Chapter 14

Using Dynamic Data Structures

Lecture slides for:

*Java Actually: A Comprehensive Primer in Programming*
Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen
http://www.ii.uib.no/~khalid/jac/

*Permission is hereby granted to use these lecture slides in conjunction with the book.*

*Modified: 16/2/19*

Overview

- Dynamic data structures
- Abstract data types (ADTs)
- Dynamic strings: `StringBuilder`
- Generic types
- Collections and the interfaces `Collection`, `List` and `Set`:
  - Dynamic arrays: `ArrayList`
  - Sets: `HashSet`
- Hashtables and the interface `Map`:
  - Hashtables: `HashMap`
- Subtyping with wildcard `?`
- Generic methods
Dynamic data structures

- A collection is a data structure for storing data, e.g., an array defines a collection that stores elements of the same type and has fixed length.
- Dynamic data structures can grow and shrink as data is inserted into and retrieved from the structure.
  - The contract specifies which operations can be performed on the structure.
  - Clients do not need to know how it is implemented.
- Two most common operations:
  - Insertion: insert data into the structure.
  - Retrieval: extract data from the structure.
- Choice of data structure is largely contingent on how expensive it is to perform the insertion and lookup.

Abstract data types: ADT = data structure + contract

- Realization of data abstractions results in ADTs, i.e., the design of a new type with the corresponding data representation and operations.
- In Java, classes are ADTs.

Dynamic strings: StringBuilder class

- The contents of a String object cannot be changed, i.e., the state can only be read.
- Java has a predefined class StringBuilder to handle sequences of characters that can be changed, and where the character sequence can dynamically grow and shrink.
- An object of class StringBuilder keeps track of:
  - size (how many characters it contains at any given time), and
  - capacity (how many characters can be inserted in it before it becomes full)
- If there is room for more characters, the capacity expands automatically.
- Choose the class StringBuilder instead of the String class if the sequence of characters is be changed frequently.
- Java has support for the declaration, creation and use of dynamic strings.

Declaration that creates a reference to a StringBuilder object:

```java
StringBuilder variableName;
StringBuilder buffer1;
```

leaves the creation of a reference that can store the reference value of a StringBuilder object:

Navn: buffer1
Type: ref(StringBuilder) null
Creating dynamic strings

- A StringBuilder object can be created by calling a StringBuilder constructor using the new operator.
- We can combine the declaration with the creation:

```java
StringBuilder buffer = new StringBuilder(argument list);
```

```java
// Create a StringBuilder object which has no characters and length 0,
// whose capacity is 16 characters:
StringBuilder nameBuffer = new StringBuilder();

// Create a StringBuilder object which has no characters and length 0,
// whose capacity is 10 characters:
StringBuilder addrBuffer = new StringBuilder(10);

// Create a StringBuilder object from a string literal,
// whose capacity is string length + 16, i.e. 19 characters:
StringBuilder colourBuffer = new StringBuilder("red");

// Create a StringBuilder object from a String object.
StringBuilder strBuffer = new StringBuilder(str);
```

Operations on string buffers

StringBuilder courseBuffer = new StringBuilder("Java is cool!");

Selectors for StringBuilder class:

- Each StringBuilder object has an instance method, `length()`, which returns the number of characters in the string buffer (size).
  - Method call `courseBuffer.length()` returns the number of characters in the string buffer, i.e. 13.

- Each StringBuilder object has an instance method, `capacity()`, which returns the number of characters that can be inserted into the string buffer before it expands to store more characters (capacity).
  - Method call `courseBuffer.capacity()` returns the number of characters that can be stored in the string buffer, i.e. 29.

- The method `charAt(int i)` returns the character given by index i the string buffer.
  - Method call `courseBuffer.charAt(2)` returns the character 'v' at index 2 in the string buffer.
  - Start index is always 0.
  - Illegal index value results in a StringIndexOutOfBoundsException.
**StringBuilder object**

```
StringBuilder buffer = new StringBuilder("bana");
```

The `StringBuilder` object has 4 characters inserted, and room for 16 more - and can expand if necessary.

---

**Operations on the string builders**

Modifiers for `StringBuilder` class:
- Character can be inserted *anywhere* in the string builder.
  - Insertion may cause the other characters to be moved to make room for the new character.
  - The size is automatically adjusted on insertion.
- The over-loaded method `append()` can be used to add primitive values, `String` objects, arrays of characters and text representation of other objects at the *end* of the string builder.

```
StringBuilder buffer = new StringBuilder("banana");
buffer.append("na");  // append a string to the end of the string builder: "banana"
buffer.append(42);   // append a number at the end of the string builder: "banana42"
```

- The over-loaded method `insert()` can be used to insert primitive values, `String` objects, arrays of characters and other objects at a given index in the string builder.

```
buffer.insert(6, "Rama");  // "bananaRama42"
buffer.insert(11, 'U');    // "bananaRama4U2"
buffer.setCharAt(6, 'm');  // "bananamama4U2"
```
Operations on the string builders (cont.)

- The class `StringBuilder` does not override the `equals()` method from the `Object` class.
- String builders must be converted to strings in order to compare them:

```java
boolean status = buffer1.toString().equals(buffer2.toString());
```

Generic types

- ADTs where we can replace the reference types, are called *generic types*.
- The first draft of a *pair* of values:

```java
// Legacy class
public class PairObj {
    private Object first;
    private Object second;
    PairObj () { }
    PairObj (Object first, Object second) {
        this.first = first;
        this.second = second;
    }
    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    public void setFirst(Object firstOne) { first = firstOne; }
    public void setSecond(Object secondOne) { second = secondOne; }
}
```
A client of the class PairObj:

```java
class PairObjClient {
    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.
        }
    }
}
```

The client must keep track of what is put into a PairObj.

Requires checking and type conversion on lookup.

Generic classes

A generic class which can be used to create pairs of objects where both objects has the same type:

```java
class Pair<T> {
    // (1)
    private T first;
    private T second;
    Pair() { }
    Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }
    public T getFirst() { return first; }
    public T getSecond() { return second; }
    public void setFirst(T firstOne) { first = firstOne; }
    public void setSecond(T secondOne) { second = secondOne; }
    public String toString() {
        return "(" + first.toString() + "," + second.toString() + ");"; // (2)
    }
}
```
A generic class specifies one or more *formal type parameters*, e.g. `<T>`.
- In the generic class `Pair<T>` we have used `T` in all locations where we used the type `Object` in the definition of the class `PairObj`.
- The type parameter is used as a reference type in the class body: as a field type, such as return type and as parameter types in methods.
- What actual type the type parameter `T` really represents is not known in the generic class `Pair<T>`.

Note that formal type parameters are not specified after the class name in a constructor.

Parameterized types

A generic class is *used* by specifying the *actual type parameters* that replaces the *formal type parameters* in the class definition at compile time.

E.g. `Pair<String>` will introduce a new reference type during compilation, that is, pairs that only allow `String` objects, where the formal type parameter `T` is replaced by the actual type parameter `String`.

The compiler checks that parameterized types are correctly used in the source code, so that no runtime errors can occur.

Actual type parameters are specified after the class name, just as formal type parameters are in a generic class definition.

Primitive data types can *not* be specified as actual type parameters.

The relationship between generic types (`Pair<T>`) and parameterized types (`Pair<String>`) is comparable to the relationship between the declaration and calling of a method.
public class ParameterizedTypes {

    public static void main(String[] args) {
        Pair<String> strPair = new Pair<>("Adam", "Eve");  // (1)
        // Pair<String> mixPair = new Pair<>("17. May", 1905);// (2) Error!
        Pair<Integer> intPair = new Pair<>(2005, 2010);   // (3)
        // strPair = intPair;                // (4) Compile-time error!
        Pair<String> tempPair = strPair;  // (5) OK

        strPair.setFirst("Ole");    // (6) OK. Only String accepted.
        // intPair.setSecond("Maria"); // (7) Compile-time error!
        String name = strPair.getSecond().toLowerCase();  // (8) "eve"
        System.out.println(name);
    }
}

• The client does not keep track of what is put into a Pair.
• No checking and type conversion by reference.

Generic Interfaces

• Example:

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

• A generic interface can be implemented by a generic (or non-generic) class:

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

• We can declare references of parameterized interfaces.

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

    – Pair<String> is a subtype of PairRelationship<String>.
• From the Java standard library:

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

• A class that will provide a natural order for its objects, can implement the `Comparable<T>` interface:

```java
class Widget implements Comparable<Widget> {
    public int compareTo(Widget widget) {
        // Implementation
    // ...
    }
}
```

• Note that we have parameterized `Comparable<T>` with `Widget`, since it is objects of the class `Widget` that the method `compareTo()` will compare.

---

**Generic types during compilation**

• The generic class `Pair<T>` is compiled and will be represented by the class `Pair`, ie only one class file (`Pair.class`) with Java byte code is created.

• Parameterized types are used by the compiler to verify that the objects that are created are used correctly in the program.

• The runtime environment is, however, unaware of the use of generic types, ie it uses the class `Pair`.

• Since there is only one class representing all the parameterizations of a generic class, and only one instance of a static member can exist in a class.
  
  – Static methods cannot refer to the formal type parameters of its generic type.

• The compiler gives an *unchecked warning* in cases where the use of a generic type without any type parameters can cause problems during execution.
Collections

- A collection is a data structure that can maintain references to objects.
  - For example, an array of references to objects is a collection.
- Java API defines several other types of collections in the java.util package.
- Central to the java.util package are a few important generic interfaces that collections implement.

```
  «interface»
Collection<E>

  «interface»
Set<E>

  «interface»
List<E>

HashSet<E>    ArrayList<E>
```

Interface Collection<E>

- Basic operations are the most frequently ones performed on collections: insertion, deletion and determine membership.

<table>
<thead>
<tr>
<th>Selected basic operations from the interface Collection &lt;E&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>int size()</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
</tr>
<tr>
<td>boolean contains(Object element)</td>
</tr>
<tr>
<td>boolean add(E element)</td>
</tr>
<tr>
<td>boolean remove(Object element)</td>
</tr>
<tr>
<td>Iterator&lt;E&gt; iterator()</td>
</tr>
</tbody>
</table>

- The elements of the collection must override the equals() method from the Object class.
• **Bulk operations** are performed on the entire collection.

<table>
<thead>
<tr>
<th>Selected bulk operations from the interface Collection &lt;E&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean containsAll(Collection&lt;?&gt; s)</td>
</tr>
<tr>
<td><strong>Subset</strong>: returns true if all elements in the collection s are in this collection.</td>
</tr>
<tr>
<td>boolean addAll(Collection&lt;? extends E&gt; s)</td>
</tr>
<tr>
<td><strong>Union</strong>: adds all items from the collection s to this collection, and returns true if this collection was modified.</td>
</tr>
<tr>
<td>boolean retainAll(Collection&lt;?&gt; s)</td>
</tr>
<tr>
<td><strong>Intersection</strong>: retains in this collection only those items that are also in the collection s, and returns true if this collection was modified.</td>
</tr>
<tr>
<td>boolean removeAll(Collection&lt;?&gt; s)</td>
</tr>
<tr>
<td><strong>Difference</strong>: deletes all items in this collection which are also in the collection s, and returns true if this collection was modified.</td>
</tr>
<tr>
<td>void clear()</td>
</tr>
<tr>
<td>Deletes all items from this collection.</td>
</tr>
</tbody>
</table>

– Note that the methods addAll(), retainAll() and removeAll() are destructive, that they can change the current collection.

---

**Set Theory**

- **A** = \{kiss, Sivle, madonna, aha, abba\}
- **B** = \{TLC, wham, madonna, abba\}

(a)

- **A** = \{kiss, TLC, Sivle, wham, aha, madonna, abba\}
  - **A.addAll(B)**: A = \{kiss, TLC, Sivle, wham, aha, madonna, abba\}
  - **A.retainAll(B)**: A = \{kiss, aha, abba\}
  - **A.removeAll(B)**: A = \{kiss, Sivle, aha\}

(b) Union

(c) Intersection

(d) Difference
Traversing over a collection

- All collections implement:
  ```java
  interface Iterable<E> {
    Iterator<E> iterator();
  }
  ```
- The `for(:)`-loop can be used to traverse a collection:
  ```java
  // Create a list of strings.
  Collection<String> collection = new ArrayList<>();
  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.
  }

  for (String str : collection)
    System.out.print(str);
  ```

Lists

Subinterface List

- A collection that allows duplicates, and where the items are ordered, is called a list.
- The subinterface List extends the Collection interface to apply for lists.
- The elements have a position (indicated by an index) in list, starting with index 0.

<table>
<thead>
<tr>
<th>Selected list of operations in the interface List</th>
</tr>
</thead>
<tbody>
<tr>
<td>E get(int index)</td>
</tr>
<tr>
<td>E set(int index, E element)</td>
</tr>
<tr>
<td>void add(int index, E element)</td>
</tr>
<tr>
<td>E remove(int index)</td>
</tr>
<tr>
<td>int indexOf(Object obj)</td>
</tr>
</tbody>
</table>
ArrayLists (ListClient.java)

- The program reads arguments from the command line into a list, and censors specific words from this list.
- We use an import statement at the beginning of the source file, so that we do not have to use the full package path to refer to classes from the java.util package.
- Words to be censored are given in a separate list (censoredWords).
- At (1) we create an empty list for strings:
  ```java
  List<String> wordList = new ArrayList<>();
  ```
- At (2) we create an empty list (censoredWords) which is populated with words to be censored.
  ```java
  List<String> censoredWords = new ArrayList<>();
  ```
- At (3) we use a for(:)-loop to traverse the word list to check every word in it.
  ```java
  for (String element : wordList) {
      if (censoredWords.indexOf(element) != -1) {
          int indexInWordList = wordList.indexOf(element);
          wordList.set(indexInWordList, CENSORED);
      }
  }
  ```

Sets: HashSet<E>

Subinterface Set<E>

- The interface Set models the mathematical concept of sets that allows operations such as union, intersection and difference from the set theory.
- A set does not allow duplicate values.
- The Set interface does not introduce any new methods beyond those that already exist in the Collection interface, but specializes them for sets.
Example: Sets (SetClient.java)

- Example: Two sets with artist names.
- An empty set for concert A is created by instantiating class HashSet:
  ```java
  Set<String> concertA = new HashSet<>();
  ```
- We add the different artists using the `add()` method:
  ```java
  concertA.add("aha"); concertA.add("madonna");
  ```
  - If the `add()` method resulted in an element being added to the set, the method returns the value `true`.
  - Truth value `false` indicates that the element already exists in the set.
- The code below shows how we can make a copy of a set by calling the appropriate constructor in class `HashSet<E>:
  ```java
  Set<String> allArtists = new HashSet<>(concertA);
  ```

Example: Create a list without duplicates (Duplicates.java)

- The code below will create a set (`wordSet`) of `String` objects from a list (`wordList`) of `String` objects, and thus remove the duplicates:

```java
import java.util.*;

public class Duplicates {
    public static void main(String args[]) {
        ArrayList<String> wordList = new ArrayList<>(); // Original list
        wordList.add("two"); wordList.add("zero");
        wordList.add("zero"); wordList.add("five");
        System.out.println("Original word list: "+ wordList);

        Set<String> wordSet = new HashSet<>(wordList);// New set
        wordList = new ArrayList<>(wordSet); // List without duplicates.
        System.out.println("Original list, without duplicates: "+ wordList);
    }
}
```

Output from the program:
Original word list: [two, zero, zero, five]
Original list, without duplicates: [two, five, zero]
Hash tables (Maps)

- A hash table is used to store entries.
- An entry is a pair of objects, where one object (called the key) is associated with the second object (called the value).
- Example: A telephone list is a hash table, where each entry associates a telephone number (key) with a name (value).
- It is a many-to-one relationship between keys and values in a hash table:
  - Different keys may have the same value but different values may not have the same key, i.e., keys are unique in a hash table.

The interface Map<K, V>

- The functionality of the hash tables is specified in the Map interface in the java.util package.
- The HashMap<K, V> class is a concrete implementation of the Map interface.
- We can create an empty hash table for entries <String, Integer> as shown below:

```java
Map<String, Integer> wordFrequency = new HashMap<>();
```
### Basic Operations in the interface `Map<K, V>`

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int size()</code></td>
<td>Returns the number of entries in the hash table.</td>
</tr>
<tr>
<td><code>boolean isEmpty()</code></td>
<td>Returns true if the hash table has no entries.</td>
</tr>
<tr>
<td><code>V put(K key, V value)</code></td>
<td>Binds key to value and stores the entry. If the key had a previous entry, returns the value from the previous entry.</td>
</tr>
<tr>
<td><code>V get(Object key)</code></td>
<td>Returns the value of the entry that is key, if the key was registered before. Otherwise, returns the reference value <code>null</code>.</td>
</tr>
<tr>
<td><code>V remove(Object key)</code></td>
<td>Tries to delete the entry of the key and returns the value of this entry if the key was registered before.</td>
</tr>
<tr>
<td><code>boolean containsKey(Object key)</code></td>
<td>Returns true if the key has an entry.</td>
</tr>
<tr>
<td><code>boolean containsValue(Object value)</code></td>
<td>Returns true if the value is included in at least one entry.</td>
</tr>
</tbody>
</table>

### Hashing

- Storage and lookup of entries in a hash table requires that it is possible to identify a key in an entry using an integer, called the **hash code**.
- Java uses the method `hashCode()` to calculate a hash code for an object, as defined in the class `Object`.
- The hash code must meet the following conditions:
  - The hash code will always be the same for an object, as long as the state of the object is not modified.
  - Two objects that are identical according to the `equals()` method must also have the same hash code.
- Class `String` and wrapper classes override both `hashCode()` and `equals()` methods from the `Object` class.

*Main rule: If a class overrides `equals()`, it should also override `hashCode()`.*
Overriding the `hashCode()` method

- The class `Point3D_V1` overrides neither the `equals()` method or the `hashCode()`-method:

```
class Point3D_V1 {                                              // (1)
    int x;
    int y;
    int z;

    Point3D_V1(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public String toString() { return "[" + x + "," + y + "," + z + "]"; }
}
```

- The class `Point3D` overrides the `equals()` method and the `hashCode()`-method:

```
class Point3D {                                                 // (2)
    private int x;
    private int y;
    private int z;
    // ...
    /** Two points are equal if they have the same x, y and z coordinates. */
    public boolean equals(Object obj) {                           // (3)
        if (this == obj) return true;
        if (!(obj instanceof Point3D)) return false;
        Point3D p2 = (Point3D) obj;
        return this.x == p2.x && this.y == p2.y && this.z == p2.z;
    }

    /** The hash value is computed based on the coordinate values. */
    public int hashCode() {                                       // (4)
        int hashValue = 11;
        hashValue = 31 * hashValue + x;
        hashValue = 31 * hashValue + y;
        hashValue = 31 * hashValue + z;
        return hashValue;
    }
}
```
The class Hashing shows what happens when we use point objects:

When equals() and hashCode() methods are not overridden:

```
Point3D_V1 p1[1,2,3]: 1671711
Point3D_V1 p2[1,2,3]: 11394033
p1.hashCode() == p2.hashCode(): false
p1.equals(p2): false
Hash table with Point3D_V1: {[1,2,3]=2, [1,2,3]=5}  <= Keys are not unique.
Value for [1,2,3]: null                             <= Cannot find the key.
```

When equals() and hashCode() methods are overridden:

```
Point3D pp1[1,2,3]: 328727
Point3D pp2[1,2,3]: 328727
pp1.hashCode() == pp2.hashCode(): true
pp1.equals(pp2): true
Hash table with Point3D: {[1,2,3]=5}               <= Keys are unique.
Value for [1,2,3]: 5                               <= Can find the key.
```

Example (MapClient.java): Basic operations in the interface Map<K,V>

<table>
<thead>
<tr>
<th>Word (key)</th>
<th>Frequency (value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>to</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>4</td>
</tr>
<tr>
<td>or</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- The method put() creates an entry and stores it in the hash table:
  ```
  wordMap.put("to", 2); // "to", new Integer(3)
  ```
  - This method will overwrite any previous entry with the same key and will return the value from the previous entry.
- Lookup the value of a key:
  ```
  int frequency = wordMap.get("to"); // 2
  ```
- Entry of a key can be deleted with the remove() method that returns the value of the entry.
- Determine whether a value occurs in one or more entries:
  ```
  boolean keyFound = wordMap.containsKey("to"); // true
  boolean valueFound = wordMap.containsValue(2001); // false
  ```
• Standard text representation of a hash table:
  \{to=2, be=4, or=1, ...\}

Map views

• A map view is a collection that is associated with an underlying hash table.
• By means of such a view we can, for example, traverse the underlying hash table.
• The changes can be made through a view, and are reflected in the underlying hash table.

<table>
<thead>
<tr>
<th>Selected view operations from the interface Map</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public Set&lt;K&gt; keySet()</code></td>
</tr>
<tr>
<td><code>(Key View) Returns the Set-view of all the keys in the hash table.</code></td>
</tr>
<tr>
<td><code>public Collection&lt;V&gt; values()</code></td>
</tr>
<tr>
<td><code>(Value View) Returns the Collection-view of all the values of all the entries in the hash table.</code></td>
</tr>
</tbody>
</table>
• **Key view:**
  
  ```java
  Set<String> setOfAllWords = wordMap.keySet(); // [be, or, to, ...]
  ```
  
  - Since the keys are unique, it is appropriate to create a set which does not allow duplicates.
  - The `for(:)`-loop can be used to iterate over the keys in this set in the usual way.
  - The method `get()` can be used on the hash table to retrieve the corresponding value of the key.

• **Value view:**

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

  - Since values are not unique, it is appropriate to create a collection that allows duplicates.
  - The `for(:)`-loop can be used to iterate over the values in this collection in the usual way.

---

**Using Maps (MapClient.java)**

• (2) Read words from the command line:

  Repeat while arguments in command line:
  
  - Lookup in the hash table with the current argument.
  - If the current argument is not registered:
    - Let the frequency of the current argument be 0, i.e. first time;
    - Increase the frequency of the current argument and insert the entry.

  ```java
  for (int i = 0; i < args.length; i++) {
    // Look up if the word is already registered.
    Integer numOfTimes = wordMap.get(args[i]);
    if (numOfTimes == null) {
      numOfTimes = 0;    // Not registered before, initialize to 0.
    }
    wordMap.put(args[i], ++numOfTimes);
  }
  ```
• (4) Determine total number of words read.
  – Total number of words read is the sum of the frequencies.
  – We create a value view, which is used to iterate over this collection to sum all the frequencies.

    Collection<Integer> freqCollection = wordMap.values();
    int totalNumOfWords = 0;
    for (int frequency : freqCollection) {
        totalNumOfWords += frequency;
    }

• (5) Determine all distinct words:
  – All distinct words are the keys that are registered.
  – Create a key view that is a set of all words (i.e., keys) from the hash table.

    Set<String> setOfAllWords = wordMap.keySet();

• (6) Determine all duplicated words:

    Create an empty set for duplicate words.
    Create a key view that is a set of all keys.
    Repeat while there are elements in the set of all keys:
      Lookup in the hash table with the current key.
      If the frequency is not equal to 1, i.e. a duplicated word:
        Insert the word in the set of duplicate words.

    Collection<String> setOfDuplicatedWords = new HashSet<>();
    for (String key : setOfAllWords) {
        int numOfTimes = wordMap.get(key);
        if (numOfTimes != 1) {
            setOfDuplicatedWords.add(key);
        }
    }
Subtyping with wildcard (?)

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

double sum = sumPair(new Pair<Integer>(100, 200));  // (1) Error!

- `Pair<Integer>` is not a subtype of `Pair<Number>`, whereas the array type `Integer[]` is a subtype of array type `Number[]`.

- The parametrized type `Pair<? extends T>`:

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

- The parametrized type `Pair<? extends Number>` stands for all pairs whose element type is either `Number` or a subtype of `Number`, in other words, `Pair<? extends Number>` is a supertype of `Pair<Number>`, `Pair<Double>` and `Pair<Integer>`.

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

- The parametrized type `Pair<? super Integer>`:

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

- The parametrized type `Pair<? super Integer>` stands for all pairs whose element type is either `Integer` or a supertype of `Integer`, i.e. `Pair<? super Number>` is the supertype of `Pair<Number>`, `Pair<Comparable>` and `Pair<Integer>`.

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

- The parametrized type `Pair<?>`:

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

- `Pair<?>` represents the type of all pairs.

```java
Pair<?> pairB = numPair;  // (8) Ok
pairB = newPair;  // (9) Ok
pairB = supPair;  // (10) Ok
pairB = new Pair<?>((100.0, 200));  // (11) Error!
pairB = new Pair<? extends Number>((100.0, 200));  // (12) Error!
```
Generic methods

- A method can declare its own formal type parameters and use them in the method (see GenericMethods.java).

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

- In a generic method, a formal type parameter T is only available in the method.
- It is not a requirement that a generic method must be declared in a generic class.

- We can call a generic method in the usual way.
  - The compiler determines the actual type parameter from the method call:

```java
String[] strArray = {":-(", ";-)", ":-)"};
List<String> strList = arrayToList(strArray); // T is String.
System.out.println(strList); // [:

Integer[] intArray = {2007, 7, 2};
List<Integer> intList = arrayToList(intArray); // T is Integer.
System.out.println(intList); // [2007, 7, 2]
```
More Examples of Generic Methods

- The method `insertionSort()` sorts a list according to the specified `Comparator` object (see the file `GenericUtilForLists.java`):

  ```java
  static <E> void insertionSort(List<E> list, Comparator<E> comp) { ... }
  ```

- The method `binarySearch()` finds the index of a key in an array, if the key exists in the array (see file `GenericUtilForArrays.java`):

  ```java
  static <T extends Comparable<T>> int binarySearch(T[] array, T key) { ... }
  ```

Selected generic sorting and search methods in Java API

<table>
<thead>
<tr>
<th>java.util.Collections</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static &lt;T&gt; int binarySearch(List&lt;? extends Comparable &lt;? super T&gt;&gt; list, T key)</code></td>
</tr>
<tr>
<td><code>static &lt;T&gt; int binarySearch(List&lt;? extends T&gt; list, T key, Comparator&lt;? super T&gt; comp)</code></td>
</tr>
<tr>
<td><code>static &lt;T&gt; void sort(List&lt;T&gt; list)</code></td>
</tr>
<tr>
<td><code>static &lt;T&gt; void sort(List&lt;T&gt; list, Comparator&lt;? super T&gt; comp)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>java.util.Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static &lt;T&gt; int binarySearch(T[] array, T key, Comparator&lt;? super T&gt; comp)</code></td>
</tr>
<tr>
<td><code>static &lt;T&gt; void sort(T[] array, Comparator&lt;? super T&gt; comp)</code></td>
</tr>
<tr>
<td><code>static &lt;T&gt; List&lt;T&gt; asList(T... argumenter)</code></td>
</tr>
</tbody>
</table>