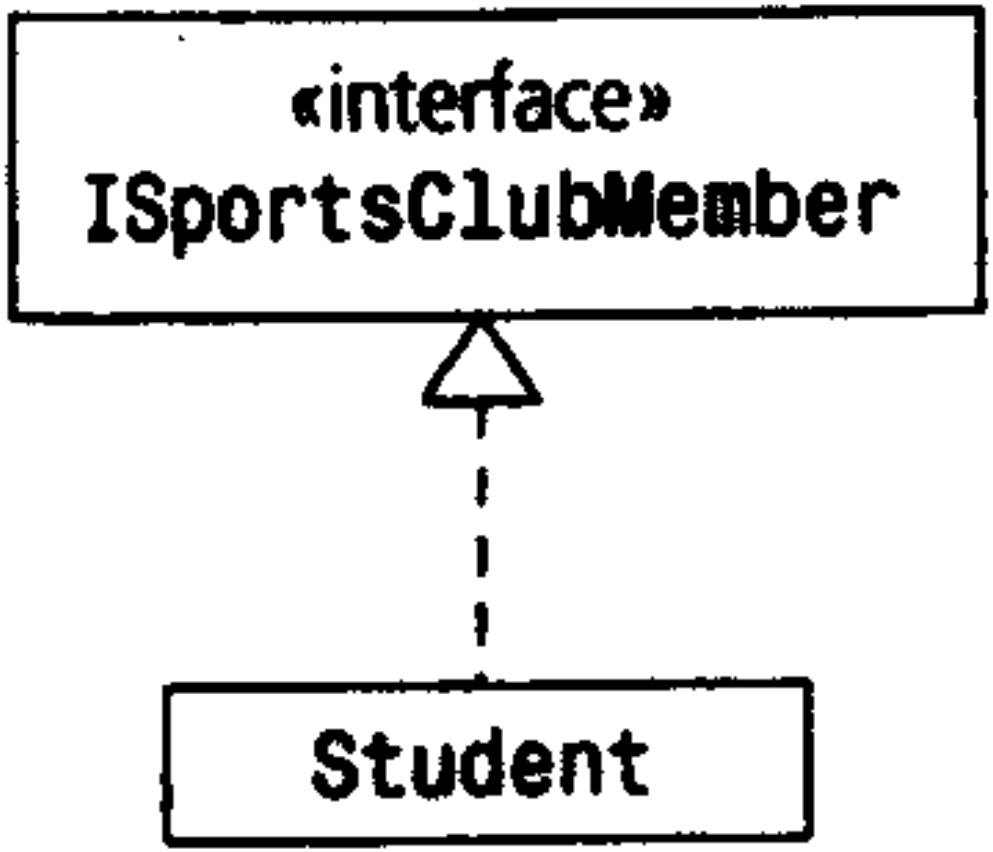
- An interface in Java specifies a group of related methods, but without their implementation
- The interface only specifies the contract for each method (parameter values and the value returned)
- There is no implementation of the method body (Called abstract method)
- The keyword interface is used to designate an interface specification
- An interface allows us to formalize a contract (contract conditions are only embedded as documentation)

- We cannot create objects from interfaces
- An interface can declare constants (implicitly, static and final)

```
interface ISportsClubMember {
   /**
    * Calculates the membership fee.
    * @post fee >= 0.0
    */
   double calculateFee();   // (1) Abstract method
}
```

For a class to implement the contract specified by an interface:

- The class must specify the name of the interface in its class header using the implements clause
- We say that the class implements the interface
- The class must also implement all the abstract methods of the interface
- If it does not, then the class must be declared abstract
- The class can provide additional services



```
class Student implements ISportsClubMember {                    // (1)

  // Field variables
  String firstName;
  String lastName;
  int credits;

  // Constructors
  /**
   * Creates a student with the specified name and credits.
   */
  public Student(String firstName, String lastName, int credits) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.credits = credits;
  }

  /**
   * Prints the name and number of credits for the student.
   */
```

```
  void printState() {
    System.out.printf("Student %s %s has %d credits.%n",
                      firstName, lastName, credits);
  }

  // Implements the interface
  /**
   * Calculates the membership fee.
   * @post return >= 0.0
   */
  public double calculateFee() {                                    // (2)
    double fee = 10.0;
    assert fee >= 0.0 : "Membership fee for students must be >= 0.0"; // @post
    return fee;
  }
  // Other members...
}
```

- We can also declare generic interfaces
- Can then be customized by supplying specific types as actual type parameters

```
1 interface PairRelationship<T> {
    T getFirst();
3   T getSecond();
    void setFirst(T firstOne);
5   void setSecond(T secondOne);
  }
```

```
class Pair<T> implements PairRelationship<T> {
      // same as before
}
```

- We can parameterize a generic interface as we did with a generic class

```
PairRelationship<String> oneStrPair = new Pair<String>( "Eve", "Adam");
```

- The Java standard library contains many generic interfaces (e.g., in java.lang.Comparable)

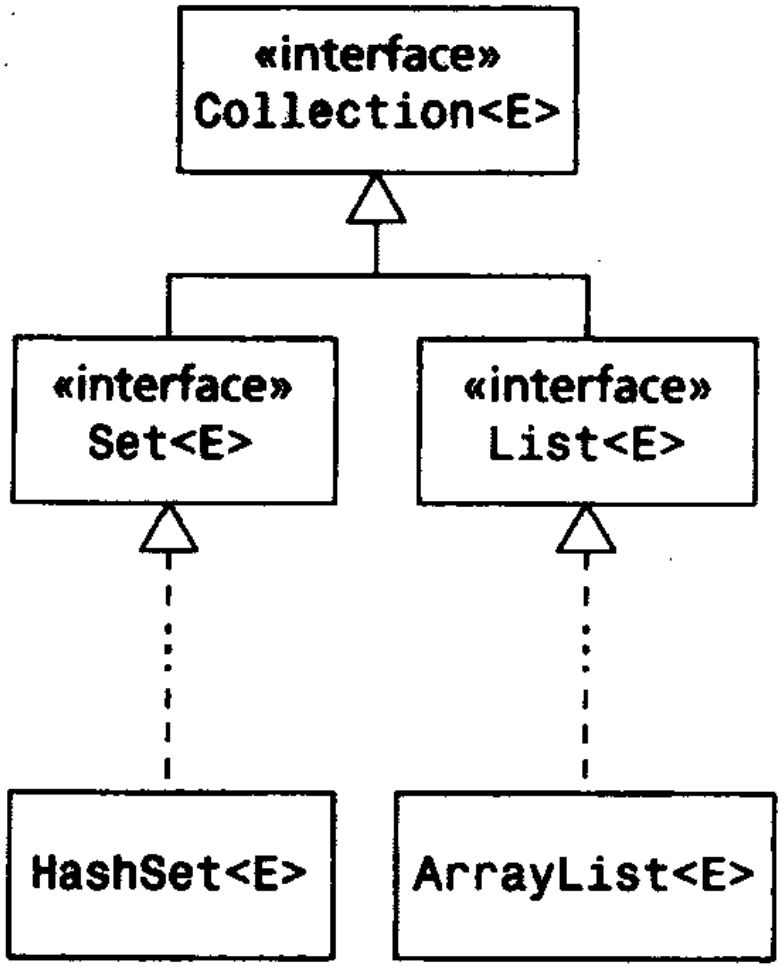
```
public interface Comparable<T> {
    int compareTo(T obj);
}
```

- Collections are a part of the java.util package in Java
- A collection provides a contract in the form of an interface that a client can use to interact with the collection
- The client uses the methods of the interface and references of the interface type to handle the collection
- The clients need not know which concrete implementation is being used for the collection
- The implementation of the collection can change without consequences for the client

- Collections in the java.util package are implemented as generic types
- This means that
  - we must specify what type of objects a collection will contain when we create it
  - the collections we create are customized for particular types of objects
- The compiler makes sure that each collection is handled correctly

Superinterface **Collection<E>**

E: element type (the type of the objects in the collection)



Basic operations from the Collection<E> interface

Method	Description
`int size()`	Returns the number of elements in the collection.
`boolean isEmpty()`	Determines whether the collection is empty.
`boolean contains(Object element)`	Determines whether the specified `element` is in the collection.
`boolean add(E element)`	Tries to *insert* the specified `element` into the collection and returns `true` if successful.
`boolean remove(Object element)`	Tries to *delete* the specified `element` from the collection and returns `true` if successful.
`Iterator<E> iterator()`	Returns an iterator that can be used to traverse the collection.

Bulk operations from the Collection<E> interface

Method	Description
`boolean containsAll(Collection<?> c)`	*Subset*: returns `true` if all the elements in the specified collection `c` are also in the current collection.
`boolean addAll(Collection<? extends E> c)`	*Union*: adds all the elements from the specified collection `c` to the current collection, and returns `true` if the current collection was modified.
`boolean retainAll(Collection<?> c)`	*Intersection*: only elements from the specified collection `c` are retained in the current collection, the rest being removed from the current collection. Returns `true` if the current collection was modified.
`boolean removeAll(Collection<?> c)`	*Difference*: elements from the specified collection `c` are removed from the current collection. Returns `true` if the current collection was modified.
`void clear()`	Removes all elements from the current collection.

- An important operation on a collection is to traverse the collection
- Collections from the Java standard library provide an iterator for this purpose
- The iterator implements a Java interface that defines the methods for traversing a collection

interface Iterator<E>

Method	Description
`boolean hasNext()`	Returns `true` if the underlying collection has elements left to iterate over.
`E next()`	Returns the next element in the underlying collection and increments the iterator. If there are no more elements left, throws a `NoSuchElementException`.
`void remove()`	Deletes the current element from the underlying collection. This method can be called only once after each call of the `next()` method.

```
// Create a list of strings.
Collection<String> collection = new ArrayList<String>();
collection.add("9");                    // Add elements.
collection.add("1");
collection.add("1");

Iterator<String> iter = collection.iterator(); // Get an iterator.
while (iter.hasNext()) {                // More elements in the collection?
  System.out.print(iter.next());        // Print the current element.
}
```

- If the method hasNext() returns true, we can call the method next()
  - return the next element in the collection
  - to increment the iterator
- It is not necessary to increment the iterator explicitly

Traversing a collection using a **for(:)**

- A collection can also be traversed with a for(:) loop, **if it implements the java.lang.Iterable<E> interface**

```
interface Iterable<E> {
        Iterator<E> iterator();
}
```

- The method iterator() returns an iterator that implements the interface Iterator<E>

- The interface Iterable<E> says that if a collection has an iterator, we can traverse the collection with a for(:) loop

- A list (or sequence) is a collection in which each element has a position
- A list also allows duplicates
- The List<E> interface extends the Collection<E> superinterface for lists
- The class ArrayList<E> in the java.util package implements lists
- $\rightsquigarrow$We can execute all operations in the List<E> interface (including the inherited ones)

Selected list operations from the List<E> interface

Method	Description
`E get(int index)`	Returns the element at the specified `index`.
`E set(int index, E element)`	Replaces the element at the specified `index` with the specified `element`. Returns the element that was replaced.
`E remove(int index)`	Removes and returns the element at the specified `index`. Elements in the list are shifted if necessary.
`void add(int index, E element)`	Inserts the specified element at the specified index. Elements in the list are shifted if necessary.
`int indexOf(Object element)`	Returns the index of the first occurrence of an element in the list that is equal to the specified `element` according to the `equals()` method, or `-1` if there is no such element.

- In a list the method **size()** (from the Collection<E> interface) returns the length of the list
- There are many ways to implement a list
  - The implementation in the class ArrayList<E> is known as a resizable array
  - There is also an implementation that is based on linked lists

```
import java.util.ArrayList;
import java.util.List;

public class ListClient {

  static final String CENSORED = "CENSORED";

  public static void main(String args[]) {
    // (1) Read words from the command line into a word list:
    List<String> wordList = new ArrayList<String>(); // <----------
    for (int i = 0; i < args.length; i++)
      wordList.add(args[i]);
    System.out.println("Original word list: " + wordList);

    // (2) Create a list with words that are censored:
    List<String> censoredWords = new ArrayList<String>();
    censoredWords.add("fun");
    censoredWords.add("cool");
    censoredWords.add("easy");

    // (3) Censor the word list:
    for (String element : wordList) {
      if (censoredWords.indexOf(element) != -1) {
        int indexInWordList = wordList.indexOf(element);
        wordList.set(indexInWordList, CENSORED);
      }
    }

    // (4) Print the censored list:
    System.out.println("Censored word list: " + wordList);
  }
}
```

- The interface Set<E> models the mathematical concept set $(\cup, \cap, -)$
- A set does not permit duplicates + there is no order to the elements
- The Set<E> interface uses the methods already found in the Collection<E> interface
- HOWEVER, it specializes the existing ones for sets
- The HashSet<E> class implements the Set<E> interface
- We will use the **class** HashSet<E> to work with sets

```
import java.util.HashSet;
import java.util.Set;

public class SetClient {
  public static void main(String args[]) {
    // (1) Create two sets for concerts:
    Set<String> concertA = new HashSet<String>(); // <----------
    Set<String> concertB = new HashSet<String>();  // <----------
    System.out.println("(1) Created an empty concert A: " + concertA);
    System.out.println("(1) Created an empty concert B: " + concertB);

    // (2) Add artists to the concerts:
    concertA.add("aha"); concertA.add("madonna");
    concertA.add("abba"); concertA.add("kiss");
    concertA.add("Sivle");
    System.out.println("(2) Artists in concert A: " + concertA);

    concertB.add("TLC"); concertB.add("madonna");
    concertB.add("abba"); concertB.add("wham");
    System.out.println("(2) Artists in concert B: " + concertB);

    // (3) Duplicates are not allowed in sets:
    String artist = "TLC";
    if (concertB.add(artist))
      System.out.println("(3) Artist " + artist +
                         " registered for concert B.");
    else
      System.out.println("(3) Artist " + artist +
                         " already registered for concert B.");

// ... Continued on the next slide ....
```

Sets: HashSet<E>


```
1  // ... Continuous from previous slide
2
3      // (4) Subset: determine whether all artists that performed
4      //     in concert B also performed in concert A:
5      if (concertA.containsAll(concertB))
6        System.out.println("(4) All artists in concert B performed" +
7                           " in concert A.");
8      else
9        System.out.println("(4) Not all artists in concert B performed" +
10                          " in concert A.");
11
12     // (5) Union: find all artists:
13     Set<String> allArtists = new HashSet<String>(concertA);
14     allArtists.addAll(concertB);
15     System.out.println("(5) All artists: " + allArtists);
16
17     // (6) Intersection: find artists that performed in both concerts:
18     Set<String> bothConcerts = new HashSet<String>(concertA);
19     bothConcerts.retainAll(concertB);
20     System.out.println("(6) Artists that performed in both concerts: " +
21                        bothConcerts);
22
23     // (7) Difference: find artists that performed in concert A only:
24     Set<String> onlyConcertA = new HashSet<String>(concertA);
25     onlyConcertA.removeAll(concertB);
26     System.out.println("(7) Artists that performed in concert A only: "
27         +
28                        onlyConcertA);
29     // (8) Difference: find artists that performed in concert B only:
30     Set<String> onlyConcertB = new HashSet<String>(concertB);
31     onlyConcertB.removeAll(concertA);
32     System.out.println("(8) Artists that performed in concert B only: "
33         +
34                        onlyConcertB);
35   }
36 }
```

Program Output

```
(1) Created an empty concert A: []
(1) Created an empty concert B: []
(2) Artists in concert A: [kiss, abba, madonna, Sivle, aha]
(2) Artists in concert B: [abba, madonna, wham, TLC]
(3) Artist TLC already registered for concert B.
(4) Not all artists in concert B performed in concert A.
(5) All artists: [kiss, abba, madonna, TLC, wham, aha, Sivle]
(6) Artists that performed in both concerts: [abba, madonna]
(7) Artists that performed in concert A only: [kiss, aha, Sivle]
(8) Artists that performed in concert B only: [TLC, wham]
```

Maps: HashMap<K, V>

- A map (or hash table) is used to store entries (bindings or mappings) (i.e., Functions)
- Each entry is a pair of objects: the first object, called the key, $\longleftrightarrow$ second object, called the value
- Examples:
  - A tel. list: telephone # (key) $\longleftrightarrow$ a name (value)
  - Employee $\longleftrightarrow$ number of hours worked in a week
- The relation between keys and values in a map is a many-to-one relation: Non injective function

Illustration of a map

Word (key)	Frequency (value)
to	2
be	4
or	1
not	1
what	1
will	2

- A hash function is any algorithm or subroutine that maps large data sets of variable length, called keys, to smaller data sets of a fixed length
- Storage and retrieval of entries in a map requires using hashing
- This integer value, called the hash value employed to
  - store the key (and its value)
  - retrieve the entry of a key in the map

- This technique is called hashing
- **Example:** a person's name, having a variable length, could be hashed to a single integer
- The hash function must be uniform
  - map the expected inputs as evenly as possible over its output range
  - Reason: the cost of hashing-based methods goes up sharply as the number of collisions increases

- In Java, the method hashCode() is used to compute the hash value of an object
- This method is defined in the Object class, but must be overridden if the objects of a class are to be used in a map
- The hash value of an object must satisfy:
  - The hash code must always be the same for an object as long as its state has not changed
  - Two objects that are equal according to the equals( ) method must have the same hash value

```
1  /**
   * The Point3D_V1 class represents a point in space,
3  * but it overrides neither the equals() method nor the hashCode()
       method.
   */
5 class Point3D_V1 {                                                  // (1)
    int x;
7   int y;
    int z;
9
    Point3D_V1(int x, int y, int z) {
11    this.x = x;
      this.y = y;
13    this.z = z;
    }
15
    public String toString() { return "["+x+","+y+","+z+"]"; }
17 }
```

Maps: HashMap<K, V>


```
1  /**
   * The Point3D class represents a point in space,
3  * and it overrides both the equals() method and the hashCode() method.
   */
5  class Point3D {                                                    // (2)
     int x;
7    int y;
     int z;
9
     Point3D(int x, int y, int z) {
11     this.x = x;
       this.y = y;
13     this.z = z;
     }
15
     public String toString() { return "["+x+","+y+","+z+"]"; }
17
     /** Two points are equal if they have the same x, y and z coordinates.
         */
19   public boolean equals(Object obj) {                               // (3)
       if (this == obj) return true;
21     if (!(obj instanceof Point3D)) return false;
       Point3D p2 = (Point3D) obj;
23     return this.x == p2.x && this.y == p2.y && this.z == p2.z;
     }
25
     /** The hash value is computed based on the coordinate values. */
27   public int hashCode() {                                           // (4)
       int hashValue = 11;
29     hashValue = 31 * hashValue + x;
       hashValue = 31 * hashValue + y;
31     hashValue = 31 * hashValue + z;
       return hashValue;
33   }
   }
```

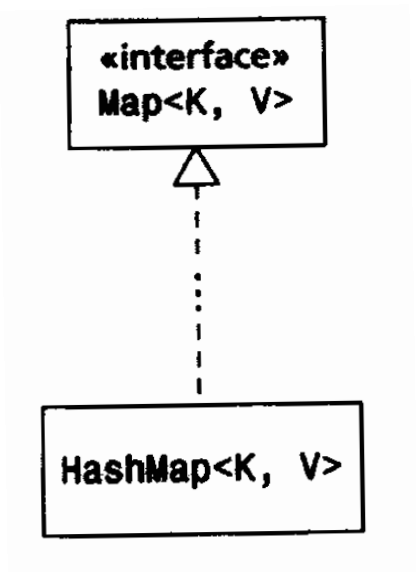
Maps: HashMap<K, V>


```
import java.util.HashMap;
import java.util.Map;


public class Hashing {                                          // (5)
  public static void main(String[]  args) {

    System.out.println("When equals() and hashCode() methods are" +
                       " not overridden:");
    Point3D_V1 p1 = new Point3D_V1(1, 2, 3);
    Point3D_V1 p2 = new Point3D_V1(1, 2, 3);
    System.out.println("Point3D_V1 p1" + p1 + ": " + p1.hashCode());
    System.out.println("Point3D_V1 p2" + p2 + ": " + p2.hashCode());
    System.out.println("p1.hashCode() == p2.hashCode(): " +
                       (p1.hashCode() == p2.hashCode()));
    System.out.println("p1.equals(p2): " + p1.equals(p2));

    Map<Point3D_V1,Integer> hashTab1 = new HashMap<Point3D_V1,Integer>()
        ;
    hashTab1.put(p1, 2);
    hashTab1.put(p2, 5);
    System.out.println("Hash table with Point3D_V1: " + hashTab1);
    Point3D_V1 p3 = new Point3D_V1(1, 2, 3);
    System.out.println("Value for " + p3 + ": " + hashTab1.get(p3));

    System.out.println();
    System.out.println("When equals() and hashCode() methods are" +
                       " overridden:");
    Point3D pp1 = new Point3D(1, 2, 3);
    Point3D pp2 = new Point3D(1, 2, 3);
    System.out.println("Point3D pp1" + pp1 + ": " + pp1.hashCode());
    System.out.println("Point3D pp1" + pp2 + ": " + pp2.hashCode());
    System.out.println("pp1.hashCode() == pp2.hashCode(): " +
                       (pp1.hashCode() == pp2.hashCode()));
    System.out.println("pp1.equals(pp2): " + pp1.equals(pp2));

    Map<Point3D,Integer> hashTab2 = new HashMap<Point3D,Integer>();
    hashTab2.put(pp1, 2);
    hashTab2.put(pp2, 5);
    System.out.println("Hash table with Point3D: " + hashTab2);
    Point3D pp3 = new Point3D(1, 2, 3);
    System.out.println("Value for " + pp3 + ": " + hashTab2.get(pp3));
  }
}
```

Program Output

```
When equals() and hashCode() methods are not overridden:
Point3D_V1 p1[1,2,3]: 899026693
Point3D_V1 p2[1,2,3]: 246688959
p1.hashCode() == p2.hashCode(): false
p1.equals(p2): false
Hash table with Point3D_V1: {[1,2,3]=2, [1,2,3]=5}
Value for [1,2,3]: null

When equals() and hashCode() methods are overridden:
Point3D pp1[1,2,3]: 328727
Point3D pp1[1,2,3]: 328727
pp1.hashCode() == pp2.hashCode(): true
pp1.equals(pp2): true
Hash table with Point3D: {[1,2,3]=5}
Value for [1,2,3]: 5
```

- The functionality of maps is specified by the interface Map<K,V> in the java.util package
- The class HashMap<K,V> is a concrete implementation of the Map<K,V> interface



- The interface Map<K,V> and the class HashMap<K,V> are generic types
  - The type parameter K denotes the type of the keys
  - The type parameter V denotes the type of the values
- We can create an empty map with strings as keys and integers as values:

```
Map<String, Integer> wordMap = new HashMap<String, Integer>();
```

Basic operations from Map<K,V> interface

Method	Description
`int size()`	Returns the number of entries in the map.
`boolean isEmpty()`	Determines whether the map is empty, i.e. has any entries.
`V put(K key, V value)`	Associates the `key` with the `value` and stores the entry in the map. If the `key` already has an entry from before, it returns the old value from this entry, otherwise it returns the reference value
`V remove(Object key)`	Tries to remove the entry of the `key` if the `key` is registered in the map. If successful, the method returns the value of the key, otherwise it returns the reference value `null`.
`boolean containsKey(Object key)`	Determines whether the specified `key` has an entry in the map.
`boolean containsValue(Object value)`	Determines whether the specified `value` is associated with one or more keys in the map.

- The interface Map<K, V> DOES NO offer an iterator
- To access all the entries, we must first create a view of the map
- A map view is a collection that is associated with the underlying map
- The method keySet() creates a view that consists of a set with all the keys in the map

```
Set<String> keyView = wordMap.keySet(); // [not, to, what, or, will, be]
```

Method	Description
`public Set<K> keySet()`	Returns the `Set` *view* of all the keys in the map.
`public Collection<V> values()`	Returns the `Collection` *view* of all the values in all the entries in the map, which may contain duplicates.

Collection<Integer> valueView = wordMap.values(); // [1, 2, 1, 1, 2, 4]

```
1  import java.util.Collection;
   import java.util.HashMap;
3  import java.util.HashSet;
   import java.util.Map;
5  import java.util.Set;

7  public class MapClient {

9    public static void main(String args[]) {
       // (1) Create an empty map for <String, Integer> entries,
11     //     representing a word and its frequency.
       Map<String, Integer> wordMap = new HashMap<String, Integer>();
13
       // (2) Read words from the command line.
15     for (int i = 0; i < args.length; i++) {
         // Look up if the word is already registered.
17       Integer numOfTimes = wordMap.get(args[i]);
         if (numOfTimes == null)
19         numOfTimes = 1;     // Not registered before, i.e. first time.
         else                  // Registered. Frequency is incremented.
21         numOfTimes++;
         wordMap.put(args[i], numOfTimes);
23     }

25     // (3) Print the word map.
       System.out.println("Whole word map: " + wordMap);
27
       // (4) Print total number of words read.
29     Collection<Integer> freqCollection = wordMap.values();
       int totalNumOfWords = 0;
31     for (int frequency : freqCollection) {
         totalNumOfWords += frequency;
33     }
       System.out.println("Total number of words read: " + totalNumOfWords)
           ;
35
       // (5) Print all distinct words.
37     Set<String> setOfAllWords = wordMap.keySet();
       System.out.println("All distinct words: " + setOfAllWords);
39     System.out.println("Number of distinct words: " + wordMap.size());

41     // (6) Print all duplicated words.
       Collection<String> setOfDuplicatedWords = new HashSet<String>();
43     for (String key : setOfAllWords) {
         int numOfTimes = wordMap.get(key);
45       if (numOfTimes != 1)
           setOfDuplicatedWords.add(key);
47     }
       System.out.println("All duplicated words: " + setOfDuplicatedWords);
49     System.out.println("Number of duplicated words: " +
                          setOfDuplicatedWords.size());
51   }
   }
```

java MapClient to be or not to be what will be will be

Program Output

```
Whole word map: {not=1, to=2, what=1, or=1, will=2, be=4}
Total number of words read: 11
All distinct words: [not, to, what, or, will, be]
Number of distinct words: 6
All duplicated words: [to, will, be]
Number of duplicated words: 3
```

```
static double sumPair(Pair<Number> pair) {
    return pair.getFirst().doubleValue() + pair.getSecond().doubleValue();
}
```

- Can we call this method with numerical pairs whose type is a subtype of Pair<Number> (e.g., Integer)?
- The code below results in a compile-time error:

  ```
  double sum = sumPair(new Pair<Integer>(100, 200));
  ```

- We need to specify a subtype

- We must rewrite the method sumPair()
- The wildcard **?** can be used to pass pairs with an element type that is a subtype of Number

```
static double sumPair(Pair<? extends Number> pair) {
    return pair.getFirst().doubleValue() + pair.getSecond().doubleValue();
}
```

```
Pair<Double> doublePair = new Pair<Double>(100.50, 200.50);
double newSum = sumPair(doublePair);              // (2) Ok
Pair<Number> numPair = doublePair;                // (3) Compile-time error!
Pair<? extends Number> newPair = doublePair;      // (4) Ok
```

Supertypes using <? super T>

```
Pair<Number> numPair = new Pair<Number>(100.0, 200);
Pair<Integer> iPair = new Pair<Integer>(100, 200);
Pair<? super Integer> supPair = numPair;        // (5) Ok
supPair = iPair;                                // (6) Ok
supPair = doublePair;                           // (7) Ok
```

Specifying any type using <?> alone

```
Pair<?> pairB = numPair;                          // (8) Ok
pairB = newPair;                                  // (9) Ok
pairB = supPair;                                  // (10) Ok
pairB = new Pair<?>(100.0, 200);                  // (11) Compile-time error!
pairB = new Pair<? extends Number>(100.0, 200); // (12) Compile-time error!
```

- A method can declare its own formal type parameters and use them in the method

```
public static <T> List<T> arrayToList(T[] array) {  // (1)
  List<T> list = new ArrayList<T>();  // Create an empty list.
  for (T element : array)             // Traverse the array.
    list.add(element);                // Copy current element to list.
  return list;                        // Return the list.
}
```

- The classes java.util.Collections and java.util.Arrays in the Java standard library provide a number of overloaded generic methods for sorting and searching

- In Section 15.9 in the textbook you find a description of each
  - binarySearch
  -
  - sort

```
import java.util.List;
import java.util.Collections;
import java.util.Arrays;

public class SortingAndSearchingLists {
  public static void main(String[] args) {
    Integer[] numArray = {8, 4, 2, 6, 1};
    // Create a list from numArray.
    List<Integer> numList = Arrays.asList(numArray);

    // Sort the list.
    System.out.println("Unsorted list: " + numList);
    Collections.sort(numList);
    System.out.println("Sorted list: " + numList);

    // Search in the list.
    int index = Collections.binarySearch(numList, 4);
    System.out.println("Key " + 4 + " : index " + index); // Key exists.
    System.out.println("Key " + 5 + " : index " + // Key does not exist.
                       Collections.binarySearch(numList, 5));
  }
}
```

Program Output

```
Unsorted list: [8, 4, 2, 6, 1]
Sorted list: [1, 2, 4, 6, 8]
Key 4 : index 2
Key 5 : index -4
```