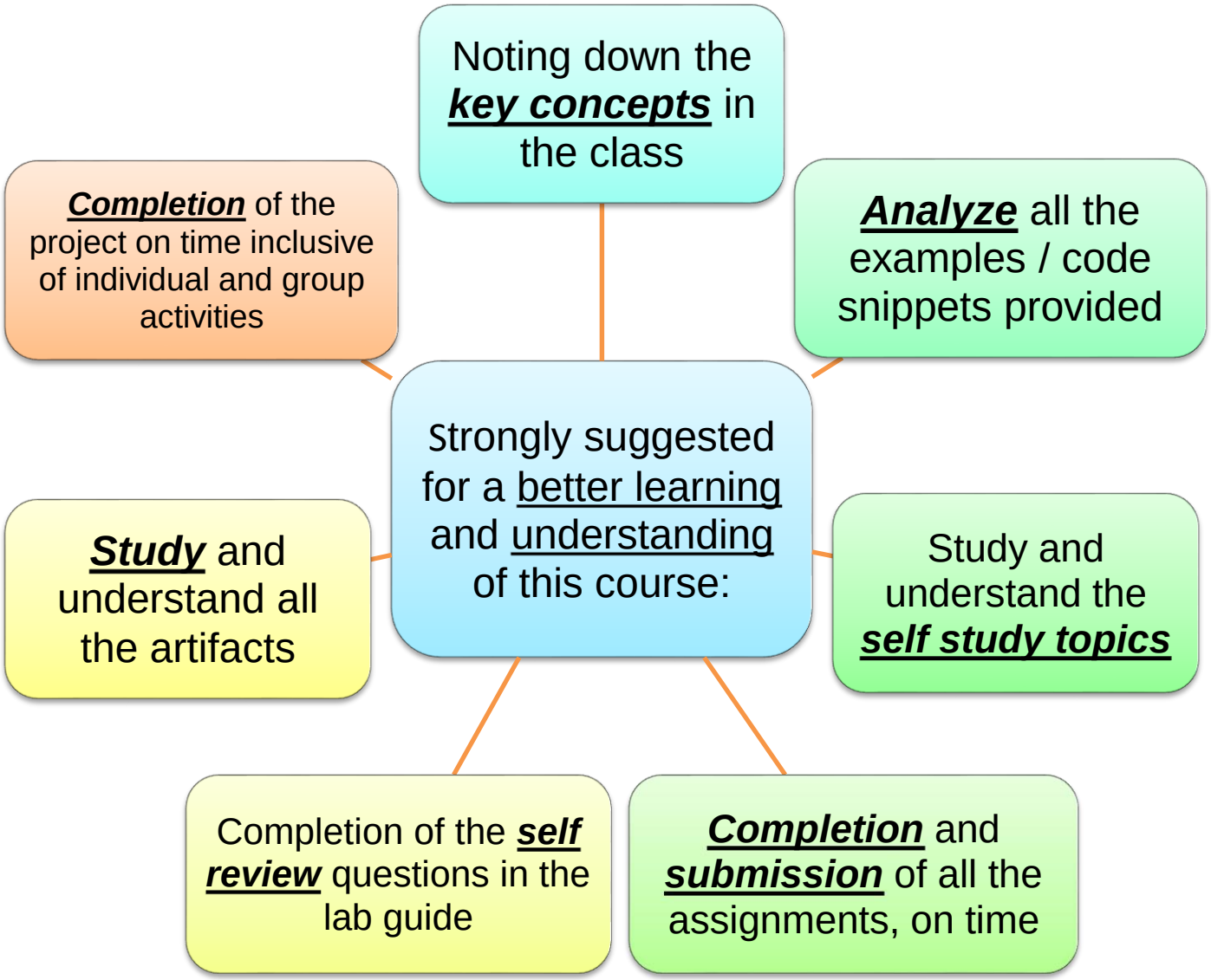
STRING COLLECTIONS

Instructor:

Table of contents

- Java String
- Generic collection

Learning Approach



Section 3

STRING CLASS

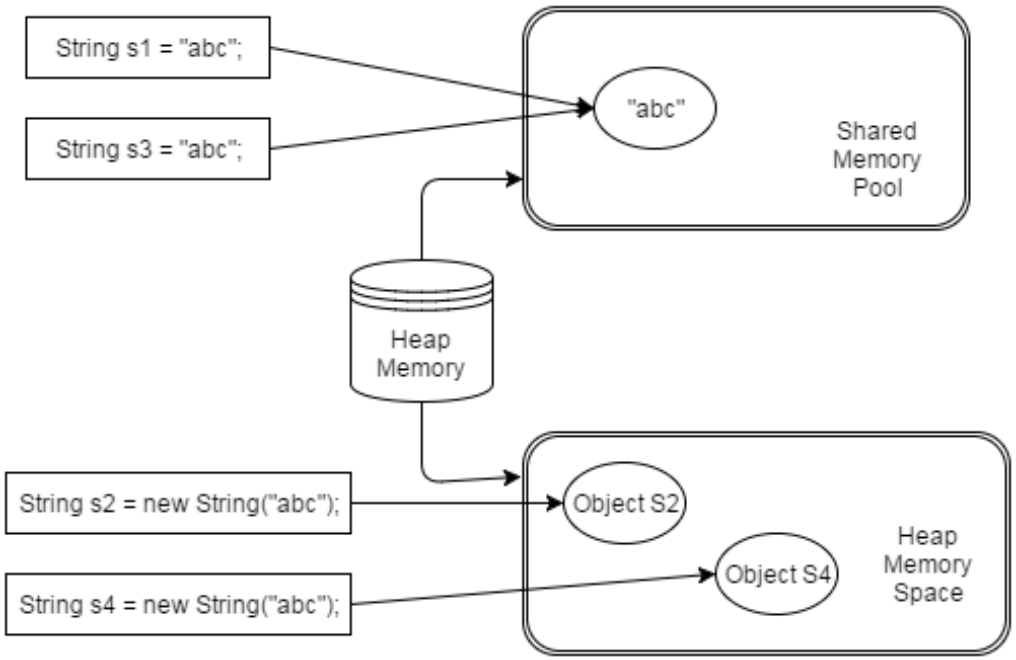
String class

- **String** is a sequence of characters, for e.g. “Hello” is a string of 5 characters.
- In java, string is an **immutable object** which means it is **constant** and can **cannot be changed** once it has been created.
- **Creating a String**
  - ✓ There are two ways to create a String in Java
    - String **literal**
    - Using **new** keyword

String class

- **String literal**
  - ✓ In java, Strings can be created like this: Assigning a String literal to a String instance.
  - ✓ Example:

```
String s1= “abc";
String s3= “abc";
```



String class

- **Using New Keyword**
  - ✓ The compiler would **create two different object** in memory having the **same string**.
  - ✓ **Example:**

```
public class StringSample {

  public static void main(String[] args) {
    // creating a string by java string literal
    String s1 = "FPT";
    String s3 = "FPT";

    char arrch[] = { 'h', 'e', 'l', 'l', 'o' };
    // converting char array arrch[] to string str2
    String s2= new String(arrch);

    // creating another java string str3 by using new keyword
    String s4 = new String("hello");

    // Displaying all the three strings
    System.out.println(s1.equals(s3));
    System.out.println(s2.equals(s4));

    System.out.println(s1 == s3);
    System.out.println(s2 == s4);
  }
}
```

Output:

```
true
true
true
false
```

String class

- **Methods:**
  - ✓ **char charAt(int index)**: It returns the character at the specified index. Specified index value should be between 0 to length() -1 both inclusive. It throws IndexOutOfBoundsException if index<0||>= length of String.
  - ✓ **boolean equals(Object obj)**: Compares the string with the specified string and returns true if both matches else false.
  - ✓ **int compareTo(String string)**: This method compares the two strings based on the Unicode value of each character in the strings.
  - ✓ **boolean startsWith(String prefix)**: It tests whether the string is having specified prefix, if yes then it returns true else false.
  - ✓ **boolean endsWith(String suffix)**: Checks whether the string ends with the specified suffix.
  - ✓ **int hashCode()**: It returns the hash code of the string.
  - ✓ **int indexOf(int ch)**: Returns the index of first occurrence of the specified character ch in the string.
  - ✓ **boolean contains(CharSequence s)**: It checks whether the string contains the specified sequence of char values. If yes then it returns true else false. It throws NullPointerException of ‘s’ is null.

String class

- **Methods:**
  - ✓ **String concat(String str)**: Concatenates the specified string “str” at the end of the string.
  - ✓ **String trim()**: Returns the substring after omitting leading and trailing white spaces from the original string.
  - ✓ **byte[] getBytes()**: This method is similar to the above method it just uses the default charset encoding for converting the string into sequence of bytes.
  - ✓ **int length()**: It returns the length of a String.
  - ✓ **boolean matches(String regex)**: It checks whether the String is matching with the specified **regular expression** regex.
  - ✓ **static String valueOf()**: This method returns a string representation of passed arguments such as int, long, float, double, char and char array.
  - ✓ **char[] toCharArray()**: Converts the string to a character array.
  - ✓ **String[] split(String regex)**: Same as split(String regex, int limit) method however it does not have any threshold limit.

String/Number casting

- **Convert a digit sequence to number**

```
// Each class in right hand side is called wrapper
// class of the corresponding primitive type
byte  b = Byte.parseByte("128");
                              // NumberFormatException
short s = Short.parseShort("32767");
int   x = Integer.parseInt("2");
int   y = Integer.parseInt("2.5");
                              // NumberFormatException
int   z = Integer.parseInt("a");
                              // NumberFormatException
long  l = Long.parseLong("15");
float f = Float.parseFloat("1.1");
double d = Double.parseDouble("2.5");
```

String builder/String buffer

- The **StringBuffer** and **StringBuilder** classes: to make a lot of modifications to Strings of characters.
- The StringBuilder class was introduced as of Java 5 and the main difference between the StringBuffer and StringBuilder is that StringBuilders methods are not thread safe(not Synchronised).
- It is recommended to use **StringBuilder** whenever possible because it is faster than StringBuffer. However if thread safety is necessary the best option is StringBuffer objects.

String/String builder/String buffe

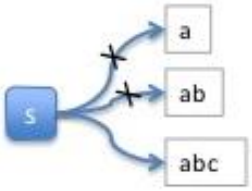
- String is **immutable**, if you try to alter their values, another object gets created,
- StringBuffer and StringBuilder are **mutable** so they can change their values.

String

Immutable

Every time you alter String values, it will allocate another exact amount of space in the heap. The previous value in the memory will be garbage-collected later.

```
String s = "a";
s += "b";
s += "c";
```

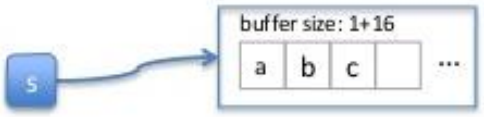


StringBuffer

Mutable

When created it reserves a certain amount of space in the heap, which can be larger than the value. Within that space, values can be modified without additional memory use. When the value requires more space, the space will automatically grow larger.

```
StringBuffer s = new StringBuffer("a");
s.append("b");
s.append("c");
```



String builder/String buffer

```
StringBuilder sb = new StringBuilder("abc");
```

- ✓ `sb.append(" def");` // "abc def"
- ✓ `char letter = str.charAt(2);` // "b"
- ✓ `char ch[] = new char[3];`
  `str.getChars(1,3,ch,0);` // Bây giờ biến "ch" chứa "abc"
- ✓ `sb.delete(3, 5);` // "abcef"
- ✓ `sb.deleteCharAt(4);` // "abce"
- ✓ `sb.insert(3, " d");` // "abc de"
- ✓ `sb.replace(2, 4, " ghi");` // "ab ghide"
- ✓ `sb.reverse();` // "edihg ba"
- ✓ `sb.setCharAt(5, 'j');` // "edihgjba"

Using StringTokenizer Class

- **`StringTokenizer`** can be used to parse a line into words
  - ✓ import `java.util.*`
  - ✓ some of its useful methods are shown in the text
    - e.g. test if there are more tokens
  - ✓ you can specify *delimiters* (the character or characters that separate words)
    - the default delimiters are "white space" (space, tab, and newline)

Example: StringTokenizer

- Display the words separated by any of the following characters: space, new line (\n), period (.) or comma (,).

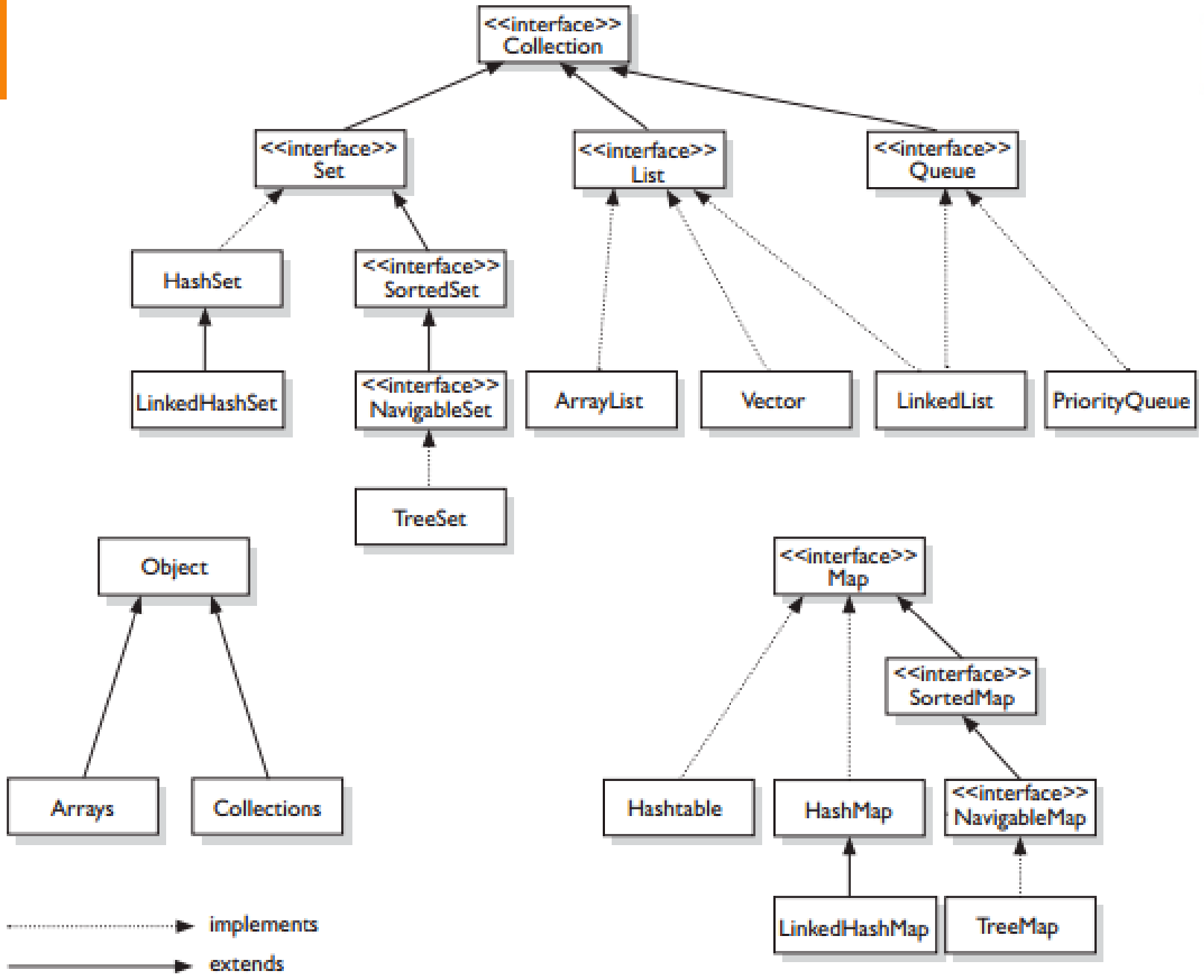
```
String inputLine = keyboard.nextLine();
StringTokenizer wordFinder =
        new StringTokenizer(inputLine, " \n.,");
//the second argument is a string of the 4delimiters
while(wordFinder.hasMoreTokens())
{
   System.out.println(wordFinder.nextToken());
}
```

Entering "Question,2b.or !tooBee." gives this output:

```
Question
2b
or
!tooBee
```

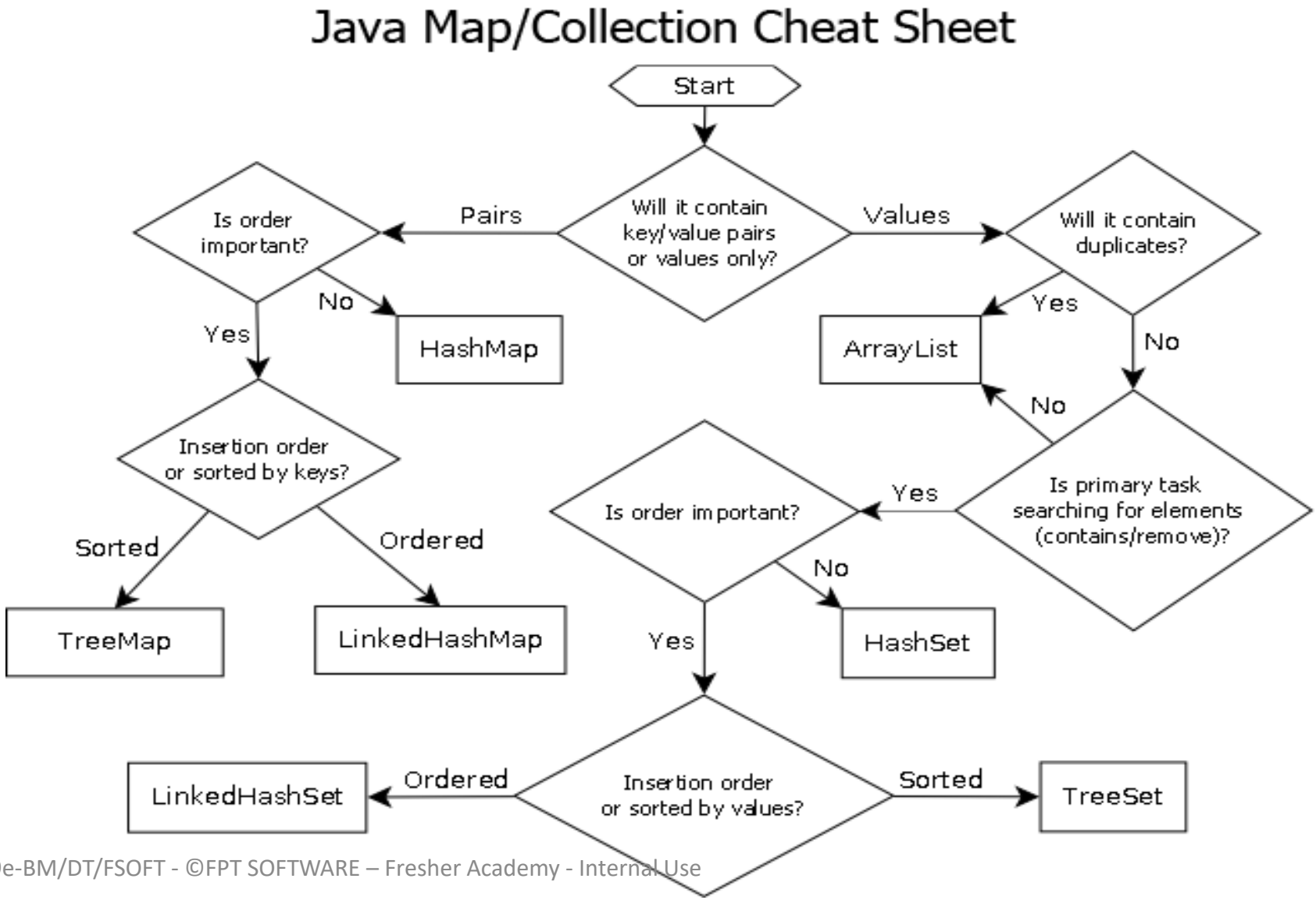
Section 4

GENERIC COLLECTION


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
HashSet
<<interface>>
SortedSet
LinkedHashSet
<<interface>>
NavigableSet
ArrayList
Vector
LinkedList
PriorityQueue
TreeSet
Object
Arrays
Collections
<<interface>>
Map
<<interface>>
SortedMap
Hashtable
HashMap
<<interface>>
NavigableMap
LinkedHashMap
TreeMap
implements
extends

Generic collection

- Collection Interface

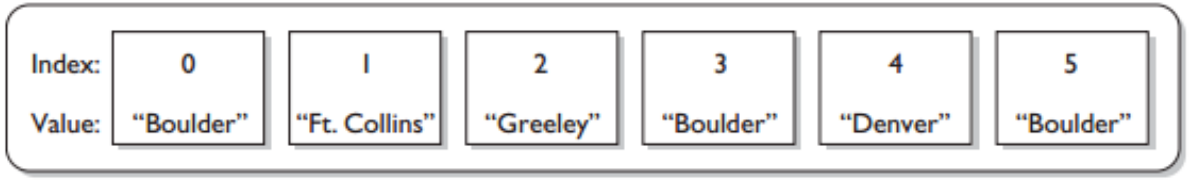


Generic collection

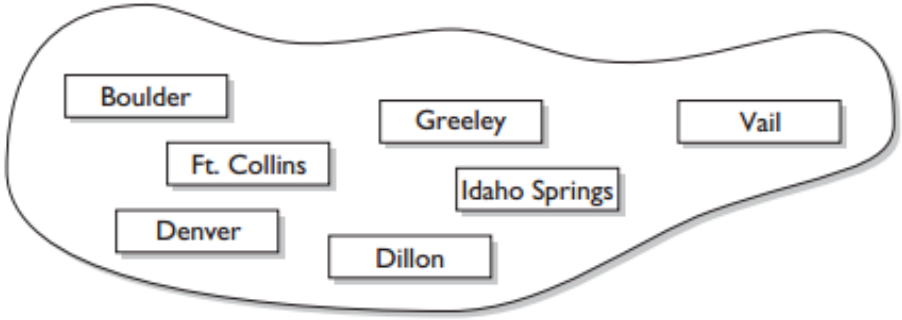
Figure 7-3 illustrates the structure of a List, a Set, and a Map.

FIGURE 7-3

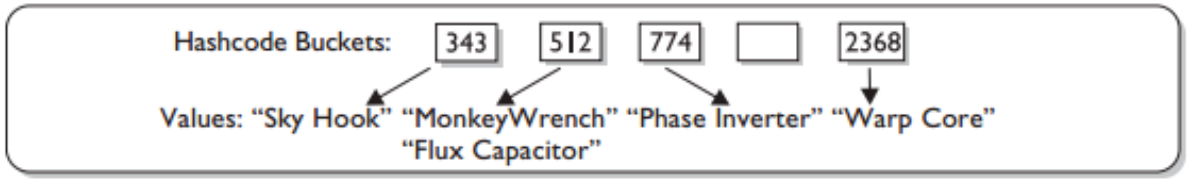
The structure of a List, a Set, and a Map



List: The salesman's itinerary (Duplicates allowed)



Set: The salesman's territory (No duplicates allowed)

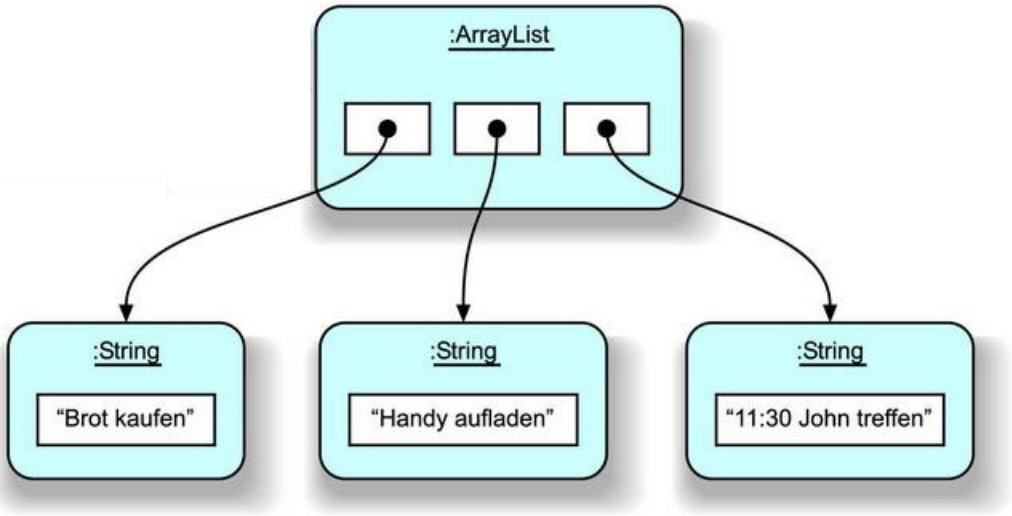


HashMap: the salesman's products (Keys generated from product IDs)

ArrayList

- **ArrayList** supports dynamic arrays that can grow as needed.
  - ✓ Array lists are created with an initial size.
  - ✓ When this size is exceeded, the collection is automatically enlarged.
  - ✓ When objects are removed, the array may be shrunk
- **Syntax:**

  List<DataType> arrName = new **ArrayList<>**();



ArrayList

- **List** is the interface which allows to store objects in a resizable container.
- **ArrayList** is implemented as a resizable array. If more elements are added to ArrayList than its initial size, its size is increased dynamically. The elements in an ArrayList can be accessed directly and efficiently by using the get() and set() methods, since ArrayList is implemented based on an array.
- **LinkedList** is implemented as a double linked list. Its performance on add() and remove() is better than the performance of Arraylist. The get() and get() methods have worse performance than the ArrayList, as theLinkedList does not provide direct access.

	ArrayList	LinkedList
get()	O(1)	O(n)
add()	O(1)	O(1) amortized
remove()	O(n)	O(n)

ArrayList: Input

```
public class ListExample {
public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    list.add(3); list.add(2);
    list.add(1); list.add(4);
    list.add(5); list.add(6);
    list.add(6);
    for (Integer integer : list) {
        System.out.println(integer);
    }
}
}
```

ArrayList: Input

```
class A{int i;}
A[] arA = new A[10];                    // Predefined capacity required
…
ArrayList<A> alA = new ArrayList<A>(); // No predefined capacity
boolean b = alA.isEmpty();              // true

A aA = new A(); aA.i = 1;
alA.add(aA);                            // add new
b = alA.isEmpty();                      // false
alA.add(aA);                  // add new again, duplicate accepted

A aoA = new A(); aoA.i = 2;
alA.add(1, aoA);           // insert to the 2nd position, (1, 2, 1)
```

ArrayList: Output

```
int s = alA.size();  // 3

A outA = alA.get(2);
b = outA == aoA;     // true

outA = alA.get(3);   // error, out of range

alA.set(2, aoA);     // replace the 3rd position, (1, 2, 2)

int i = alA.indexOf (aoA); // 1
i = alA.lastIndexOf (aoA); // 2

for (A a: alA){System.out.println(a.i);}          // 1, 2, 2

alA.remove(1);       // remove the 2nd position, (1, 2)
```

ArrayList Example

- Can use ArrayList to store String, Number:

ArrayListOfString.java

```
13 public class ArrayListOfString {
14     public static void main(String[] args) {
15
16         ArrayList<String> listOfNames = new ArrayList<String>();
17
18         listOfNames.add("Wartian Herkku");
19         listOfNames.add("Wellington Importadora");
20         listOfNames.add("White Clover Markets");
21         listOfNames.add("Wilman Kala");
22         listOfNames.add("Wolski");
23
24         System.out.println("Value at the first index: " + listOfNames.get(0));
25
26         // Use for loop to look up arraylist
27         System.out.println("Before array :");
28         int arrSize = listOfNames.size();
29         for (int i = 0; i < arrSize; i++) {
30             System.out.println(i + ": " + listOfNames.get(i));
31         }
32
33         listOfNames.remove("Wilman Kala");
34         listOfNames.remove(1);
35         listOfNames.add(2, "Matti Karttunen");
36
37         System.out.println("After array (Unsorted List):");
38
39         arrSize = listOfNames.size();
40         for (int i = 0; i < arrSize; i++) {
41             System.out.println(i + ": " + listOfNames.get(i));
42         }
```

Instance of ArrayList

Add value into ArrayList

Get value from ArrayList

Using for loop to lookup value

Remove by Value

Remove by Index

Add value by Index

ArrayList Example

- Can use ArrayList to store String, Number:

```
43          /* Sort statement */
44          Collections.sort(listOfNames);
45
46          System.out.println("After Sorting:");
47
48          arrSize = listOfNames.size();
49          for (int i = 0; i < arrSize; i++) {
50              System.out.println(i + ": " + listOfNames.get(i));
51          }
52      }
53  }
```

Sort statement

```
Value at the first index: Wartian Herkku
Before array :
0: Wartian Herkku
1: Wellington Importadora
2: White Clover Markets
3: Wilman Kala
4: Wolski
After array (Unsorted List):
0: Wartian Herkku
1: White Clover Markets
2: Matti Karttunen
3: Wolski
After Sorting:
0: Matti Karttunen
1: Wartian Herkku
2: White Clover Markets
3: Wolski
```

ArrayList with Object

- Create an **Animal** class:

```
public class Animal {
  private String name;
  private float weight;

  public Animal() {
  }

  public Animal(String name, float weight) {
    super();
    this.name = name;
    this.weight = weight;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public float getWeight() {
    return weight;
  }

  public void setWeight(float weight) {
    this.weight = weight;
  };

}
```

ArrayList with Object

- Can use ArrayList to store Objects:

ArrayListOfObject.java

```
14 public class ArrayListOfObject {
15
16     public static void main(String[] args) {
17
18         ArrayList<Animal> listOfAnimal = new ArrayList<Animal>();
19
20         listOfAnimal.add(new Animal("Cat", 2.0f));
21         listOfAnimal.add(new Animal("Dog", 8.0f));
22         listOfAnimal.add(new Animal("Turtle", 1.2f));
23         listOfAnimal.add(new Animal("Bear", 60.0f));
24         listOfAnimal.add(new Animal("Rabbit", 1.6f));
25         listOfAnimal.add(new Animal("Bird", 0.6f));
26
27         // Using for loop to lookup listOfAnimal
28         int arrSize = listOfAnimal.size();
29         for (int i = 0; i < arrSize; i++) {
30             System.out.println(listOfAnimal.get(i).getName() + "\t"
31                     + listOfAnimal.get(i).getWeight());
32         }
33
34         listOfAnimal.remove(3);
35     }
36 }
```

Instance of ArrayList

Add Animal to ArrayList

Use for loop to get

Remove by Index

ArrayList: Sort by Arrays

- **Comparable** and **Comparator** both are interfaces and can be used to sort collection elements.
- But there are many differences between Comparable and Comparator interfaces that are given below.
- **See:** fpt.clc.btjb.unit04.collections

Comparable	Comparator
1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc.	Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc.
2) Comparable **affects the original class** i.e. actual class is modified.	Comparator **doesn't affect the original class** i.e. actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is found in **java.lang** package.	Comparator is found in **java.util** package.
5) We can sort the list elements of Comparable type by**Collections.sort(List)** method.	We can sort the list elements of Comparator type by**Collections.sort(List,Comparator)** method.

ArrayList: Sort by Arrays

```
class A implements Comparable<A>{     // implement Comparable<T>
    int i;
    public int compareTo(A another){  // implement compareTo(T t)
        if (i == another.i) return 0;
        if (i < another.i) return -1;
        return 1;
    }
}

Object[] arA = alA.toArray();         // convert to array
Arrays.sort(arA);                     // using Arrays.sort
for (Object a: arA){
    A a1 = (A)a;                      // revert to original type
    System.out.println(a1.i);
}
```

Map and HashMap

- The **Map** interface defines an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.
- The **HashMap** class is an efficient implementation of the Map interface. The following code demonstrates its usage.
- Example:

Map and HashMap

```
public class MapTester {
    public static void main(String[] args) {
        // keys are Strings and objects are also Strings
        Map<String, String> map = new HashMap<>();
        map.put("Android", "Mobile");      map.put("Eclipse IDE", "Java");
        map.put("Eclipse RCP", "Java"); map.put("Git", "Version control system");
        // write to command line
        Set<String> keys = map.keySet();
        for (String key : keys){
            System.out.println(key + " " + map.get(key));
        }
        // add and remove from the map
        map.put("iPhone", "Created by Apple"); map.remove("Android");
        // write again to command line
        Set<String> keys = map.keySet();
        for (String key : map.keySet()) {
            System.out.println(key + " " + map.get(key));
        }
    }
}
```

HashMap: Input

```
class A{int i;}
HashMap<int, A> aMap = new HashMap<int, A>();
                              // Error, key must be an object type
HashMap<Integer, A> aMap = new HashMap<Integer, A>();
           // use the hash code of key then no order is warranted
boolean b = aMap.isEmpty();                  // true

A aA = new A(); aA.i = 1;
aMap.put(1, aA);                             // add new
b = aMap.isEmpty();                          // false
int i = aMap.size();                         // 1

aMap.put(1, aA);                  // replace the older one
i = aMap.size();                  // no new adding with the same key
```

HashMap: Output

```
b = aMap.containsKey(1);                    // true
b = aMap.containsValue(aA);                 // true

A oA = aMap.get(1);                         // access by key
B = oA == aA;                               // true

oA = aMap.get(2);                           // oA = null

b = aMap.remove(1);                         // access by key
```

Summary

- String/StringBuffer/StringBuilder class
- Generic Collection

Thank you