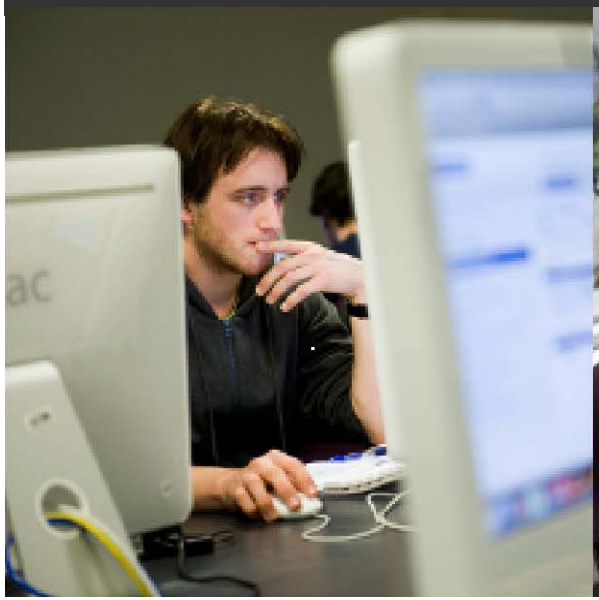
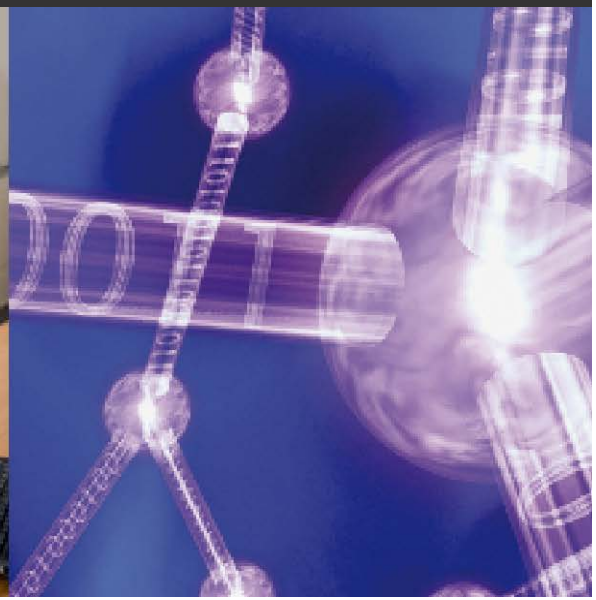
MONASH University

Information Technology

Module 11:
Java's Collections API

FIT2034 Computer Programming 2
Faculty of Information Technology





Introduction to Data Structures

Abstract Data Types

- An ***Abstract Data Type*** (***ADT***) is the specification of a set of data plus a set of allowable operations on the data. This matches closely the *OO* paradigm of *Encapsulation*, where classes provide the specifications of attributes and the methods to manipulate the attributes.

- The abstractions permit the users of the ADT to be able to ignore the actual implementation details of the data/algorithms, and concentrate on the logic required to use it. This also encourages re-usability.

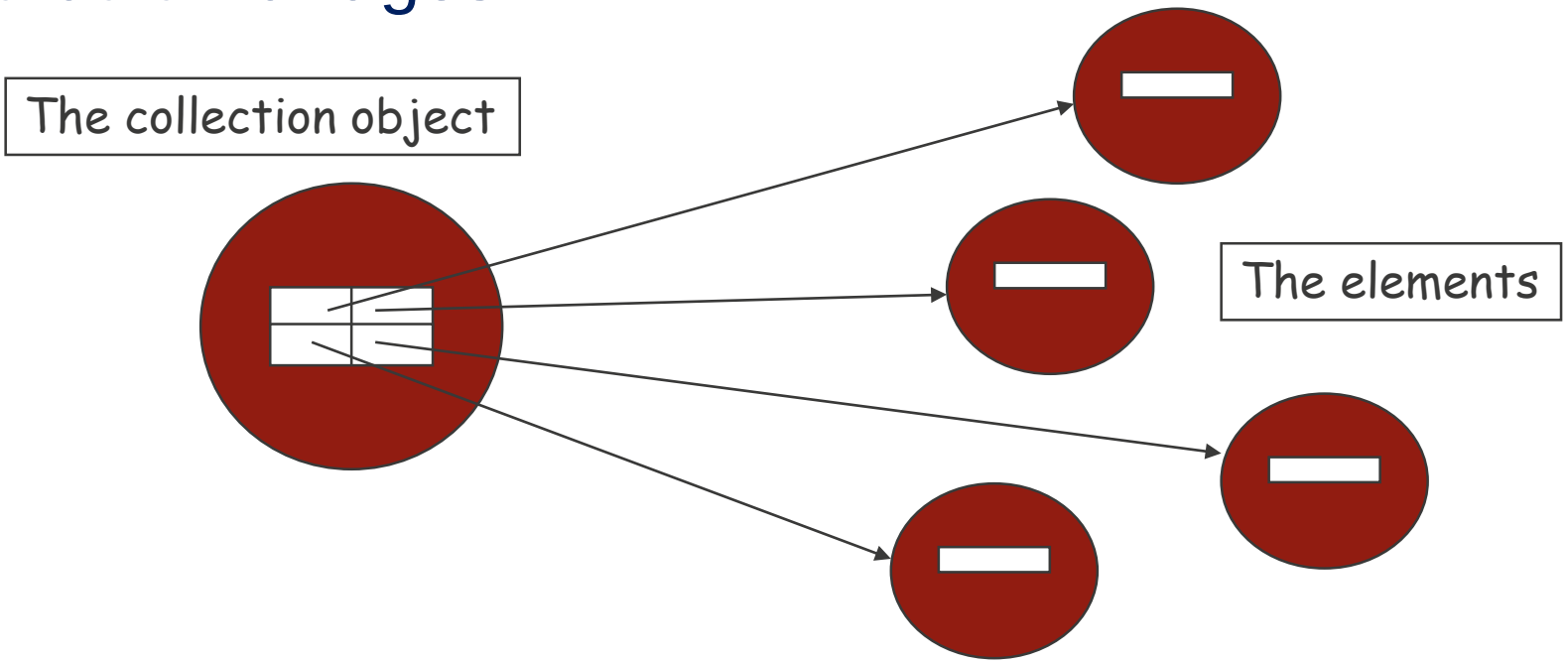
Data Structures

Data Structures (D.S.) are ADTs that:

- Manage a collection/group of data items (objects)
- Provide a range of standard operations to work on the data:
  - Insert/Add
  - Remove
  - Sort
  - Find/Search
- Hides the implementation of the above actions from the programmer's view
- A Collection is a class which provides an interface to the data structure.

Collections – Visual Interpretation

- The Collection Object has some set of elements that it manages



  - The exact manner in which it manages them is determined by the type of data structure it implements internally

Static versus Dynamic Data Structures

- Static Data Structures
  - Example: **plain array**
  - Size is fixed
  - Easy to set up & manipulate
  - Not always flexible or efficient
- Dynamic D.S.
  - Size grows & shrinks as required
  - More complicated to manipulate
  - More flexible
  - Often faster for large amounts of data
  - Eg. a *Java* **ArrayList**

Applications of Collections

- Assume we have a large set of unique objects.
  - Example: Customers of a Bank, and Accounts of the Customers.
- Customers may have:
  - Name, Address, tax file number, phone number, etc.
- Accounts may have:
  - Type, balance, restricted access, account number, etc.
- How will we organise the Customer and Account objects?
  - For accessing a particular customer's details
  - For adding a new customer
  - For adding a new account for a particular customer

Variety of Ways to Manage Collections

- There are different ways to organise a collection of data of some common type:
  - List
  - Map
  - Set

ADT: List

- The List ADT is for an ordered collection of elements accessed by integer indexes
- You can add to the end of the list
- You can access any element of the list using the element's index
- You can remove an element at a particular index – other elements will be "moved left" by one index.
- You can search for an element
  - If the list is sorted, you can search relatively quickly.

ADT: Map

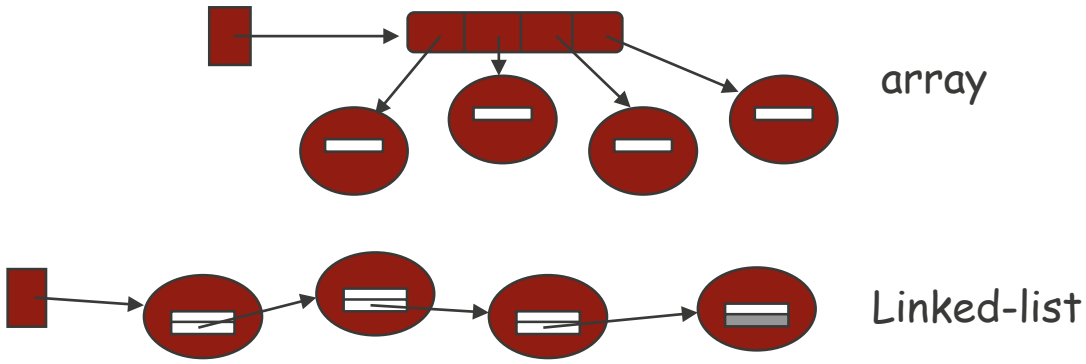
- The Map ADT is a collection of key-value pairs.
- The key is some value that uniquely maps to a particular element
- Examples:
  - a Dictionary contains words, which maps to definitions of the word.
  - A Customer number maps to the corresponding Customer object
  - An account number maps to the corresponding Account object
- Using the key, you can quickly obtain the associated data
  (the value)
- Multiple keys could map to the same value (alias)

ADT: Set

- The Set ADT is a collection of elements that is guaranteed to contain no duplicates
- Often Sets are compared against each other, or joined
  - Find the difference
  - Find the similarities
  - Find the combination of the elements
- Example:
  - Find the commonalities between set of products purchased by different customers – e.g. does everyone who bought bread, also buy milk?
  - Find the differences between people – what things do people generally buy when they buy socks

Implementations of Collections

- Collections can generally be implemented with two underlying implementation structures : array-based structures or link-based structures.



- For this week, we will discuss the pros and cons of using these to implement *Dynamic Data Structures*.

Arrays

- Built-in support in the Java programming language, in different forms (both static & dynamic).
- Available in almost every high-level programming language.
- Map easily to underlying machine memory architecture.
- Use indices to access individual elements.
- Access speed is typically constant – independent of an element's position.

Linked-List

- Can be implemented in every language that supports references or pointers.
- Each ***element*** of the data structure (called a ***node***), contains 2 things:
  - A reference to the actual data,
  - a reference to another node, (i.e. a “link” or component of the structure)
- The final node typically has a link to “null”
- Elements are not necessarily stored in consecutive positions in memory. (cf. Array)
- Access speed is dependent of an element’s position in the structure. Traversal is typically linear.

Typical Operations on Dynamic Data Structures

- Insertion – adding an element into the data structure
- Removal – removing an element from the data structure
- Access/Search – for a particular element or group of elements matching criteria
- Sort – using some criteria to order the elements for later operations.

Comparisons of Arrays vs Linked structures

- When choosing the implementation, there are several issues to consider:
  - Capacity & Size
    - Both can grow and shrink, but can have vastly different overheads
    - For an array, the maximum capacity is not always the same as the actual size
  - Access
    - Simple & quick for arrays; slightly more complicated for linked-lists (but modern languages provides classes to represent these easily)
  - Memory Requirement
    - Linked-list typically has slightly higher memory requirements

Verdicts

Issue	*Arrays*	*Linked-Lists*
Growing	Expensive	Cheap
Inserting to Maintain Order	Expensive	Cheap
Inserting if order is unimportant	Cheap	Cheap
Deleting to Maintain Order	Expensive	Cheap
Deleting if order is unimportant	Cheap	Cheap
Accessing Elements	Cheap	Expensive
Sorting	Expensive	Expensive
Memory Requirements	Cheap	Expensive (slightly)

Data Structures Summary

- Most collections are based on either of these structures (or a combination of them).

- Understanding the issues is very important to assess suitability of collections for a specific purpose.

Generics

Generics

- Allows a class to be specified with the data type of some attributes or parameters being deferred.
- Example:

```
public class Example<T>
{
  private T data;
  public void setData(T newData)
  { data = newData; }

  public T getData()
  { return data;    }
}
```

Type Parameter

```
public Example<String> word;

…

word = new Example<String>();
```

word's setData() method will expect a String reference

Generics cont'd.

- Generics allows programmers to specify the type to be used at compile time, so that the compiler can perform type-checking, to avoid objects of the wrong type being used
- Used by the Collection classes and interfaces in Java.
  - Checks correct data items are added to the collection
  - Removes the need to type-cast objects within methods which uses the collections.

The Java Collections Framework

The Java Collections Framework

- In *Java 1.1*: Some (not many) collection implementations (Vector, Hashtable)
- In *Java 2* (a.k.a. JDK 1.2): A collection Framework
- New since *Java 5.0*: Typed Collections (using Generics); and several new implementations

- ***Framework*** = a set of interfaces, classes and algorithms focused on a specific purpose
- We will examine **ArrayList, LinkedList, HashSet**, **TreeSet, HashMap** and **TreeMap** in this lecture.

Framework Components

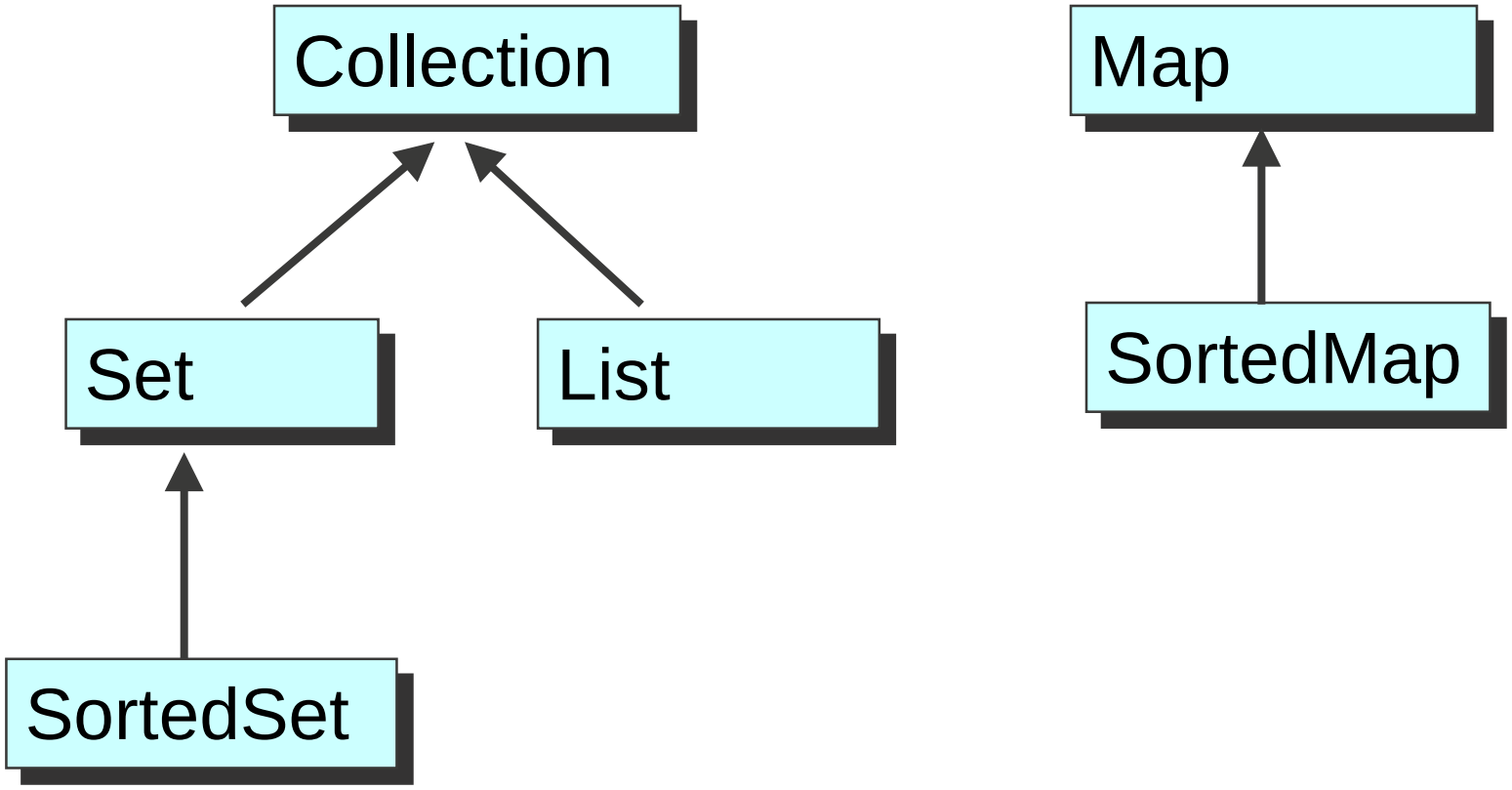
- Interfaces: Java interfaces; *describe* different logical types of collections
- Implementations: Java classes; *implement* the logical type
- Algorithms: methods for *manipulating* the collections; available for different collection types

Interfaces

- The logical or “outside” view of collections
- Concentrate on functionality
- Abstract (hide details) from implementation

Usually, when using a collection data structure, its interface (the functionality provided by the data structure) should be chosen first.

Organisation: The Core Collections Interfaces



Interfaces of the Core Collections

- **List**
  - single-value elements
  - ordered (sequence); allows duplicates; indexed access
  - includes **ArrayList** & **LinkedList**
  - *example*: help desk schedule
- **Set**
  - no duplicates; unordered
  - includes **HashSet** & **TreeSet**
  - *example*: students in a class
- **Map**
  - Key-value map (unique keys); keyed access
  - includes **HashMap** & **TreeMap**
  - Unordered
  - example: student-address pairs in a database

The List Interface

```
public interface List<E> extends Collection
{
   // Positional Access
   E get(int index);
   E set(int index, E element);
   void add(int index, E element);
   E remove(int index);
   boolean addAll(int index, Collection<? Extends E> c);

   // Search
   int indexOf(Object o);
   int lastIndexOf(Object o);

   ListIterator<E> listIterator();
   ListIterator<E> listIterator(int index);
   List<E> subList(int from, int to);
}
```

The Set Interface

```
public interface Set<E> extends Collection<E>
{
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(E element);
   boolean remove(Object o);
   Iterator<E> iterator();

   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? Extends E> c);
   boolean removeAll(Collection<?> c);
   boolean retainAll(Collection<?> c);
   void clear();
   Object[] toArray();
   Object[] toArray(Object a[]);
}
```

The Map Interface

This slide does not show the complete Interface definition.

```
public interface Map<K, V>
{
      Object put(K key, V value);
      Object get(Object key);
      boolean containsKey(Object key);
      boolean containsValue(Object value);

      public Set<K> keySet();
      public Collection<v> values();
      public Set<Map.Entry><K,V> entrySet();

        // Interface for entrySet elements
      public interface Entry<K, V> {
            K getKey();
            V getValue();
            V setValue(V value);
      }
}
```

The Basis for Implementations

Java provides various implementations of the different types of collections with different underlying mechanisms:

- Array-based structure
- Link-based structure
- Tree (a more complex link-based structure)
  - Aim to get a “balanced” tree to minimize access time
- Hashtable (combines links and arrays)
  - very fast lookup
  - uses "hash values" to determine position

Java Collections Implementations

- For each Interface, you can choose from one or more implementations (classes)
- For *List*:
  - **ArrayList** (constant access time, used most of the time);
  - **LinkedList** (fast insertion)
- For *Set*:
  - **HashSet** (faster; used most of the time);
  - **TreeSet** (sorted)
- For *Map*:
  - **HashMap** (fast);
  - **TreeMap** (sorted)

How to Use the Framework (typical steps)

- Decide on which collection provides the required functionality – e.g. **List**
- Choose an implementation (class), eg. **LinkedList**
- Declare an instance of the Collection using the interface type, eg. **List<Person> myCollection;**
- Create an instance of the Collection using the implementation type, and assign that to the interface variable, eg.

  myCollection = new LinkedList<Person>();
- Use the collection varable via its interface methods, eg. **myCollection.add(aPerson);**

Declare using the Interface type only

- Use only interfaces (e.g. List) for declarations.
- Use the implementation class only once – for the creation (instantiation) of the object.

```
List<Things> things = new LinkedList<Things>();
process(things);

void process(List<Things> listToProcess)
{
    ...
}
```

Working with Sets

Example: Using a Set

- We want a **Set** if the following requirements are to be satisfied :
  - Unordered collection
  - Ability to add and remove objects easily
  - Ability to reject duplicate objects (ones which are already in the set)

Example: Using a Set (cont.)

- Once we have chosen the **Set** interface, we have two implementations to choose from :
  - **HashSet**
  - **TreeSet**
- We would choose **HashSet** :
  - If we have many unique elements and access them frequently
    (This is the fastest type of collection)
- We would choose **TreeSet** :
  - If we want to always have elements returned in sorted order

Example: Using a Set (cont.)

- Our Code:

```
public class AClass
{
    Set<Things> mySetCollection;
    . . .
    public AClass()
    {
        mySetCollection = new HashSet<Things>();
              // or
        mySetCollection = new TreeSet<Things>();
    }
}
```

Example: Using a Set (cont.)

- Now we can use the methods of the **Set** interface to manipulate our collection
- No need to know which implementation was used.

  e.g.

  mySetCollection.add(element);

  works regardless of whether HashSet or TreeSet is used

Working with Iterators

Iterators

- Iterators are used to iterate through collections retrieving each object

returns an Iterator object for a collection

Checks if any more elements

Returns the next object

Removes the last object returned by next

```
public interface Collection<E>
{
        ...
    Iterator<E> iterator();
}

public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove();
}
```

When to use an Iterator instead of a for loop?

- The traditional **for** loop can always be used to manipulate collections, however sometimes an ***iterator*** is more convenient/safer to use, e.g.:
  - When you are removing elements from a collection midway through iterating over the collection
  - When you need to filter or replace elements in a collection
  - When you need to iterate over multiple collections
- No counters/indices are needed!

Using Iterators

- A typical way for iterating through a collection:

```
List<String> list1;
list1 = new LinkedList<String>();  // or ArrayList()

public void iteratingThrough()
{
  Iterator<String> list1Iterator = list1.iterator();
  while (list1Iterator.hasNext())
  {
      String temp = list1Iterator.next();
      // Do something with the element.....
  }
}
```

Enhanced For loop (The “for each” loop)

- Can be used in place of an iterator to improve code readability.
- Eg.

```
List<String> list1;
list1 = new ArrayList <String>();
//  some code here to add elements into list1….

for (String s : list1)
//  this means: “for each String s found in the List list1, …”
{
  System.out.println(s);
  //  Do something else useful with the element …
}
```

Example: Processing an ArrayList of OrderLines

- Using normal for loop to calculate total cost (sample solution):

```
private ArrayList<OrderLine> orderLines;

    ...              // code to initialise orderLines

public double calculateTotalCost() {
    double totalCost = 0;
    for (int i=0; i<orderLines.size(); i++) {
        OrderLine orderLine = orderLines.get(i);
        totalCost += orderLine.getFood().getPrice() *
           orderLine.getQuantity();
    }
    return totalCost;
 }
```

Example: Processing an ArrayList of OrderLines

- Using enhanced for loop to calculate total cost:

```
private ArrayList<OrderLine> orderLines;

    ...              // code to initialise orderLines

public double calculateTotalCost() {
    double totalCost = 0;
    for (OrderLine orderLine : orderLines) {
        totalCost += orderLine.getFood().getPrice() *
           orderLine.getQuantity();
    }
    return totalCost;
 }
```

Working with Maps

Methods of the Map Interface

- Many methods are the same as for **Collection**
- The methods apply to both **HashMaps** and **TreeMaps**

Implementations of the Map Interface

- **HashMap**
  - Stores a pair which represents a key **and** a value associated with that key
  - Examples
    - Name and phone number
      - And many more possibilities, eg. the value could be another collection
  - An unordered collection – makes no guarantees about the order in which the objects will be stored.
  - Uses a hashcode created from the key element to provide very fast access to elements

Implementations of the Map Interface (cont)

- **TreeMap**
  - Same as the **HashMap** except that the elements are guaranteed to be returned for viewing in sorted order.

    (Note this does not mean stored in sorted order)
  - To have keys stored in sorted order use **SortedMap**

Example: Using a Map

- Adding elements: **put(K key,V value)**

```
Map<String,Integer> map1 = new HashMap<String,Integer>();
Map<String,Integer> map2 = new TreeMap<String,Integer>();
. . .
for (String s: names)
{     map1.put(s, number1);
      map2.put(s, number1);
}
```

Example: Using a Map

- Retrieving a value
  - **Value get(Object key)**
  - Returns the value specified by the key

```
Integer id = map1.get("Joe");

System.out.println("Joe's student id is " + iD.toString());
```

Example: Using a Map

- Calling **keySet()** on the **Map** returns a **Set** of **keys.**
- Calling **values()** on the **Map** returns a **Collection** (set or list) of the **Map** entries
- We can now use these in the same way as we can use other Collections:

```
Set<String> myKeySet = new HashSet<String>();
Collection<String> myValues = new HashSet<String>();
myKeySet = map1.keySet();
myValues = map1.values();
```

Multi-Dimensional Arrays

Multi-Dimensional Arrays

- Previously we have looked at arrays
- We have as many dimensions in an array as we want (1, 2, 3, etc.)

Index 0	Index 1	Index 2

- 1D Array

Index 0,0	Index 0,1	Index 0,2
Index 1,0	Index 1,1	Index 1,2
Index 2,0	Index 2,1	Index 2,2

- 2D Array

Multi-Dimensional Arrays

- As we add more dimensions we increase memory footprint
- Need to be careful with how we use them
- Examples of multi-dimensional arrays
  - Final Assignment will need to use 2D array
  - The game Minecraft makes use of 3D arrays

Declaring a Multi-Dimensional Array

- Creating a 2D Array

```
String[][] stringTable;
stringTable = new String[5][5];
```

- Creating a 3D Array

```
String[][][] stringCube;
stringCube = new String[5][5][5];
```

- NOTE: This creates the array but does not initialize any values (All will be null to begin with)

Accessing Elements of an array

- Previously we have used a for-loop to access elements of an array
- We can still do this for multi-dimensional arrays
- Requires nested for loops

Example: Accessing Elements 2D

```
final int size = 5;

String[][] stringTable;
stringTable = new String[size][size];

for(int row = 0; col < size; col++)
{
      for(int col = 0; col < size; col++)
      {
            // Access our elements here
      }     stringTable[row][col] = “Hello”

}
```