IN1007 Programing in Java

Lecture 7 (Week 10): Collections

Announcements

- **Final assessment (Viva) →** Exact times and room details now available on Moodle
- **IMPORTANT:** Module evaluation survey

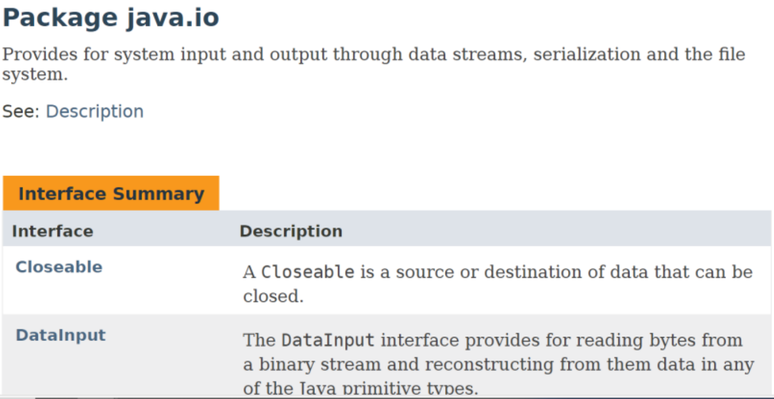
Today's Lecture

- What is a Package in Java?
- What is an Interface in Java?
- What is a Collection in Java?
- The Java Collections Framework
  - Lists
  - Sets
  - Maps

What is a Package in Java?

- Java provides a large number of classes
- Keeping all classes in the same folder can make them hard to access
- Java uses packages to arrange classes (similar to how folders arrange files)
- Java API groups classes into packages according to functionality

Example: the **java.io** package



Package java.io

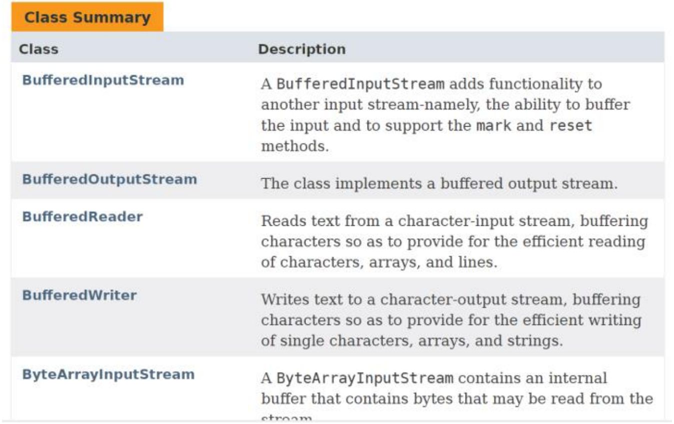
Provides for system input and output through data streams, serialization and the file system.

See: Description

Interface Summary

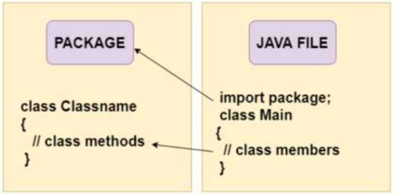
Interface	Description
Closeable	A `Closeable` is a source or destination of data that can be closed.
DataInput	The `DataInput` interface provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types.

Example: the **java.io** package



Class Summary

Class	Description
BufferedInputStream	A `BufferedInputStream` adds functionality to another input stream-namely, the ability to buffer the input and to support the `mark` and `reset` methods.
BufferedOutputStream	The class implements a buffered output stream.
BufferedReader	Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
BufferedWriter	Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
ByteArrayInputStream	A `ByteArrayInputStream` contains an internal buffer that contains bytes that may be read from the stream.



```
class Classname
{
  // class methods
}
```

```
import package;
class Main
{
 // class members
}
```

syntax:

```
import package.name.ClassName;   // To import a certain class only
import package.name.*   // To import the whole package
```

Example:

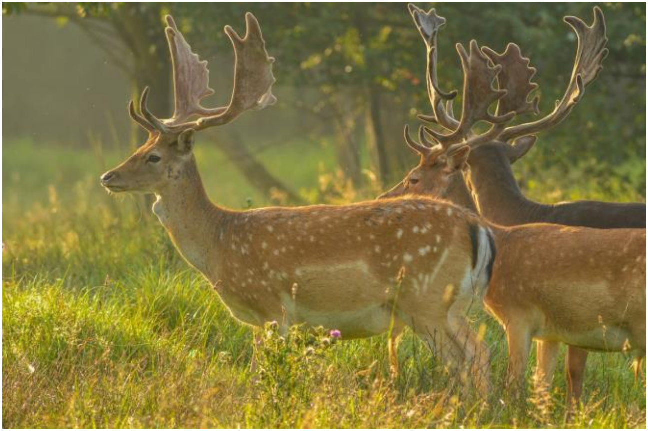
```
import java.util.Date; // imports only Date class
import java.io.*;     // imports everything inside java.io package
```

Importing packages

Today’s Lecture

- What is a Package in Java?
- **What is an Interface in Java?**
- What is a Collection in Java?
- The Java Collections Framework
  - Lists
  - Sets
  - Maps

Abstraction: Try sketching this scene (1 min)



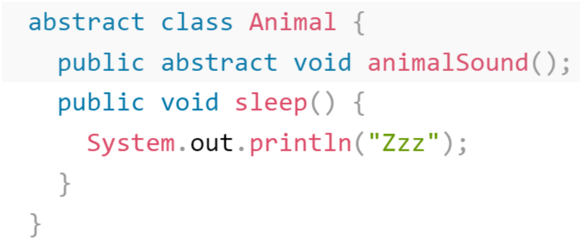
Abstraction Explained

- How many deer were in the scene?
- How many antlers?
- How many dots in their fur?
- How many leaves on the tree?

Sometimes you don’t need every detail to create something.

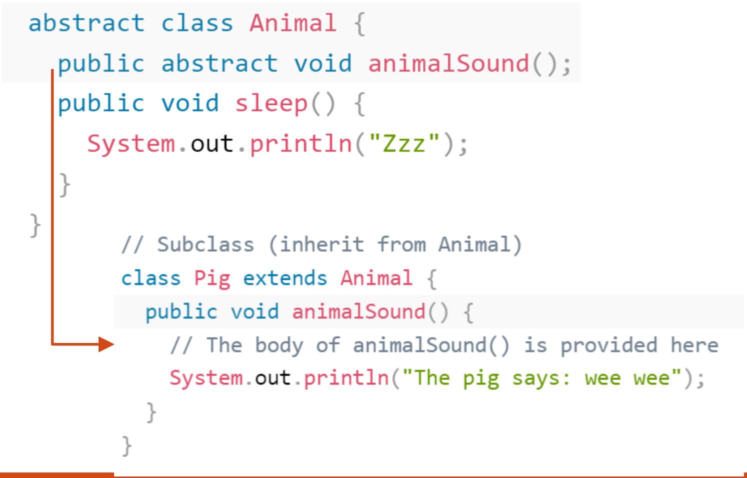
Abstraction = reducing detail

Abstract Method



```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

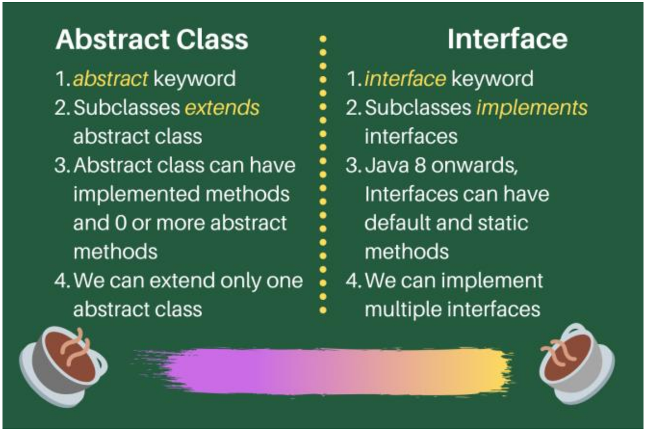
Abstract Method



```java
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

```java
// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}
```

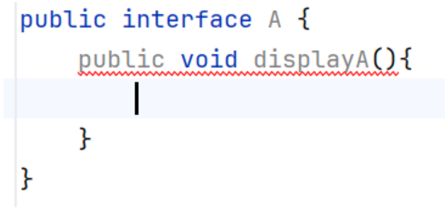
Abstract Classes VS. Interfaces



Abstract Class	Interface
1. *abstract* keyword	1. *interface* keyword
2. Subclasses *extends* abstract class	2. Subclasses *implements* interfaces
3. Abstract class can have implemented methods and 0 or more abstract methods	3. Java 8 onwards, Interfaces can have default and static methods
4. We can extend only one abstract class	4. We can implement multiple interfaces

Image source: Difference between Abstract Class and Interface in Java | DigitalOcean

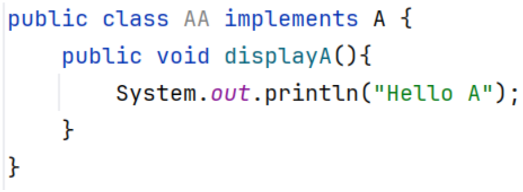
Interface: Try this…



```
public interface A {
    public void displayA(){

    }
}
```

What is the error?

Implementing an interface (Try it!)

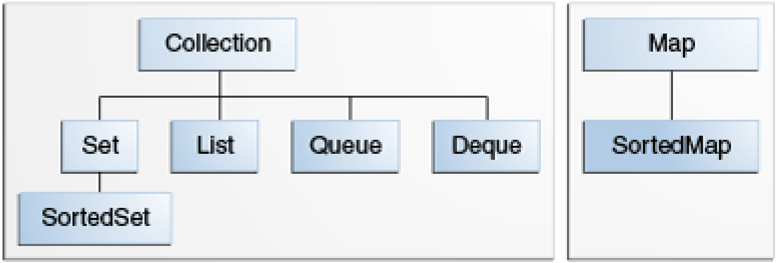

```java
public class AA implements A {
    public void displayA(){
        System.out.println("Hello A");
    }
}
```

Today’s Lecture

- What is a Package in Java?
- What is an Interface in Java?
- **What is a Collection in Java?**
- **The Java Collections Framework**
  - Lists
  - Sets
  - Maps

Java Collections

A *collection* is an object that represents a group of objects



Java Collections Framework

A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated **independently of implementation details**.

Components of the framework include:

Interfaces **Implementations** **Algorithms**

General framework: Collections

$\longrightarrow$ General way to put together classes and objects, with specific methods to manipulate them.

$\longrightarrow$ Many ways to talk about "collections" of objects. We focus on three of them:

- Lists
- Sets
- Maps

Lists

A List is an ordered Collection (sometimes called a *sequence*).

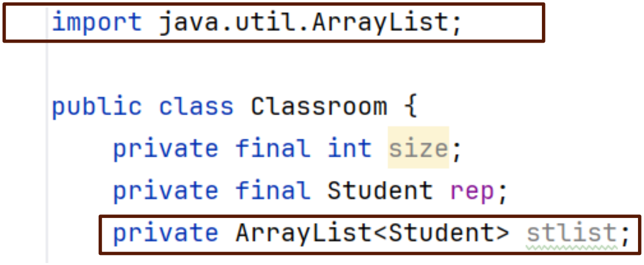
Lists may contain duplicate elements.

In addition to the operations inherited from Collection, the List interface includes operations for the following:

- ✓ Positional access
- ✓ Search
- ✓ Iteration
- ✓ Range-view

Back to our Student class (from Week 8 material)

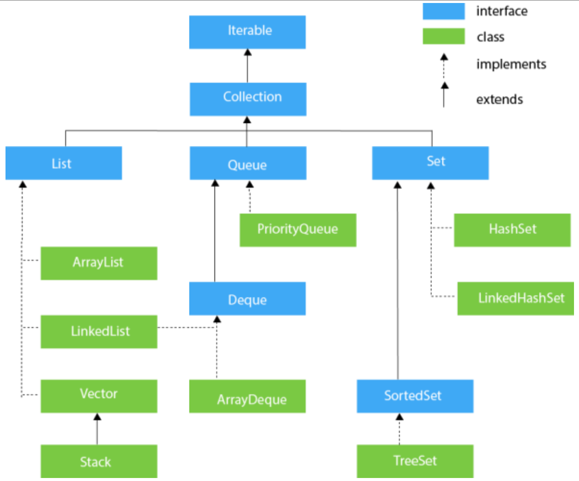
To create a list of students:



```
import java.util.ArrayList;

public class Classroom {
    private final int size;
    private final Student rep;
    private ArrayList<Student> stlist;
```

Notice the < and >

Java Generics



Generic Classes

A class might be made generic

```
public class RepositoryT <T> {...}
```

Objects of different types obtained (Integer)

```
RepositoryT <Integer> arithmeticRepository =
                new RepositoryT<>();
```

Useful when a class is instantiated with different types

ArrayList as a generic

- ArrayList class in java.util
- Describes an array of variable size (a list) of a generic type

- `public class ArrayList <T>...{...}`

- Declaration and usage (with type String)

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

Using `add()` and `get()`:

```
  myList.add("Some string");
  String s = myList.get(0);
```

ArrayList vs Array

An array in Java has a *size:*

```
Integer[] array = new Integer[100];
System.out.println("Size of an array:" +
array.length);
```

```
Size of an array:100
```

An ArrayList has a *capacity:*

```
List list = new ArrayList(100);
System.out.println("Size of the list is :" +
list.size());
```

```
Size of list is :0
```

Increasing Capacity
(Try something similar in Student class)

```
 list.add(10);
 System.out.println("Size of the list is :"
+ list.size());
```

```
Size of the list is :1
```

Arrays are fixed size. Once we initialize the array with some *int* value as its size, it can't change. The size and capacity are equal to each other.

ArrayList*'s size and capacity are not fixed.** The logical size of the list changes based on the insertion and removal of elements in it. ***This is managed separately from its physical storage size. Also when the threshold of *ArrayList* capacity is reached, it increases its capacity to make room for more elements.

Data Types for ArrayList

ArrayList internally uses an array of Object references

The following code snippet will generate a compile time error: *why?*

```
List list = new ArrayList();
list.add("hello");
String s = list.get(0);
```

Data Types for ArrayList

ArrayList internally uses an array of Object references

The following code snippet will generate a compile time error: *why?*

```
List list = new ArrayList();
list.add("hello");
String s = list.get(0);
```

Need explicit casting to tell your code what data type to expect:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Data Types for ArrayList

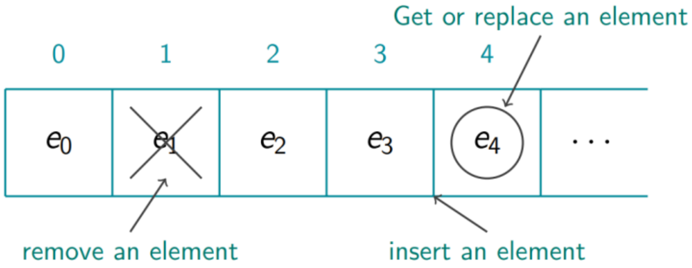
Instead of explicit casting to tell your code what data type to expect:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

We can also tell the ArrayList what data type to store using **generics <>**

```
List<String> list = new ArrayList();
list.add("hello");
String s = list.get(0);
```

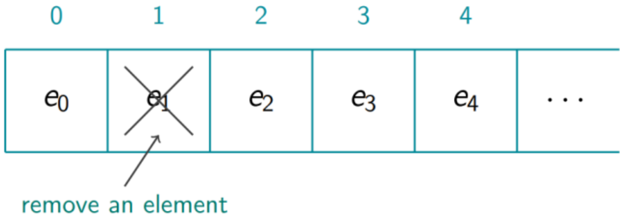
ArrayList Operations



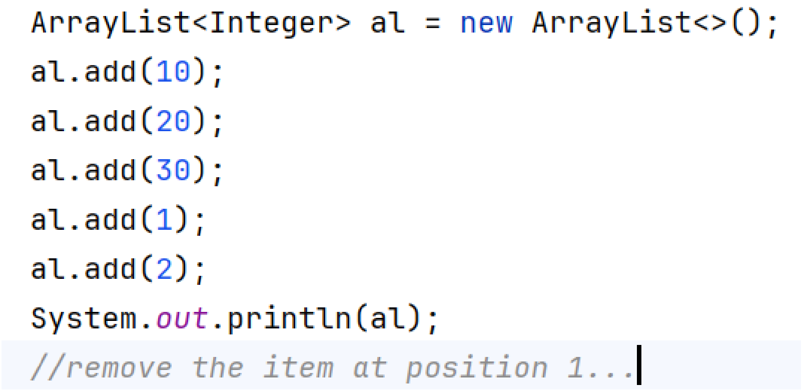
You can: add elements at any position, remove elements, replace an element in a specific position, test if the list is empty, return the number of elements in the list...

Think about memory...

What do you think should happen to elements e2-e4?

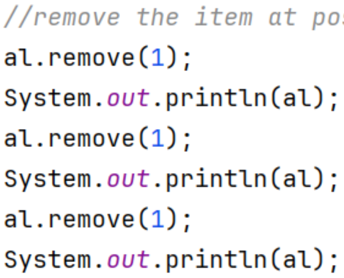


Remove by index: try removing 3 times

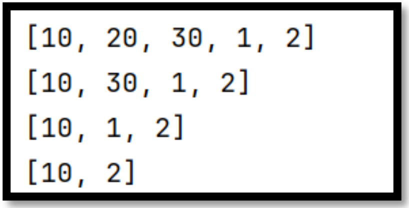


```java
ArrayList<Integer> al = new ArrayList<>();
al.add(10);
al.add(20);
al.add(30);
al.add(1);
al.add(2);
System.out.println(al);
//remove the item at position 1...
```

Remove by index: try removing 3 times



```java
//remove the item at position 1...
al.remove(1);
System.out.println(al);
al.remove(1);
System.out.println(al);
al.remove(1);
System.out.println(al);
```



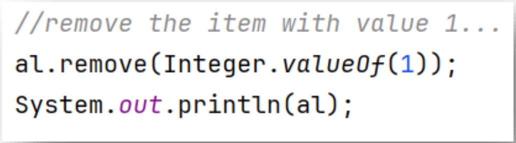
```
[10, 20, 30, 1, 2]
[10, 30, 1, 2]
[10, 1, 2]
[10, 2]
```

Removing by Value

Change your code to remove any occurrence of the value 1

Removing by Value

Change your code to remove any occurrence of the value 1



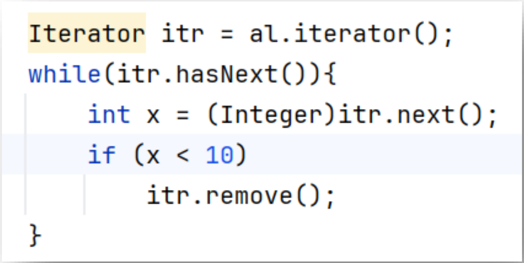
```java
//remove the item with value 1...
al.remove(Integer.valueOf(1));
System.out.println(al);
```

Removing by iteration

Try removing all items whose values are < 10

Removing by iteration

Try removing all items whose values are < 10



```
Iterator itr = al.iterator();
while(itr.hasNext()){
    int x = (Integer)itr.next();
    if (x < 10)
        itr.remove();
}
```

Lists

Many methods to manipulate lists...

- get the index of the first/last occurence of an element
- get a sublist between two indices
- sort
- shuffle
- reverse
- swap two elements
- replace of the occurrences of an element by another one...

To create: `ArrayList<Student> l = new ArrayList<Student>();`

`LinkedList<Student> l = new LinkedList<Student>();`

To add something at the end: `list1.add(thingToAdd);`

To add something in position 4: `list1.add(thingToAdd, 4);`

To concatenate two lists: `list1.addAll(list2);`

To remove (the first occurence will be removed):

`list1.remove(thingToBeRemoved);`

To remove the element in position 4: `list1.remove(4);`

To return the element in position 4: `list1.get(4);`

To modify the element in position 4:

`list1.set(4, thingToReplace);`

! You cannot create a list on primitive types (see autoboxing and unboxing)

Autoboxing and Unboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.

For example, converting:

- `int` to an `Integer`,
- `double` to a `Double`, and so on.

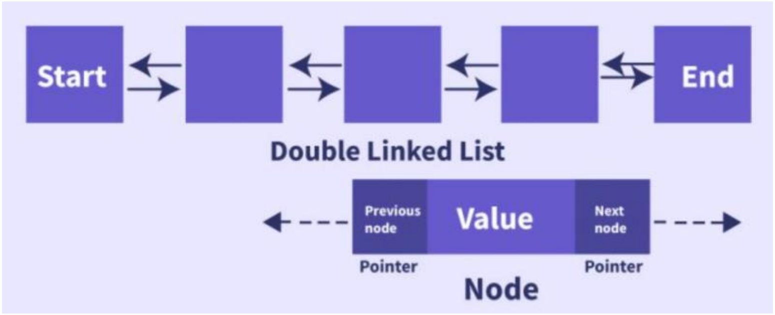
If the conversion goes the other way, this is called *unboxing*.

Two Types of LinkedList

- **Singly Linked List –** every node stores the address of only the next node in the list. We can traverse the list only in one direction(start to end).

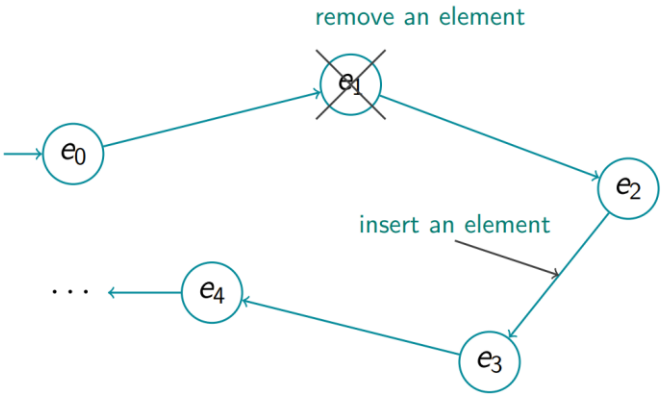
- **Doubly Linked List –** every node stores the address of the previous as well as next node in the list. We can traverse the list only in both directions(start to end and end to start).

LinkedList



Source: ArrayList vs LinkedList in Java - Scaler Topics

LinkedList Operations

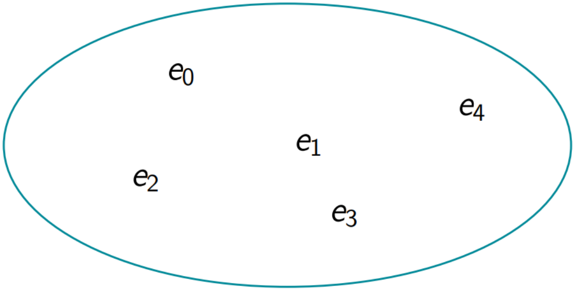


Today’s Lecture

- What is a Package in Java?
- What is an Interface in Java?
- What is a Collection in Java?
- The Java Collections Framework
  - Lists
  - **Sets**
  - Maps

Sets

$\longrightarrow$ Unordered with no repetition



Three Sets implementations:

- HashSet
- TreeSet
- LinkedHashSet

Sets: Implementation

To create: `Set<Student> set1 = new HashSet<Student>();`

`Set<Student> set1 = new TreeSet<Student>();`

`Set<Student> set1 = new LinkedHashSet<Student>();`

or: `Set<Student> l = new HashSet<Student>(set1);`

To add something: `set1.add(thingToAdd);`

To add set2 to set1: `set1.addAll(set2);`

To test if set1 contains set2: `set1.containsAll(set2);`

To only keep elements in set2 in set1: `set1.retainAll(set2);`

To remove set2 from set1: `set1.removeAll(set2);`

To iterate over the elements:

```
for (Student s :  set1) { ...  }
```

What is a Hash Table?

- When you use lists, and you are looking for a special item you normally have to iterate over the complete list. This is very expensive when you have large lists.

- A hashtable can be a lot faster, under best circumstances you will get the item you are looking for with only one access.

How does hashing work?

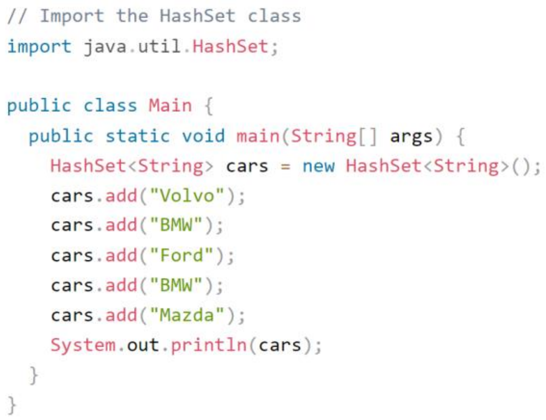
Like a dictionary ...

When you are looking for the word “classroom" in a dictionary, you are not starting with the first word under 'a’.

But rather you go straight forward to the letter ‘c'. Then to ‘cl', 'has' and so on, until you found your word.

You are using an **index** within your dictionary to **<u>speed up</u>** your search.

HashSet Example



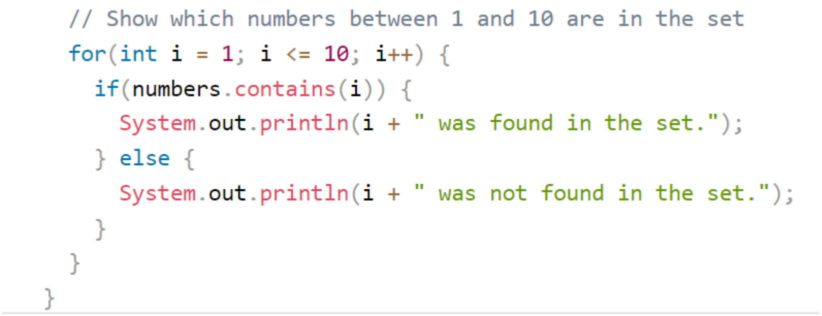
```java
// Import the HashSet class
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {
    HashSet<String> cars = new HashSet<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Exercise

```
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {

    // Create a HashSet object called numbers
    HashSet<Integer> numbers = new HashSet<Integer>();

    // Add values to the set
    numbers.add(4);
    numbers.add(7);
    numbers.add(8);
```

For all numbers between 1 and 10, display whether or not they are in the set

Answer:



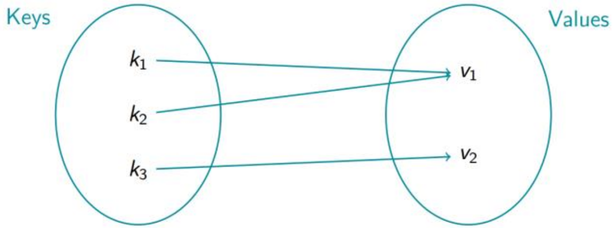
```
  // Show which numbers between 1 and 10 are in the set
  for(int i = 1; i <= 10; i++) {
    if(numbers.contains(i)) {
      System.out.println(i + " was found in the set.");
    } else {
      System.out.println(i + " was not found in the set.");
    }
  }
}
```

Today’s Lecture

- What is a Package in Java?
- What is an Interface in Java?
- What is a Collection in Java?
- The Java Collections Framework
  - Lists
  - Sets
  - **Maps**

Maps

⟶ Mathematical functions - can be viewed as a particular set of pairs (key, value)



$$\begin{tikzcd} k_1 \arrow[r] & v_1 \\ k_2 \arrow[ur] & \\ k_3 \arrow[r] & v_2 \end{tikzcd}$$

Three Maps implementations:

. HashMap

. TreeMap

. LinkedHashMap

Map Implementation

To create:

```
Map <Student, String> m = new HashMap<Student, String>();
```

```
Map <Student, String> m = new TreeMap<Student, String>();
```

```
Map <Student, String> m = new LinkedHashMap<Student, String>();
```

To add a pair (and return the old value (or null)):

```
m.put(key, value);
```

To get the value a key is mapped to: `m.get(key);`

To remove a key and its value: `m.remove(key);`

And many more...

- `containsKey()`, `containsValue()`, `size()` (number of pairs), `isEmpty()`
- `clear()`, `putAll()`

! You cannot create a set on primitive types (see autoboxing and unboxing)

Iteration over Map elements

- Get the set of keys:

```
m.keySet()
```

Iterate over the set of keys:

```
for (Student s :  m.keySet()) { ...  }
```

- Get the set of values:

```
m.values()
```

Iterate over the set of values:

```
for (String s :  m.values()) { ...  }
```

- Get the set of pairs:

```
m.entrySet()
```

To iterate over the pairs and get the key and value in each pair:

```
for (Map.Entry<Student, String> e :  m.entrySet()) {
```

```
System.out.println(e.getKey() + ":" + e.getValue()}
```

Today's Lecture

- What is a Package in Java?
- What is an Interface in Java?
- What is a Collection in Java?
- The Java Collections Framework
  - Lists
  - Sets
  - Maps

Thank you!

Reminder - Module Evaluation Survey

- A formal way to gather your feedback to improve the student experience
- More information on the Student Hub
- Access all open surveys on the Student Survey Portal or via the QR code (you will also be emailed a link)