Advanced Java

LESSON 4: POLYMORPHISM, GENERICS & COLLECTIONS

Creative Commons

Attribution 4.0 International (CC BY 4.0)

This workforce product was funded by a grant awarded by the U.S. Department of Labor's Employment and Training Administration. The product was created by the grantee and does not necessarily reflect the official position of the U.S. Department of Labor. The U.S. Department of Labor makes no guarantees, warranties, or assurances of any kind, express or implied, with respect to such information, including any information on linked sites and including, but not limited to, accuracy of the information or its completeness, timeliness, usefulness, adequacy, continued availability, or ownership. This is an equal opportunity program. Assistive technologies are available upon request and include Voice/TTY (771 or 800-947-6644).

Agenda

- Statics
- Interfaces
- Polymorphism
- Generics
- Collections

Statics

Statics

Static Fields

Static fields and *static methods* do not belong to a single instance of a class.

To invoke a static method, the class name, rather than the instance name, is used.

```
double val = Math.sqrt(25.0);
```

To use a static field, the class name or an object name may be used.

Statics

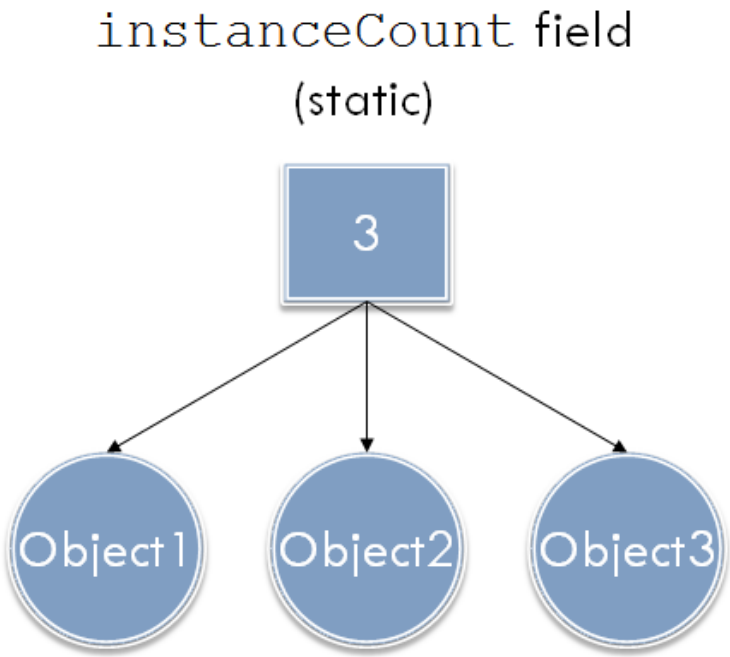
Static Fields

Class fields are declared using the `static` keyword between the access specifier and the field type.

```
private static int
        instanceCount = 0;
```

The field is initialized to 0 only once, regardless of the number of times the class is instantiated.

- Primitive static fields are initialized to 0 if no initialization is performed.



Statics

Static Methods

Methods can also be declared static by placing the `static` keyword between the access modifier and the return type of the method.

```
public static double
 milesToKilometers(double miles)
{…}
```

When a class contains a static method, it is not necessary to create an instance of the class in order to use the method.

```
double kilosPerMile =
 Metric.milesToKilometers(1.0);
```

Statics

Static Fields

```
Counter obj1 = new Counter();
Counter obj2 = new Counter();
System.out.println(Counter.iCount);
Counter obj3 = new Counter();
Counter obj4 = new Counter();
System.out.println(obj1.getICount());
System.out.println(obj4.getICount());
```

```
2
4
4
```

```
package com.staticcount;
public class Counter {
  public static int iCount = 0;
  public Counter() {
      iCount++;
  }
  public int getICount() {
      return iCount;
  }
}
```

Statics

Static Methods

```
package com.staticcount;
    public class Startup {
        public static void main(String[] args) {
        System.out.println(Converter.kToM(12.5));
        System.out.println(Converter.mToK(235));
    }
}
```

```
System.out.println(Converter.kToM(12.5));
System.out.println(Converter.mToK(235));
```

```
7.768800497203232
378.115
```

In-Class Activity

1. Create a “Person” class with 3 instance properties: firstName (a String), lastName (a String) and age (an int). In addition to a default constructor, how may additional, overloaded constructors can you think of to create? Create them.
2. Class properties (declared static) are shared by all object instances created from that class. Can you think of a Person property that should be shared? Add that to your Person class. Then write a method that retrieves that data. How should this method be written?
3. From a separate “Startup” class, create several Person objects and store them in an array. Now write a for-loop to iterate over that array and output the full name and age of each of the Person objects stored in that array.

Interfaces

Interfaces

Abstract Classes

- Cannot be instantiated.
- Only serves as a superclass for other classes.

```
public abstract class ClassName
```

Abstract Methods

- Does not have a body and must be overridden.
- Not all Methods in an Abstract Class have to be abstract.

```
AccessSpecifier abstract ReturnType
MethodName(ParameterList);
```

Interfaces

An *interface* is similar to an abstract class with all abstract methods.

- They cannot be instantiated

Interfaces specify the behavior of a group of classes.

An *interface* is similar to a class, except:

- the keyword `interface` is used instead of the keyword `class`, and
- the methods headers are listed, terminated by semicolons.

```
public interface InterfaceName {
   (Method headers...)
}
```

Interfaces

Implementing Interfaces

- Classes may implement one or more interfaces.
- Classes use the implements keyword in the class header.
- Classes may can only extend one other class but may implement multiple Interfaces.

```
public class ClassName implements InterfaceName {
   (Properties and Methods...)
}
```

```
public class ClassName extends SuperClassName
      implements InterfaceName1, InterfaceName2 {
   (Properties and Methods...)
}
```

Interfaces

```
public class FinalExam extends
GradedActivity implements Relatable {
    . . .
    public double getPointsEach() {
        return pointsEach;
    }

    public int getNumMissed() {
        return numMissed;
    }
```

Interfaces

Interface Properties

- Properties in Interfaces behave as static and final but do not use those keywords.
- Since there are no method bodies or constructors in Interfaces, the value must be assigned when the variable is declared.
- All Classes that implement the Interface have access to the constants.

```
public interface InterfaceName {
   (type identifier = value...)
   (Method headers...)
}
```

In-Class Activity

Using the Person Class form the previous Activity, create a Weighable Interface.

The Interface should define the following methods:

- **`addWeight(double pounds)`**
- **`loseWeight(double pounds)`**

Modify the Person Class to implement the Class.

Polymorphism

Polymorphism

Polymorphism

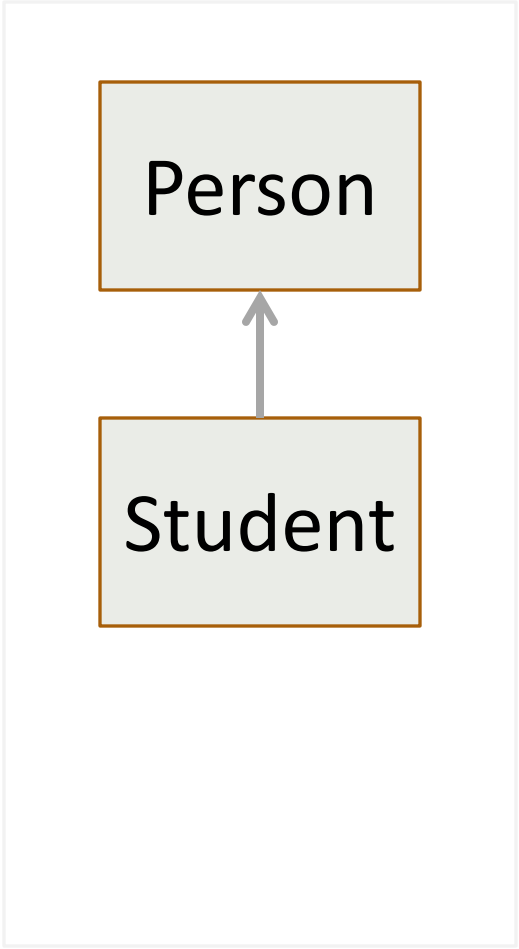
According to the Polymorphism pattern, responsibility of defining the variation of behaviors based on type is assigned to the types for which this variation happens. This is achieved using polymorphic operations.

Polymorphism

A reference variable may reference objects of classes that are derived from the variable's class.

```
Person bob = new Student();
```

- An object of the **`Student`** class is a **`Person`** object.



Polymorphism

Dynamic Binding or Late Binding

- The case where compiler is not able to resolve the call at compile time and the binding is done at runtime.
- Instance method calls are always resolved at runtime, but all the static method calls are resolved at compile time itself and hence we have static binding for static method calls.
- Because static methods are class methods and hence they can be accessed using the class name itself (in fact they are encouraged to be used using their corresponding class names only and not by using the object references) and therefore access to them is required to be resolved during compile time only using the compile time type information.

Polymorphism

Dynamic Binding

- If the object of the subclass has overridden a method in the superclass:
- If the variable makes a call to that method the subclass's version of the method will be run.

```
GradedActivity exam = new PassFailActivity(60);
exam.setScore(70);
System.out.println(exam.getGrade());
```

- Java performs dynamic binding or late binding when a variable contains a polymorphic reference.
- The Java Virtual Machine determines at runtime which method to call, depending on the type of object that the variable references

Polymorphism

Interfaces

- An “is A” Relationship does not just refer to a parent class, it may also refer to Interfaces.
- This is very useful for:
  - Method Calls
  - Arrays
- This is one of the primary reasons to use Interfaces

Polymorphism

Interfaces

```
package com.day3;
public interface Addable {
    public int sumThem(int a, int b);
    public double sumThem(double a, double b);
}
```

```
package com.day3;
public class Type1 implements Addable {
    private int num;
    public int getNum() {
        return num;
    }
    public void setNum(int num) {
        this.num = num;
    }
    @Override
    public int sumThem(int a, int b) {
        return a+b;
    }
    @Override
    public double sumThem(double a, double b) {
        return a+b;
    }
}
```

Polymorphism

Method Call

```
package com.day3;
public class RunMain {
    public static void main(String[] args) {
        Type1 o1 = new Type1();
        Type2 o2 = new Type2();
        doSomething(o1,o2);
    }
    public static void doSomething(Addable a, Addable b) {
        System.out.println(a.sumThem(5, 5));
        System.out.println(b.sumThem(5, 5));
    }
}
```

```
10
50
```

Polymorphism

Array

```
package com.day3;
public class RunMain {
   Addable[] addArray = new Addable[3];
   addArray[0] = new Type1();
   addArray[1] = new Type2();

   System.out.println(addArray[0].sumThem(5, 5));
   System.out.println(addArray[1].sumThem(5, 5));
}
```

```
10
50
```

Polymorphism

Casting

```
package com.day3;
public class RunMain {
    public static void main(String[] args) {
        Addable[] addArray = new Addable[3];
        addArray[0] = new Type1();
        addArray[1] = new Type2();
    }
    public static void setTheprop(Addable a) {
        if (a instanceof Type1) {
            Type1 o1 = (Type1)a;
            o1.setNum(5);
        } else
        if (a instanceof Type2) {
            Type2 o1 = (Type2)a;
            o1.setName("Bob");
        }
    }
}
```

Generics

Generics

Dynamic Binding

- Added to the Java programming language in 2004 as part of J2SE 5.0.
- Generics are used for type safety and avoiding dangerous casts.
- Generics allow the compiler to identify and prevent type errors that would have happened at runtime using casting.

Generics

```
package com.day3;

public class SomeThing<T> {
    private T t;
    public void add(T t) {
        this.t=t;
    }
    public T get() {
        return t;
    }
    public void printInfo() {
       System.out.println("This is " + t.getClass());
    }
}
```

Generics

```
package com.day3;
public class RunMain {
    public static void main(String[] args) {
        SomeThing<Type1> o1 = new SomeThing<Type1>();
        SomeThing<Type2> o2 = new SomeThing<Type2>();
        o1.add(new Type1());
        o2.add(new Type2());
        o1.printInfo();
        o2.printInfo();
    }
}
```

```
This is class com.day3.Type1
This is class com.day3.Type2
```

Collections

Collections

What is a Collection?

- A collection — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

Collections

Types of Collections

General-purpose Implementations					
Interfaces	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Collections

ArrayList

- Similar to an array that stores objects
- Different then an array:
  - May change in size
    - Automatically expands when a new item is added
    - Automatically shrinks when items are removed
- Must import the following package

```
import java.util.ArrayList;
```

Collections

ArrayList

- Creating

```
ArrayList carList = new ArrayList();
```

- Populating

```
carList.add("Mustang");
carList.add("Saab");
```

- Accessing the number of Elements

```
carList.size();
```

- Accessing the contents of an Element

```
carList.get(1);
```

Collections

ArrayList

- toString()

```
System.out.println(carList);
```

- Removing

```
carList.remove(0);
```

- Inserting at a specific location (otherwise they are added to the end)

```
carList.add(0, "Horizon");
```

- Replace one item with another

```
carList.set(1, "Caravan");
```

Collections

ArrayList

```
ArrayList carList = new ArrayList();
carList.add("Mustang");
carList.add("Saab");
System.out.println(carList.size());
System.out.println(carList.get(0));
System.out.println(carList);
carList.remove(0);
carList.add(0, "Horizon");
System.out.println(carList);
carList.set(1, "Caravan");
System.out.println(carList);
```

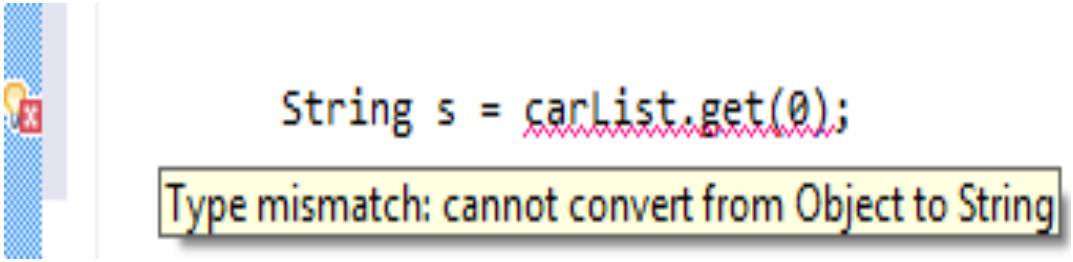
```
2
Mustang
[Mustang, Saab]
[Horizon, Saab]
[Horizon, Caravan]
```

Collections

ArrayList

- ArrayList elements are of type Object
- If A Generic is not used, they need to be casted to be used

```
String s = (String)carList.get(0);
```



Collections

ArrayList - Generics

```
ArrayList<String> stuff = new ArrayList();
stuff.add("First");
stuff.add("Second");
stuff.add("Third");
```

Collections

ArrayList – Generics – For Loops

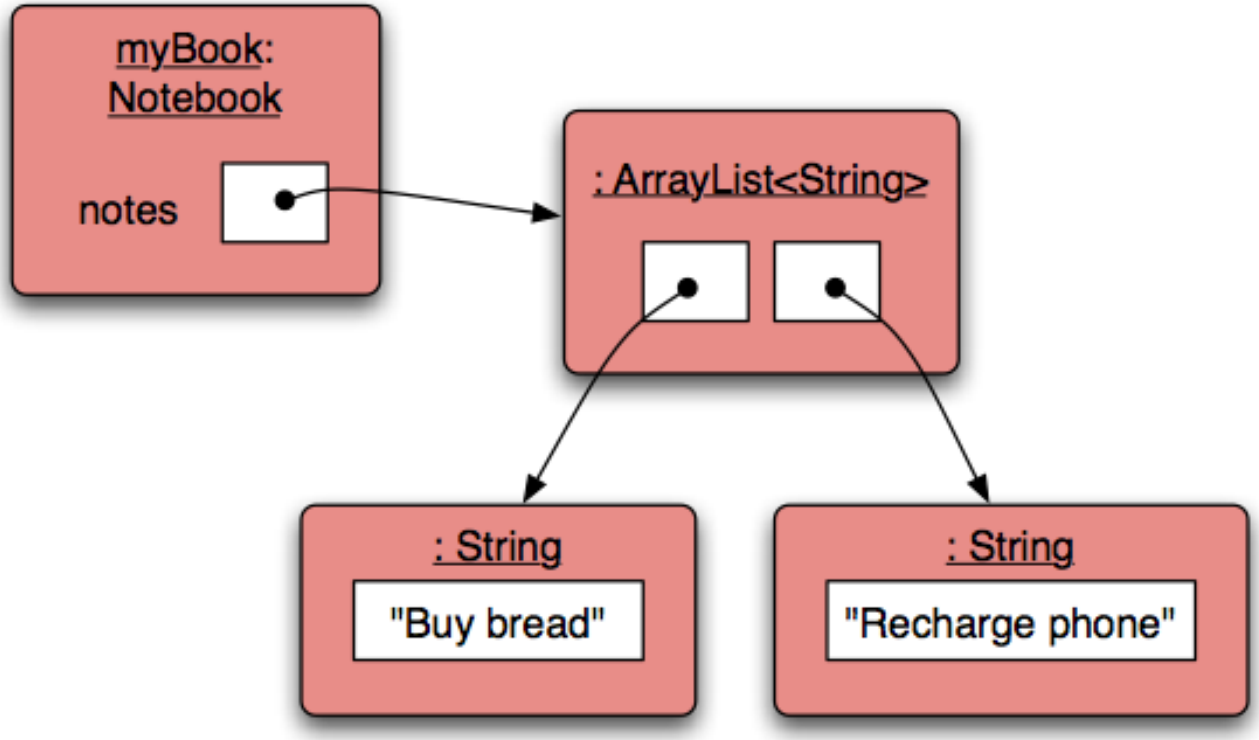
```
ArrayList<String> stuff = new ArrayList();
stuff.add("First");
stuff.add("Second");
stuff.add("Third");

for (String s : stuff) {
     System.out.println(s);
}
```

```
First
Second
Third
```

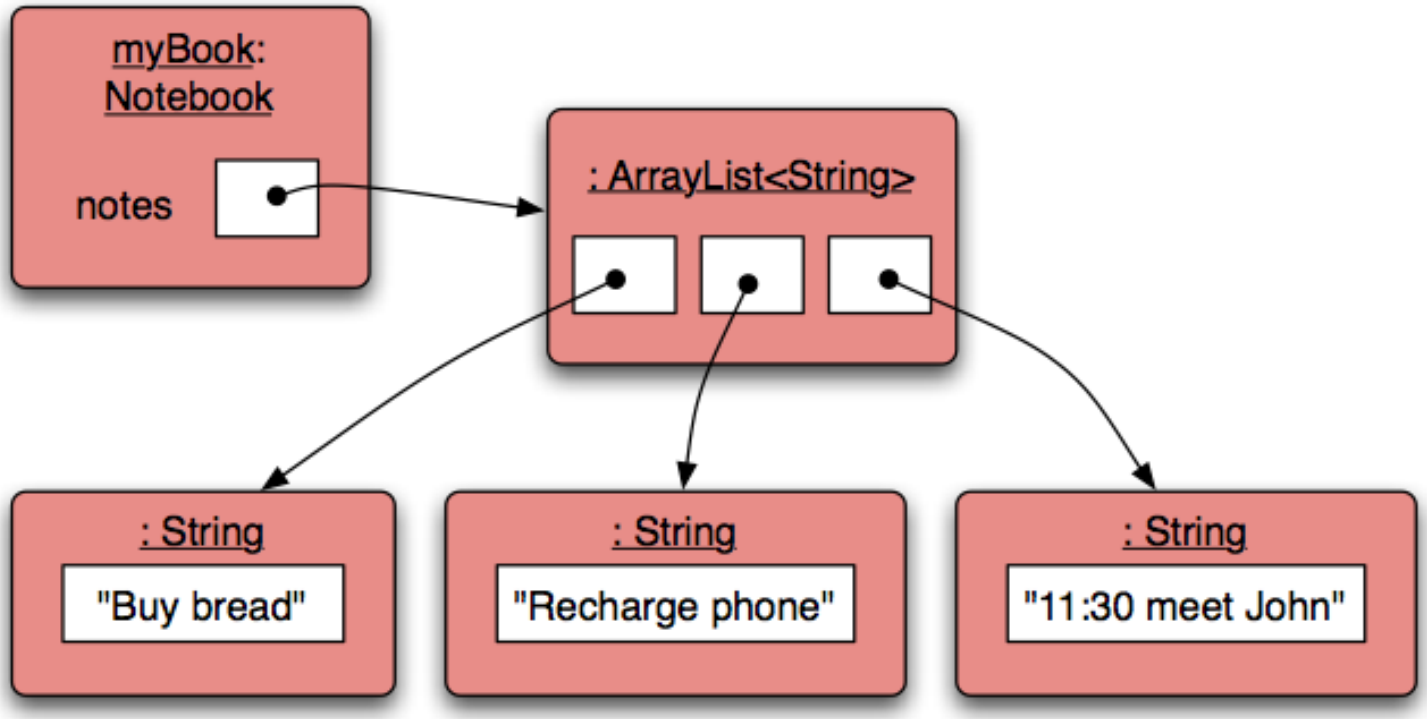
Collections

ArrayList



Collections

ArrayList – Adding an Element



Collections

Iterators

- Cycles through the elements in a collection.
- An object that implements either the Iterator or the ListIterator interface.
- Enables the program to cycle through a collection, obtaining or removing elements.
- ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
  - Insatiate an iterator to the start of the collection by calling the collection's iterator( ) method.
  - Set up a loop that makes a call to hasNext( ).
  - Have the loop iterate as long as hasNext( ) returns true.
  - Within the loop, obtain each element by calling next( ).

```
import java.util.Iterator;
import java.util.ListIterator;
```

Collections

Iterators

```
ArrayList<String> stuff = new ArrayList();
stuff.add("First");
stuff.add("Second");
stuff.add("Third");

Iterator it = stuff.iterator();

while (it.hasNext()) {
     System.out.println(it.next());
}
```

```
First
Second
Third
```

Collections

List Iterators

```
ArrayList<String> stuff = new ArrayList();
stuff.add("First");
stuff.add("Second");
stuff.add("Third");

ListIterator it = stuff.listIterator();
while(it.hasNext()) {
    Object element = it.next();
    it.set(element + "+");
}
```

First+
Second+
Third+

Collections

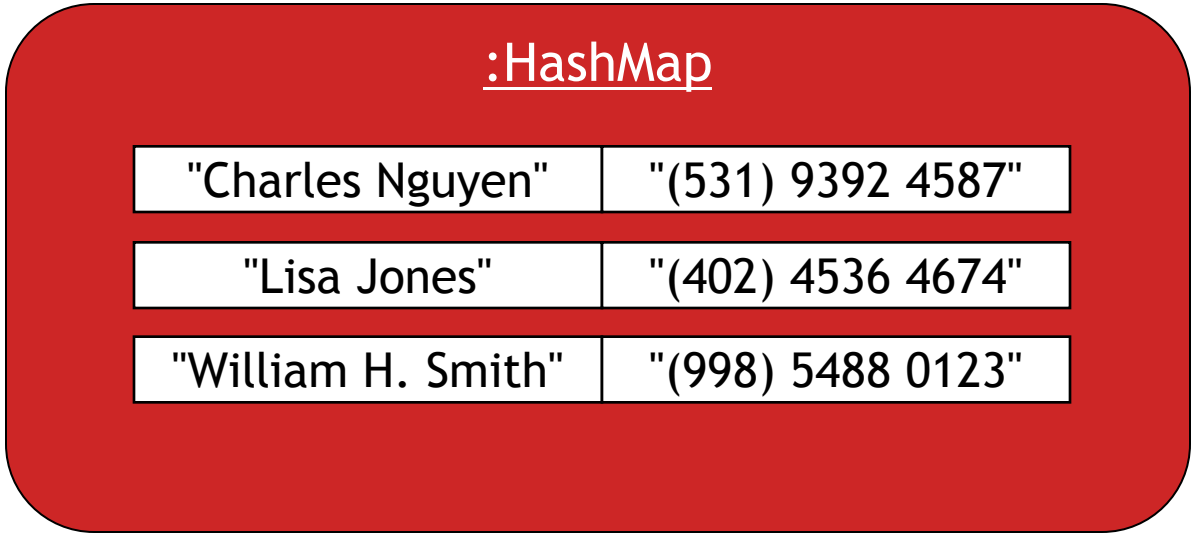
What is a Set?

- A collection that contains no duplicate elements.
- Contains no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction.

Collections

What is a Map?

- Collections that contain pairs of values.
- Pairs consist of a key and a value.
- Lookup works by supplying a key, and retrieving a value.



Collections

What is a Hash Table?

- A hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code.

Collections

HashMap

- Uses a hash table to implement the Map interface. This allows the execution time of basic operations, such as get( ) and put( ), to remain constant even for large sets.

Collections

HashMap

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
```

```
HashMap hm = new HashMap();
hm.put("Mustang", new Double(2345.23));
hm.put("Horizon", new Double(1500.00));
hm.put("Saab", new Double(1800.00));

Set set = hm.entrySet();
Iterator i = set.iterator();
while(i.hasNext()) {
     Map.Entry me = (Map.Entry)i.next();
     System.out.print(me.getKey() + ": ");
     System.out.println(me.getValue());
}
```

```
Horizon: 1500.0
Saab: 1800.0
Mustang: 2345.23
```

Collections

HashSet

- Creates a collection that uses a hash table for storage.
- The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

Collections

HashSet

```
import java.util.HashSet;
```

```
HashSet hs = new HashSet();
hs.add("B");
hs.add("A");
hs.add("D");
hs.add("E");
hs.add("F");
hs.remove("D");
System.out.println(hs);
```

[E, F, A, B]

Collections

TreeSet

- Provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

Collections

TreeSet

```
import java.util.TreeSet;
```

```
TreeSet hs = new TreeSet();
hs.add("B");
hs.add("A");
hs.add("D");
hs.add("E");
hs.add("F");
hs.remove("D");
System.out.println(hs);
```

[E, F, A, B]

Collections

TreeMap

- Implements the Map interface by using a tree. A TreeMap provides an efficient means of storing key/value pairs in sorted order, and allows rapid retrieval.
- Note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.

Collections

TreeMap

```
import java.util.TreeMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
```

```
TreeMap hm = new TreeMap();
hm.put("Mustang", new Double(2345.23));
hm.put("Horizon", new Double(1500.00));
hm.put("Saab", new Double(1800.00));

Set set = hm.entrySet();
Iterator i = set.iterator();
while(i.hasNext()) {
     Map.Entry me = (Map.Entry)i.next();
     System.out.print(me.getKey() + ": ");
     System.out.println(me.getValue());
}
```

```
Horizon: 1500.0
Saab: 1800.0
Mustang: 2345.23
```

In-Class Activity

1. Download “Collection Work” from Blackboard and open in IntelliJ.
2. List the top most used words and how many times they were used.
3. List the number of words that are only used once.