Object-oriented programming

Lecture №6

String, Collections

PhD, Alexander Vlasov

Question?

- *Contract*
- Object
- Class
- Abstraction
- Encapsulation
- Hierarchy
- Modularity

Strings

- **string**: An object storing a sequence of text characters.
  - Unlike most other objects, a `String` is not created with `new`.

```
String name = "text";
String name = expression;
```

  - Examples:

```
String name = "Marla Singer";
int x = 3;
int y = 5;
String point = "(" + x + ", " + y + ")";
```

Indexes

- Characters of a string are numbered with 0-based *indexes*:

```
String name = "P. Diddy";
```

index	0	1	2	3	4	5	6	7
char	P	.		D	i	d	d	y

  - The first character's index is always 0
  - The last character's index is 1 less than the string's length

  - The individual characters are values of type `char` (seen later)

`String` methods

Method name	Description
`indexOf(`**`str`**`)`	index where the start of the given string appears in this string (-1 if it is not there)
`length()`	number of characters in this string

`substring(`**`index1`**`, `**`index2`**`)`
or
`substring(`**`index1`**`)`
	the characters in this string from *index1* (inclusive) to *index2* (exclusive);
	if *index2* omitted, grabs till end of string

`toLowerCase()`	a new string with all lowercase letters
`toUpperCase()`	a new string with all uppercase letters

- These methods are called using the dot notation:

```
String gangsta = "Dr. Dre";
System.out.println(gangsta.length());    // 7
```

`String` method examples

```
//      index 012345678901
String s1 = "Stuart Reges";
String s2 = "Marty Stepp";
System.out.println(s1.length());           // 12
System.out.println(s1.indexOf("e"));       // 8
System.out.println(s1.substring(7, 10))    // "Reg"

String s3 = s2.substring(2, 8);
System.out.println(s3.toLowerCase());      // "rty st"
```

- Given the following string:

```
//        index 0123456789012345678901
String book = "Building Java Programs";
```

  - How would you extract the word "Java" ?
  - How would you extract the first word from any string?

Modifying strings

- Methods like `substring`, `toLowerCase`, etc. create/return a new string, rather than modifying the current string.

```
String s = "lil bow wow";
s.toUpperCase();
System.out.println(s);   // lil bow wow
```

- To modify a variable, you must reassign it:

```
String s = "lil bow wow";
s = s.toUpperCase();
System.out.println(s);   // LIL BOW WOW
```

Strings as parameters

```
public class StringParameters {
    public static void main(String[] args) {
        sayHello("Marty");

        String teacher = "Helene";
        sayHello(teacher);
    }

    public static void sayHello(String name) {
        System.out.println("Welcome, " + name);
    }
}
```

Output:

```
Welcome, Marty
Welcome, Helene
```

Strings as user input

- `Scanner`'s `next` method reads a word of input as a `String`.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
name = name.toUpperCase();
System.out.println(name + " has " + name.length() +
    " letters and starts with " + name.substring(0, 1));
```

Output:

```
What is your name? Madonna
MADONNA has 7 letters and starts with M
```

- The `nextLine` method reads a line of input as a `String`.

```
System.out.print("What is your address? ");
String address = console.nextLine();
```

Comparing strings

- Relational operators such as < and == fail on objects.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name == "Barney") {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

  - This code will compile, but it will not print the song.

  - == compares objects by *references*, so it often gives `false` even when two `String`s have the same letters.

The `equals` method

- Objects are compared using a method named `equals`.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Barney")) {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

- Technically this is a method that returns a value of type `boolean`, the type used in logical tests.

`String` test methods

Method	Description
`equals(`**`str`**`)`	whether two strings contain the same characters
`equalsIgnoreCase(`**`str`**`)`	whether two strings contain the same characters, ignoring upper vs. lower case
`startsWith(`**`str`**`)`	whether one contains other's characters at start
`endsWith(`**`str`**`)`	whether one contains other's characters at end
`contains(`**`str`**`)`	whether the given string is found within this one

```
String name = console.next();
if (name.startsWith("Dr.")) {
    System.out.println("Are you single?");
} else if (name.equalsIgnoreCase("LUMBERG")) {
    System.out.println("I need your TPS reports.");
}
```

Type `char`

- `char` : A primitive type representing single characters.
  - Each character inside a `String` is stored as a `char` value.
  - Literal `char` values are surrounded with apostrophe (single-quote) marks, such as `'a'`, `'4'`, `'\n'`, `'\''`, `'\t'`, `'\\'`

  - It is legal to have variables, parameters, returns of type `char`

```
char letter = 'S';
System.out.println(letter);                 // S
```

- `char` values can be concatenated with strings.

```
char initial = 'P';
System.out.println(initial + " Diddy");   // P Diddy
```

The `charAt` method

- The `chars` in a `String` can be accessed using the `charAt` method.

```
String food = "cookie";
char firstLetter = food.charAt(0);   // 'c'

System.out.println(firstLetter + " is for " + food);
System.out.println("That's good enough for me!");
```

- You can use a `for` loop to print or examine each character.

```
String major = "CSE";
for (int i = 0; i < major.length(); i++) {
    char c = major.charAt(i);
    System.out.println(c);
}
```

Output:

```
C
S
E
```

`char` vs. `int`

- All `char` values are assigned numbers internally by the computer, called *ASCII* values.
  - Examples:
    `'A'` is 65, `'B'` is 66, `' '` is 32
    `'a'` is 97, `'b'` is 98, `'*'` is 42
  - Mixing `char` and `int` causes automatic conversion to `int`.
    `'a' + 10` is 107, `'A' + 'A'` is 130
  - To convert an `int` into the equivalent `char`, type-cast it.
    `(char) ('a' + 2)` is `'c'`

`char` vs. `String`

- `"h"` is a `String`
  `'h'` is a `char` (the two behave differently)

- `String` is an object; it contains methods

```
String s = "h";
s = s.toUpperCase();          // 'H'
int len = s.length();         //  1
char first = s.charAt(0);     // 'H'
```

- `char` is primitive; you can't call methods on it

```
char c = 'h';
c = c.toUpperCase();    // ERROR: "cannot be dereferenced"
```

  - What is `s + 1`? What is `c + 1`?
  - What is `s + s`? What is `c + c`?

Comparing `char` values

- You can compare `char` values with relational operators:

```
'a' < 'b'  and  'X' == 'X'  and  'Q' != 'q'
```

  - An example that prints the alphabet:

```
for (char c = 'a'; c <= 'z'; c++) {
    System.out.print(c);
}
```

- You can test the value of a string's character:

```
String word = console.next();
if (word.charAt(word.length() - 1) == 's') {
    System.out.println(word + " is plural.");
}
```

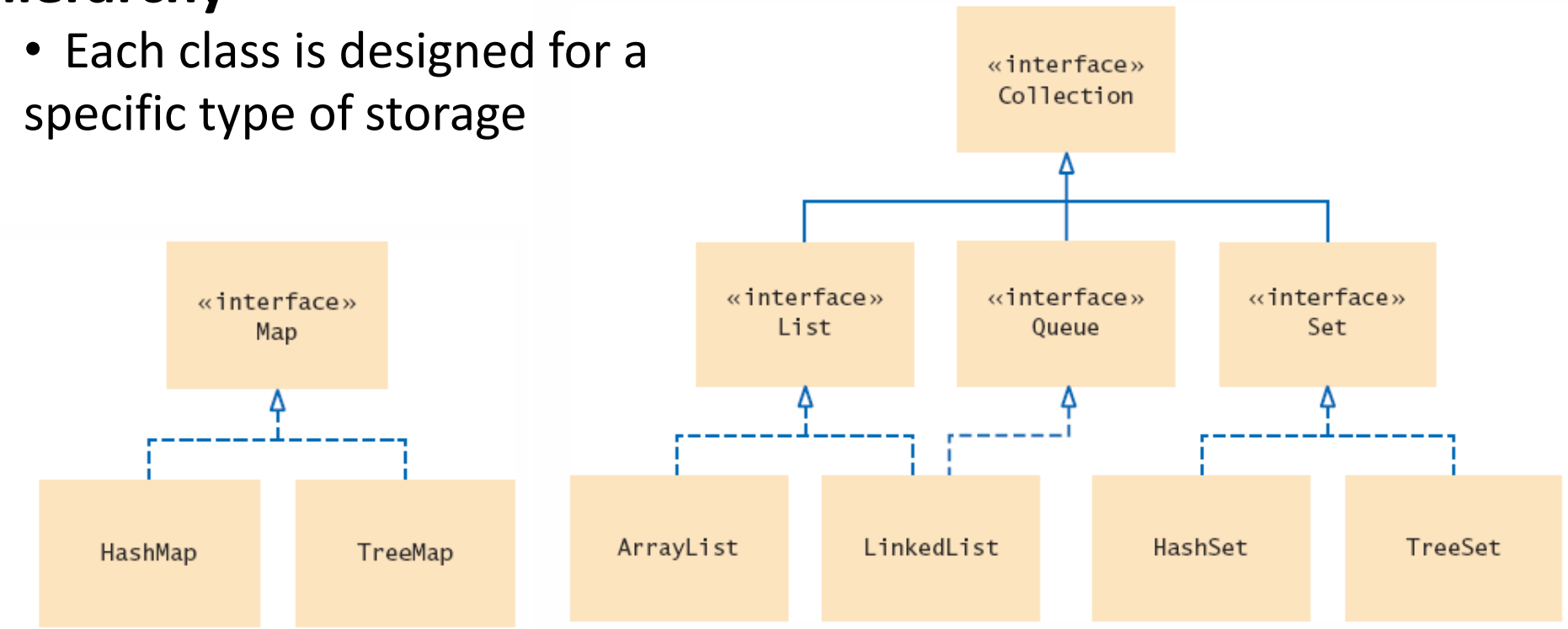
Java Collections Framework

1. When you need to organize **multiple objects** in your program, you can place them into a **collection**
2. The Array class that was introduced later is one of many collection classes that the standard Java library supplies
3. Each **interface type is implemented by one or more classes**

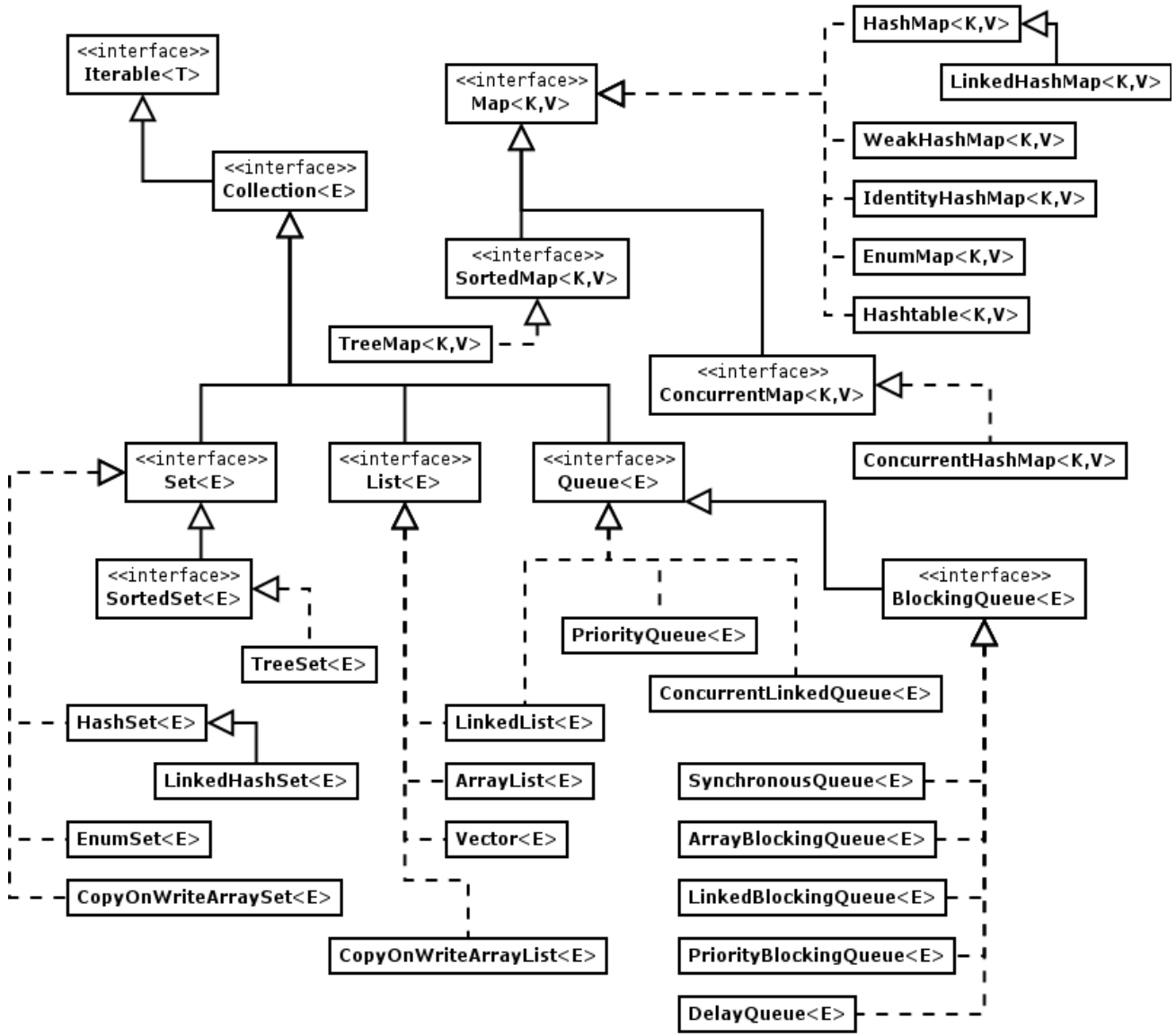
A collection groups together elements and allows them to be accessed and retrieved later

Collections Framework Diagram

- Each collection class implements an **interface** from a **hierarchy**
  - Each class is designed for a specific type of storage

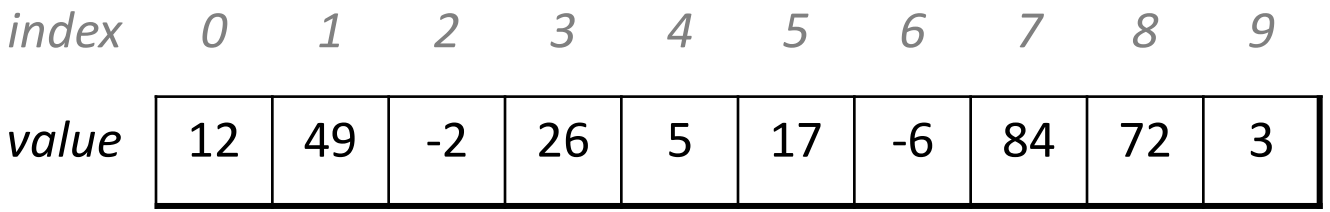


Collections Framework Diagram



Limitations of Arrays

- Size must be specified upon creation
- Can’t add/remove/insert elements later
- No built-in methods for searching, etc.
- Can’t print arrays without Arrays.toString (or Arrays.deepToString)



ArrayLists

- A variable type that represents a list of items.
- You access individual items by *index*.
- Store a single type of **object** (String, etc.)
- Resizable – can add and remove elements
- Has helpful methods for searching for items

```
import java.util.*;

ArrayList<String> myArrayList = new ArrayList<>();
```

ArrayList Methods

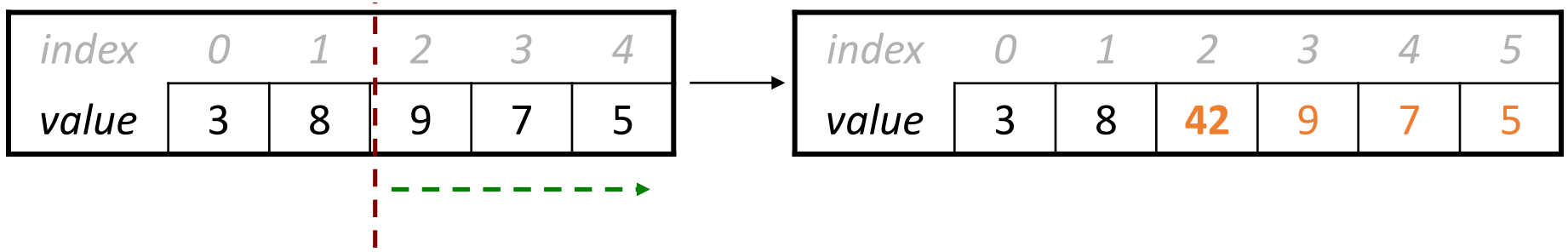
`list.add(value);`	appends value at end of list
`list.add(index, value);`	inserts given value just before the given index, shifting subsequent values to the right
`list.clear();`	removes all elements of the list
`list.get(index)`	returns the value at given index
`list.indexOf(value)`	returns first index where given value is found in list (-1 if not found)
`list.isEmpty()`	returns `true` if the list contains no elements
`list.remove(index);`	removes/returns value at given index, shifting subsequent values to the left
`list.remove(value);`	removes the first occurrence of the value, if any
`list.set(index, value);`	replaces value at given index with given value
`list.size()`	returns the number of elements in the list
`list.toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

Insert/remove

- If you insert/remove in the front or middle of a list, elements **shift** to fit.

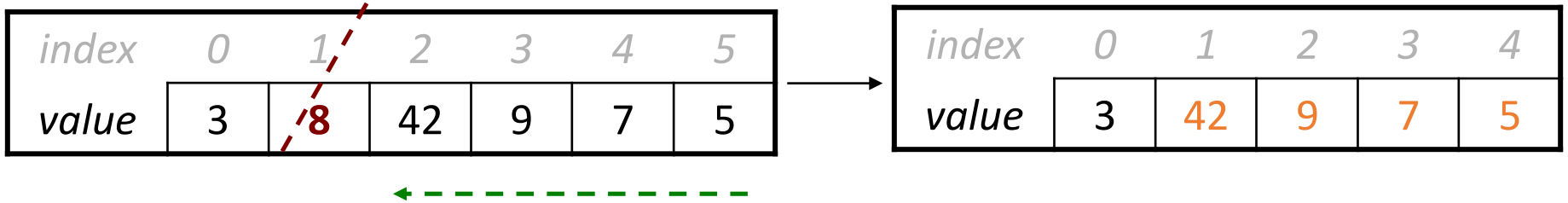
```
list.add(2, 42);
```

  - shift elements right to make room for the new element




```
list.remove(1);
```

  - shift elements left to cover the space left by the removed element



Lists and Sets

- **Ordered** Lists

- **`ArrayList`**
  - Stores a list of items in a **dynamically sized array**
- **`LinkedList`**
  - Allows **speedy** insertion and removal of items from the list

A **list** is a collection that maintains the order of its elements.

Lists and Sets

- **Unordered** Sets

- **`HashSet`**
  - Uses **hash tables** to speed up **finding**, **adding**, and **removing** elements
- **`TreeSet`**
  - Uses a **binary tree** to speed up **finding**, **adding**, and **removing** elements

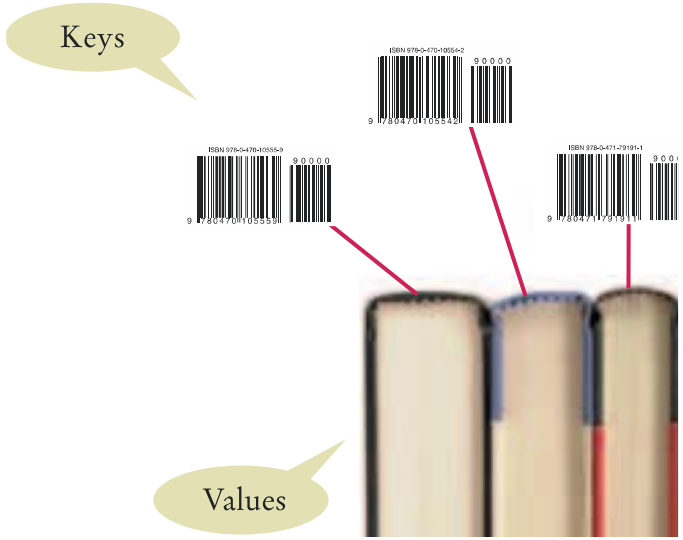
A **set** is an unordered collection of unique elements.

Stacks and Queues

- Another way of gaining efficiency in a collection is to **reduce the number of operations** available
- Two examples are:
  - **Stack**
    - Remembers the **order** of its elements, but it does not allow you to insert elements in every position
    - **You can only add and remove elements at the top**
  - **Queue**
    - Add items to one **end (the tail)**
    - Remove them from **the other end (the head)**
    - Example: A **line of people** waiting for a bank teller

Maps

- A map stores **keys**, **values**, and the associations between them
  - Example:
  - **Barcode keys and books**



A map keeps associations between key and value objects.

- **Keys**
  - Provides an easy way to represent an object (such as a **numeric bar code, or a Student Identification Number**)
- **Values**
  - The **actual object** that is associated with the key

The `Collection` Interface (1)

- **`List`, `Queue` and `Set`** are specialized interfaces that inherit from the **`Collection`** interface
  - **All share the following commonly used methods**

Table 1 The Methods of the `Collection` Interface

`Collection<String> coll = new ArrayList<String>();`	The `ArrayList` class implements the `Collection` interface.
`coll = new TreeSet<String>()`	The `TreeSet` class implements the `Collection` interface.
`int n = coll.size();`	Gets the size of the collection. `n` is now `0`.
`coll.add("Harry");` `coll.add("Sally");`	Adds elements to the collection.
`String s = coll.toString();`	Returns a string with all elements in the collection. `s` is now `"[Harry, Sally]"`
`System.out.println(coll);`	Invokes the `toString` method and prints `[Harry, Sally]`.

The `Collection` Interface (2)

Table 1 The Methods of the `Collection` Interface

`Collection<String> coll = new ArrayList<String>();`	The `ArrayList` class implements the `Collection` interface.
`coll.remove("Harry");` `boolean b = coll.remove("Tom");`	Removes an element from the collection, returning `false` if the element is not present. `b` is false.
`b = coll.contains("Sally");`	Checks whether this collection contains a given element. `b` is now `true`.

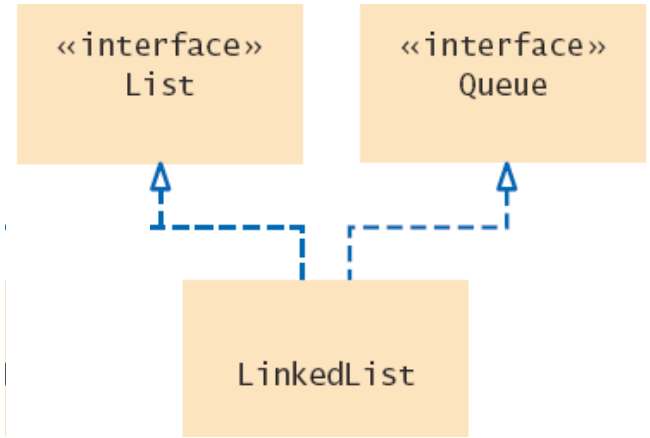
`for (String s : coll)`
`{`
`  System.out.println(s);`
`}`
	You can use the "for each" loop with any collection. This loop prints the elements on separate lines.

`Iterator<String> iter = coll.iterator()`	You use an iterator for visiting the elements in the collection

Linked Lists

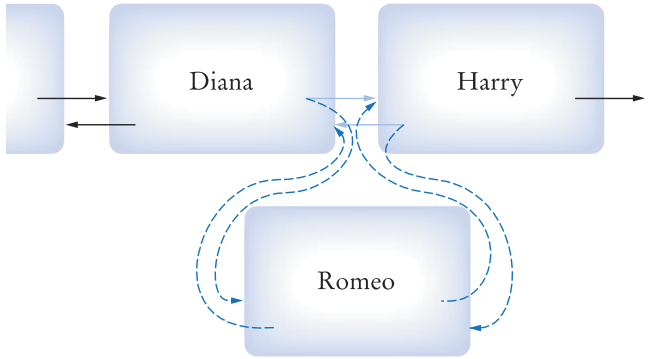
- **Linked lists** use references to maintain an ordered lists of ‘nodes’
  - The ‘**head**’ of the list references the **first node**
  - Each **node** has a **value** and a **reference** to the next node

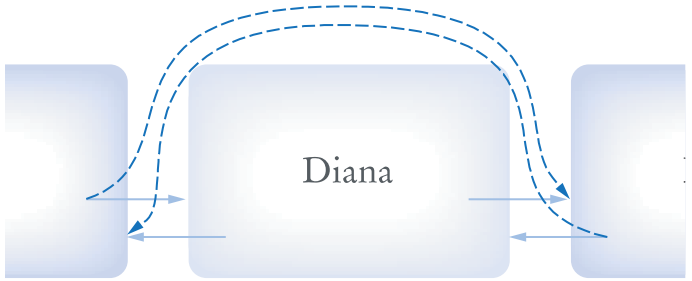
  - They can be used to implement
    - A **`List`** Interface
    - A **`Queue`** Interface



Linked Lists Operations

- Efficient Operations
  - **Insertion** of a node
    - **Find** the elements it goes between
    - **Remap** the **references**
  - **Removal** of a node
    - **Find** the element to remove
    - **Remap** neighbor's **references**
  - Visiting all elements **in order**
- Inefficient Operations
  - **Random access**





Each instance variable is declared just like other variables we have used.

`LinkedList`: Important Methods

Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<String>();`	An empty list.
`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements

Generic Linked Lists

- The **Collection Framework** uses **Generics**
  - Each list is declared with a type field in **< > angle brackets**

```
LinkedList<String> employeeNames = . . .;
```

```
LinkedList<String>
LinkedList<Employee>
```

List Iterators

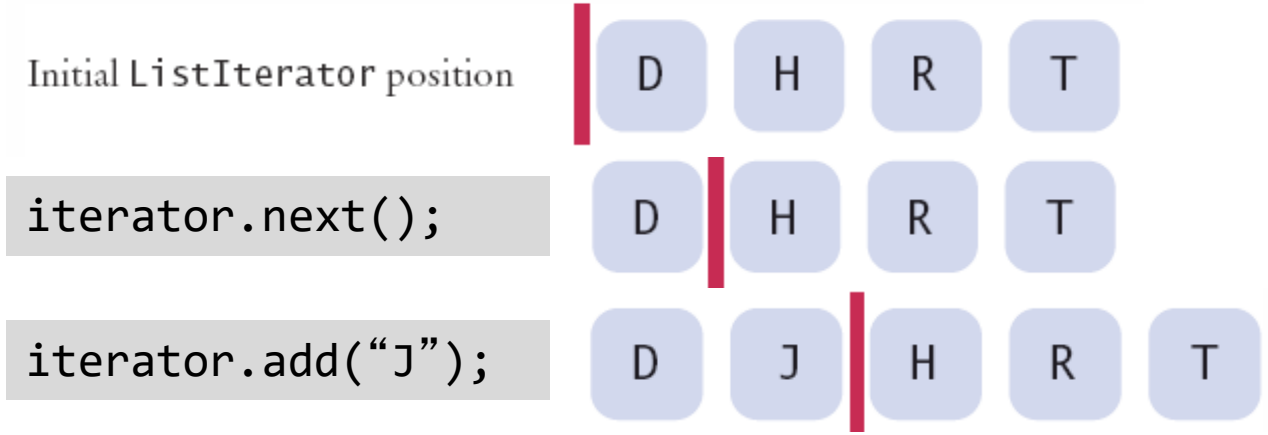
- When **traversing** a `LinkedList`, use a **`ListIterator`**
  - **Keeps track** of where you are in the list.
- Use an **iterator** to:
  - Access elements **inside** a linked list
  - Visit **other** than the **first** and the **last** nodes

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iter = employeeNames.listIterator()
```

Using Iterators

- Think of an iterator as pointing **between** two elements (think of cursor in word processor)

```
ListIterator<String> iter = myList.listIterator()
```



- Note that the **generic type** for the `listIterator` must match the **generic type** of the `LinkedList`

`Iterator` and `ListIterator` Methods

- **Iterators** allow you to move through a list easily
  - Similar to an index variable for an array

Table 3 Methods of the `Iterator` and `ListIterator` Interfaces

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.previous();` `iter.set("Juliet");`	The `set` method updates the last element returned by `next` or `previous`. The list is now `[Juliet]`.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`  s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list. `previous` and `hasPrevious` are `ListIterator` methods.

`iter.add("Diana");`	Adds an element before the iterator position (`ListIterator` only). The list is now `[Diana, Juliet]`.
`iter.next();` `iter.remove();`	`remove` removes the last element returned by `next` or `previous`. The list is now `[Diana]`.

Iterators and Loops

- Iterators are often used in **while** and "**for-each**" **loops**
  `hasNext` returns true if **there is a next element**
  `next` returns a **reference** to the value of the **next element**

```
while (iterator.hasNext())
{
   String name = iterator.next();
   // Do something with name
}
```

```
for (String name : employeeNames)
{
   // Do something with name
}
```

- **Where is the iterator** in the "for-next" loop?
  - Iterators are used '**behind the scenes**'

Adding and Removing with Iterators

- **Adding**

```
iterator.add("Juliet");
```

  - A new node is added **AFTER** the Iterator
  - The Iterator is **moved past the new node**
- **Removing**
  - Removes the object that was returned with **the last call** to **`next`** or **`previous`**
  - It can be **called only once** after `next` or `previous`
  - You **cannot call it immediately after a call to add**.(why?)

If you call the `remove` method improperly, it throws an `IllegalStateException`.

```
while (iterator.hasNext())
{
  String name = iterator.next();
  if (condition is true for name)
  {
    iterator.remove();
  }
}
```

ListDemo.java (1)

- Illustrates adding, removing and printing a list

```
1  import java.util.LinkedList;
2  import java.util.ListIterator;
3
4  /**
5     This program demonstrates the LinkedList class.
6  */
7  public class ListDemo
8  {
9     public static void main(String[] args)
10    {
11       LinkedList<String> staff = new LinkedList<String>();
12       staff.addLast("Diana");
13       staff.addLast("Harry");
14       staff.addLast("Romeo");
15       staff.addLast("Tom");
16
17       // | in the comments indicates the iterator position
18
19       ListIterator<String> iterator = staff.listIterator(); // |DHRT
20       iterator.next(); // D|HRT
21       iterator.next(); // DH|RT
22
```

ListDemo.java (2)

```
23          // Add more elements after second element
24
25          iterator.add("Juliet"); // DHJ|RT
26          iterator.add("Nina"); // DHJN|RT
27
28          iterator.next(); // DHJNR|T
29
30          // Remove last traversed element
31
32          iterator.remove(); // DHJN|T
33
34          // Print all elements
35
36          System.out.println(staff);
37          System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
38       }
39    }
```

Program Run

```
[Diana, Harry, Juliet, Nina, Tom]
Expected: [Diana, Harry, Juliet, Nina, Tom]
```

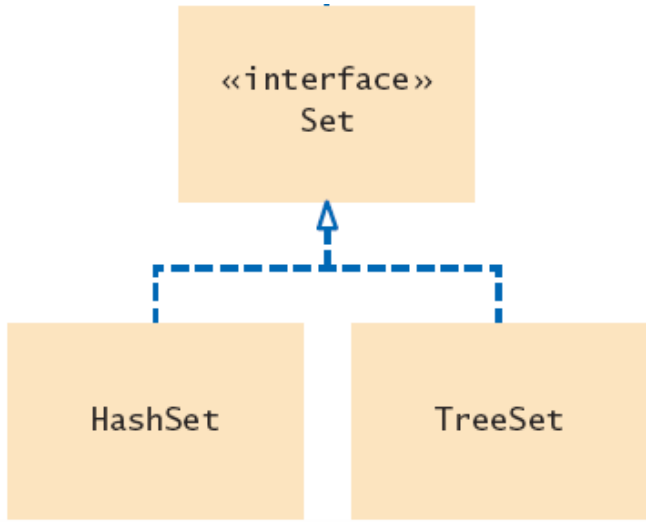
15.3 Sets

- A set is an **unordered** collection
  - It does **not support duplicate elements**
- The collection does **not keep track of the order** in which elements have been added
  - Therefore, it can carry out its operations **more efficiently** than an **ordered collection**

The `HashSet` and `TreeSet` classes both implement the Set interface.

Sets

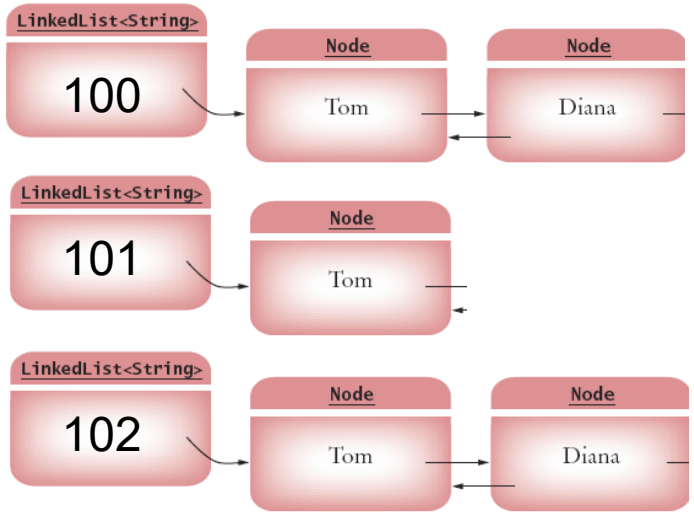
- **`HashSet`**: Stores data in a **Hash Table**
- **`TreeSet`**: Stores data in a **Binary Tree**
- **Both implementations** arrange the set elements so that **finding**, **adding**, and **removing** elements is efficient



Set implementations arrange the elements so that they can locate them quickly

Hash Table Concept

- Set elements are grouped into smaller collections of elements that **share the same characteristic**
  - It is usually based on the result of a **mathematical calculation** on the contents that results in an **integer value**
  - In order to be stored in a hash table, elements must have a **method to compute their integer values**



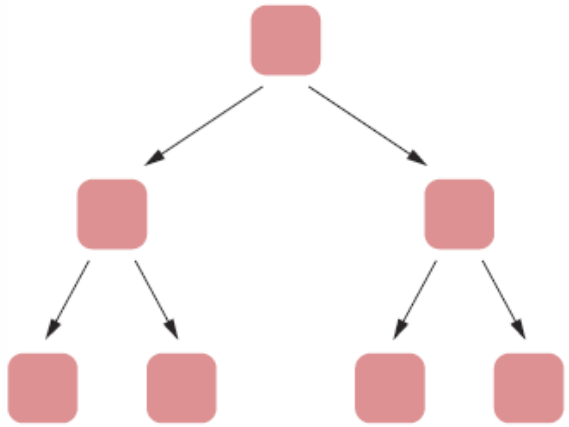
hashCode

- The method is called **hashCode**
  - **If multiple elements have the same hash code (so-called clash), they are stored in a LinkedList**
- The elements must also have an **`equals` method** for checking whether an element equals another like:
  - String, Integer, Point, Rectangle, Color, and all collection classes

```
Set<String> names = new HashSet<String>();
```

Tree Concept

- Set elements are kept in **sorted order**
  - Nodes are not **arranged** in a linear sequence but in a **tree** shape



  - In order to use a `TreeSet`, it must be possible to **compare** the elements and determine which one is "**larger**"

TreeSet

- Use `TreeSet` for classes that **implement** the **`Comparable`** interface
  - `String` and `Integer`, for example
  - The nodes are arranged in a ‘tree’ fashion so that each ‘**parent**’ node has **two child nodes**.
    - The node to the **left** always has a ‘**smaller**’ value
    - The node to the **right** always has a ‘**larger**’ value

```
Set<String> names = new TreeSet<String>();
```

Iterators and Sets

- Iterators are also used when processing sets
  - **`hasNext`** returns true if **there is a next** element
  - **`next`** returns a **reference** to the value of the **next element**
  - **`add`** via the iterator is **not supported** for TreeSet and HashSet

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
  String name = iter.next();
  // Do something with name
}
```

```
for (String name : names)
{
  // Do something with name
}
```

- Note that the elements are **not visited in the order** in which you inserted them.
- They are visited in the order in which the set keeps them:
  - **Seemingly random** order for a HashSet
  - **Sorted** order for a TreeSet

Working With Sets (1)

Table 4 Working with Sets	
`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<String>();`	Use a `TreeSet` if you need to visit the elements in sorted order.
`names.add("Romeo");`	Now `names.size()` is 1.
`names.add("Fred");`	Now `names.size()` is 2.
`names.add("Romeo");`	`names.size()` is still 2. You can't add duplicates.
`if (names.contains("Fred"))`	The `contains` method checks whether a value is contained in the set. In this case, the method returns `true`.

Working With Sets (2)

Table 4 Working with Sets

`System.out.println(names);`	Prints the set in the format `[Fred, Romeo]`. The elements need not be shown in the order in which they were inserted.

`for (String name : names)`
`{`
`. . .`
`}`
	Use this loop to visit all elements of a set.

`names.remove("Romeo");`	Now `names.size()` is 1.
`names.remove("Juliet");`	It is not an error to remove an element that is not present. The method call has no effect.

SpellCheck.java (1)

```
1  import java.util.HashSet;
2  import java.util.Scanner;
3  import java.util.Set;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6
7  /**
8     This program checks which words in a file are not present in a dictionary.
9  */
10 public class SpellCheck
11 {
12    public static void main(String[] args)
13       throws FileNotFoundException
14    {
15       // Read the dictionary and the document
16
17       Set<String> dictionaryWords = readWords("words");
18       Set<String> documentWords = readWords("alice30.txt");
19
20       // Print all words that are in the document but not the dictionary
21
22       for (String word : documentWords)
23       {
24          if (!dictionaryWords.contains(word))
25          {
26             System.out.println(word);
27          }
28       }
```

SpellCheck.java (2)

```
29    }
30
31    /**
32       Reads all words from a file.
33       @param filename the name of the file
34       @return a set with all lowercased words in the file. Here, a
35       word is a sequence of upper- and lowercase letters.
36    */
37    public static Set<String> readWords(String filename)
38       throws FileNotFoundException
39    {
40       Set<String> words = new HashSet<String>();
41       Scanner in = new Scanner(new File(filename));
42       // Use any characters other than a-z or A-Z as delimiters
43       in.useDelimiter("[^a-zA-Z]+");
44       while (in.hasNext())
45       {
46          words.add(in.next().toLowerCase());
47       }
48       return words;
49    }
50 }
```

Program Run

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
...
```

Programming Tip

- Use **Interface References** to Manipulate Data Structures
  - It is considered good style to store a **reference** to a `HashSet` or `TreeSet` in a variable of type `Set`.

```
Set<String> words = new HashSet<String>();
```

    - This way, you have to **change only one line** if you decide to use a `TreeSet` instead.

Programming Tip

- **Unfortunately** the same is not true of the `ArrayList`, `LinkedList` and `List` classes
  - The **`get`** and **`set`** methods for random access are **very inefficient (why)**
- Also, if a method can operate on **arbitrary collections**, use the **`Collection` interface** type for the parameter:

```
public static void removeLongWords(Collection<String> words)
```

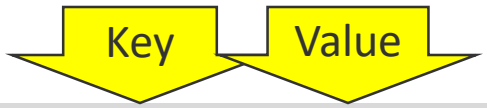
Maps

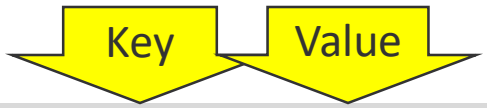
- A **map** allows you to associate elements from a **key** set with elements from a **value** collection.
  - The **`HashMap`** and **`TreeMap`** classes both implement the `Map` interface.
  - Use a map to look up objects by using a key.

HashMap Examples

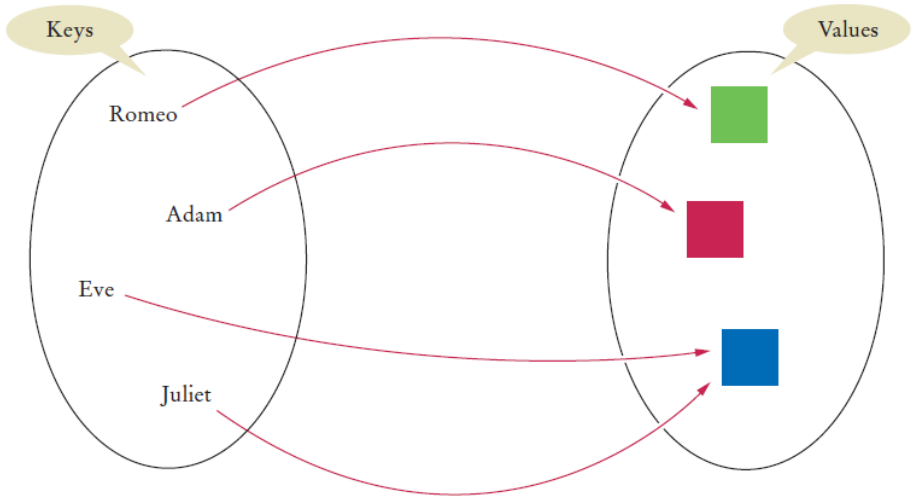
- **Phone book:** name -> phone number
- **Search engine:** URL -> webpage
- **Dictionary**: word -> definition
- **Bank**: account # -> balance
- **Social Network**: name -> profile
- **Counter**: text -> # occurrences
- And many more…

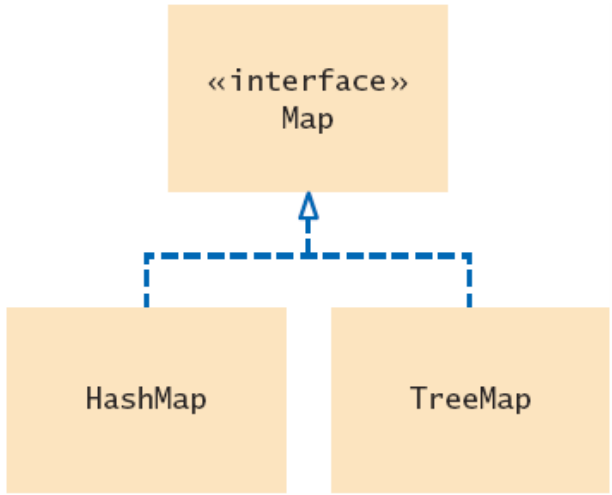
Maps






```
Map<String, Color> favoriteColors = new HashMap<String, Color>();
```





The key "**unlocks**" the "**data**" (value)
A map is like a mathematical function
Mapping between two sets.

Working with Maps (Table 5)

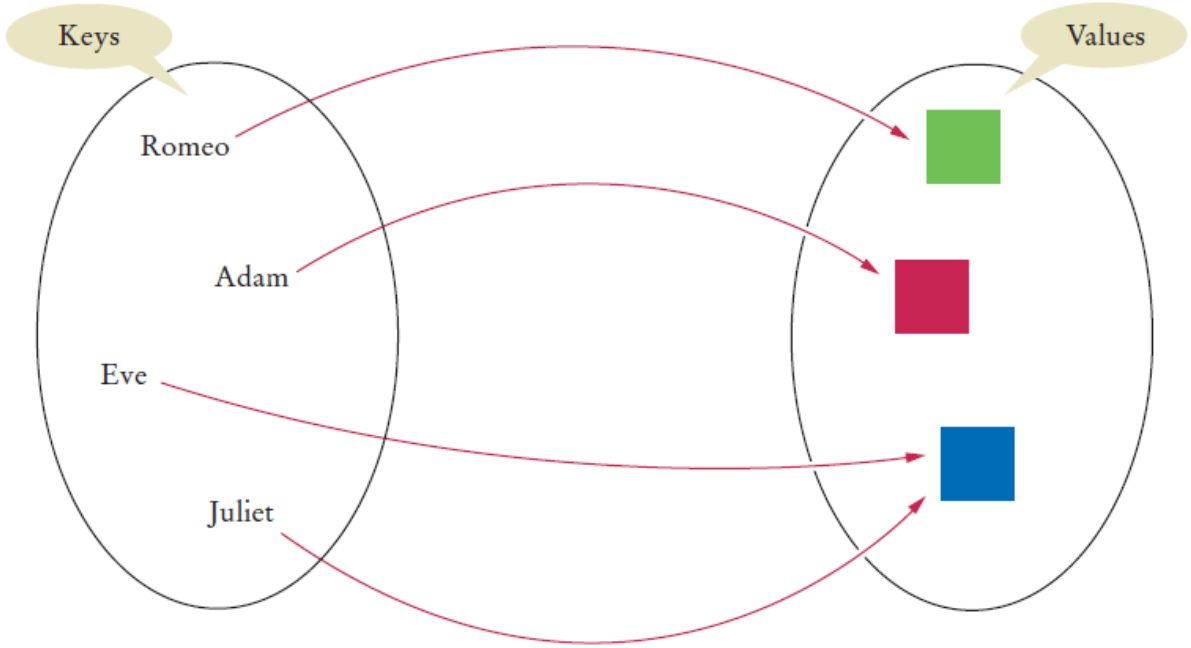
`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<String, Integer>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.
`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

`for (String key : scores.keySet())`
`{`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

Key Value Pairs in Maps

- Each key is **associated** with a value




```
Map<String, Color> favoriteColors = new HashMap<String, Color>();
favoriteColors.put("Juliet", Color.RED);
favoriteColors.put(“Romeo", Color.GREEN);
Color julietsFavoriteColor = favoriteColors.get("Juliet");
favoriteColors.remove("Juliet");
```

Iterating through Maps

- To **iterate** through the map, use a `keySet` to get the list of keys:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
  Color value = m.get(key);
  System.out.println(key + "->" + value);
}
```

To **find all values in a map,**
1/ **iterate** through the **key set** and
2/ find the **values** that **correspond** to the **keys**.

MapDemo.java

```
1  import java.awt.Color;
2  import java.util.HashMap;
3  import java.util.Map;
4  import java.util.Set;
5
6  /**
7     This program demonstrates a map that maps names to colors.
8  */
9  public class MapDemo
10 {
11    public static void main(String[] args)
12    {
13       Map<String, Color> favoriteColors = new HashMap<String, Color>();
14       favoriteColors.put("Juliet", Color.BLUE);
15       favoriteColors.put("Romeo", Color.GREEN);
16       favoriteColors.put("Adam", Color.RED);
17       favoriteColors.put("Eve", Color.BLUE);
18
19       // Print all keys and values in the map
20
21       Set<String> keySet = favoriteColors.keySet();
22       for (String key : keySet)
23       {
24          Color value = favoriteColors.get(key);
25          System.out.println(key + " : " + value);
26       }
27    }
28 }
```

Program Run

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

What data structure should I use?

- Use an **array** if…
  - Order matters for your information
  - You know how many elements you will store
  - You need the most efficiency
- Use an **ArrayList** if…
  - Order matters for your information
  - You do not know how many elements you will store, or need to resize
  - You need to use ArrayList methods
- Use a **HashMap** if…
  - Order doesn’t matter for your information
  - You need to store an *association* between two types of information
  - You do not know how many elements you will store, or need to resize
  - You need to use HashMap methods

Algorithms for collection

- sort(List), sort(List, Comparator)
- binarySearch(List, Object), binarySearch(List, Object, Comparator)
- reverse(List)
- shuffle(List), shuffle(List,Random)
- fill(List,Object)
- copy(List, List)
- min(Collection), min(Collection, Comparator)
- max(Collection), max(Collection,Comoarator)

```
List<String> strList = new ArrayList<String>();
        strList.add("A");
        strList.add("C");
        strList.add("B");
        strList.add("Z");
        strList.add("E");
Collections.sort(strList);
for (String str: strList) {
                System.out.print(" " + str);
}
```