12 HASH SETS

Terry Marris 28 April 2001

12.1 OBJECTIVES

By the end of this lesson the student should be able to

- understand what a set is
- use the *HashSet* operations
- use the *Iterator* operations

12.2 PRE-REQUISITES

The student should be comfortable with using *String* class methods, messages, parameters, methods, selections and iterations.

12.3 PREVIEW

We look at the properties of a set. We review the operations provided by the *HashSet* and *Iterator* classes. We use the *HashSet* and *Iterator* methods to solve simple problems involving sets.

12.4 COLLECTIONS

Objects are often arranged in collections. You might have a collection of student objects (a cohort) or a collection of book objects (a library) or a collection of buses (a fleet). A collection is itself an object.

You would expect a collection class to have methods that add a new object to a collection, find an object in a collection and remove an object from a collection. You would expect the collection to know how many objects it contains.

12.5 SETS

A set is perhaps the simplest and most fundamental of all collection types. A set is a collection of objects with

- no duplicates
- no order

So, for example,

{ tom, dee, hari } and *{ hari, tom, dee }* represent identical sets because order does not matter.

{ tom, dee, hari } and *{ tom, dee, hari, hari }* represent identical sets because repetitions are ignored.

The objects contained in a set are known as its elements. The number of elements a set contains is known as its cardinality.

12.6 THE HASH SET CLASS

The standard Java library provides a *HashSet* class. A *HashSet* is an implementation of *Set*. It has operations to *add* an object to a set, *remove* an object from a set, determine whether a set *contains* a given object and report the *size* of a set i.e. the number of elements it contains.

java.util.HashSet

Constructors	
HashSet()	initialises a new, empty set.
HashSet(Collection c)	initialises a new set with the given collection.

Methods		
boolean	add(Object obj)	adds the given object to this set and returns true but only if the given object is not already present in the set.
void	clear()	removes all objects from this set.
Object	clone()	returns a duplicate copy of this set.
boolean	contains(Object obj)	returns true if this set contains the given object.
boolean	isEmpty()	returns true if this set contains no elements.
Iterator	iterator()	returns an iterator over the element in this set. The elements are returned in no particular order.
boolean	remove(Object obj)	removes the given object from this set and returns true but only if this set contains the given object.
int	size()	returns the number of elements in this set (i.e. its cardinality).

An iterator visits elements in a container one by one. We shall discuss iterators in more detail in §12.9 below.

12.7 USING THE HASH SET METHODS

The standard Java library contains packages of useful classes. One package, named *java.util*, contains the *HashSet* class. We use the *import* statement to make the *HashSet* (and all the classes in the *java.util* package) available for us to use.

```
import java.util.*;
```

We create a new empty set named *aSet* with a call to the *HashSet* constructor.

```
Set aSet = new HashSet();
```

We place a string object, *"tom"* in the set with an *add("tom")* message.

```
aSet.add("tom")
```

Since the *add(Object)* method returns *true* if the given object was not already in the set, we can display the value returned by using the message as a parameter to *System.out.println()*.

```
System.out.println(aSet.add("tom"));
```

We expect *true* to be displayed because *"tom"* had not previously been added to the set.

If we tried to add *"tom"* for a second time we would expect *false* to be displayed.

We could see whether the set contains *"xray"* by sending the message *contains("xray")* to *aSet*. Since *contains(Object)* returns a *boolean*, we can display the value returned.

```
System.out.println(aSet.contains("xray"));
```

We expect *false* to be displayed because *"xray"* has not been added to *aSet*.

The *clone()* method allows us to make a duplicate copy of a set. First, we create a *HashSet* object named *bSet* and then make *bSet* a copy of *aSet*. The *clone()* method returns an object of no specific kind. So we use the cast operator, *(HashSet)*, to convert the non-specific object in to a *HashSet* object.

```
Set bSet = new HashSet();
bSet = (HashSet)aSet.clone();
```

The complete program, and its run, is shown below.

```
/* TestHashSet.java
   Terry Marris  28 April 2001
*/

import java.util.*;

public class TestHashSet {
  public static void main(String[] s)
  {
    Set aSet = new HashSet();
    System.out.println("Adding three names to a set ...");
    System.out.println(aSet.add("tom"));
    System.out.println(aSet.add("dee"));
    System.out.println(aSet.add("hari"));

    System.out.println("Adding a duplicate name ...");
    System.out.println(aSet.add("dee"));

    System.out.println("Checking whether the set contains " +
                       "tom, hari and xray ... ");
    System.out.println(aSet.contains("tom"));
    System.out.println(aSet.contains("hari"));
    System.out.println(aSet.contains("xray"));

    System.out.println("Removing dee ...");
    System.out.println(aSet.remove("dee"));

    System.out.println("Checking clone() and size() ...");
    Set bSet = new HashSet();
    bSet = (HashSet)aSet.clone();
    System.out.println(bSet.size());

    System.out.println("Checking clear() and isEmpty() ...");
    bSet.clear();
    System.out.println(bSet.isEmpty());

    System.out.println("Printing out the contents of " +
                       "aSet ...");
    System.out.println(aSet);
  }
}
```

Output

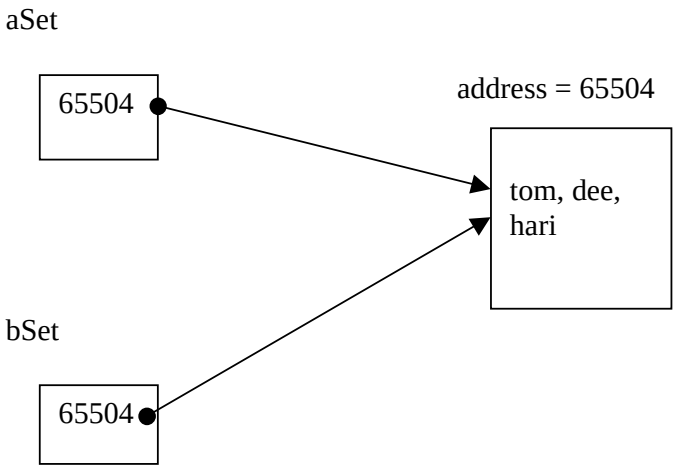
```
Adding three names to a set ...
true
true
true
Adding a duplicate name ...
false
Checking whether the set contains tom, hari and xray ...
true
true
false
Removing dee ...
true
Checking clone() and size() ...
2
Checking clear() and isEmpty() ...
true
Printing out the contents of aSet ...
[tom, hari]
```

Exercise. Identify the line (or lines) of the program that was responsible for each line of the output.

12.8 ASSIGNMENT AND CLONING

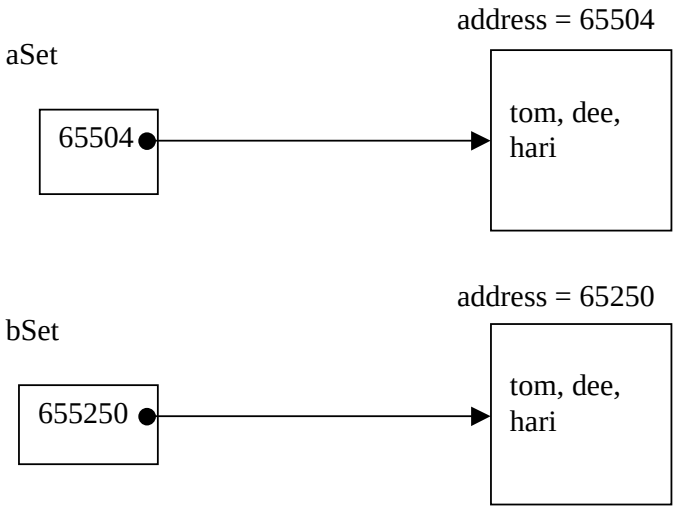
Notice the difference between assignment and cloning.

After the assignment *bSet = aSet*, both refer to the same object in memory.



So whatever changes are made to *bSet* are also automatically made to *aSet.*

After cloning, *bSet = aSet.clone()*, both sets refer to different objects in memory.



So changes to *aSet* and *bSet* can be made independently of each other.

In cloning a set, the contents of the set themselves are not cloned.

Incidentally, you could make a new, independent copy of *aSet* by using it as a parameter to construct *bSet.*

```
Set bSet = new HashSet(aSet);
```

12.9 THE ITERATOR CLASS

We use an iterator to look at each element in a set in turn.

An Iterator object has three methods.

```
Object next()
boolean hasNext()
void remove()
```

By repeatedly calling the *next()* method we can visit the elements of a set one by one. *hasNext()* returns *true* if the iterator object still has more elements to visit. It would be an error if a call to *next()* was made and there was no next element to visit. So each call to *next()* should be guarded by a call to *hasNext()*.

If *aSet* contains *tom*, *dee* and *hari* the following code fragment will display

{ tom, dee, hari }

```
System.out.println("{ ");        print the opening brace
Iterator it = aSet.iterator();  ask aSet to return its iterator
while (it.hasNext()) {          loop for as long as there is another element to visit
 String s = (String)it.next();  get the next element from aSet
 System.out.println(s);         and print it.
 if (it.hasNext())              if it is not the last element,
  System.out.println(", ");     print a comma
}
System.out.println(" }");       print the closing brace
```

The Iterator class is summarised in the table below.

Class java.util.Iterator

Methods		
boolean	hasNext()	returns true if the iteration has more elements.
Object	next()	returns the next element in the iteration.
void	remove()	removes the last element returned by next(). Each call to remove(0 must be preceded by a call to next(). Removing an element by any other method usually corrupts the iterator.

12.10 EXAMPLES

We are given the sets

littleVillainSet = { tom, sam, dee, pat }

bigVillainSet = { pam, vim, may, sue }

cleverVillainSet = { sam, sue pat, vim }

To create and print out the *littleVillainSet* we write

```
Set littleVillainSet = new HashSet();
littleVillainSet.add("tom");
littleVillainSet.add("sam");
littleVillainSet.add("dee");
littleVillainSet.add("pat");
System.out.println("Little villains: " +
                   littleVillainSet);
```

We expect the output to be something like:

```
Little villains: [tom, pat, dee, sam]
```

(a) Some villain has committed a crime. We want to know who all the known villains are.

We create the set containing all little villains.

```
Set all = new HashSet(littleVillainSet);
```

We add all the big villains to it.

```
Iterator it = bigVillainSet.iterator();
while (it.hasNext()) {
  String aVillain = (String)it.next();
  all.add(aVillain);
}
```

We then add all the clever villains to it.

```
it = cleverVillainSet.iterator();
while (it.hasNext()) {
  String aVillain = (String)it.next();
  all.add(aVillain);
}
System.out.println("All villains: " + all);
```

We rely on the *add(Object)* method to not add duplicates.

We expect the output to be something like:

```
All villains: [tom, vim, sue, pat, dee, sam, may, pam]
```

(b) We establish that the villain (or villains) we want is (are) both little and clever.

```
Set littleAndClever = new HashSet();
```

We look at each element in little villain set. If a little villain is also in the clever villain set then it must be both little and clever.

```
it = littleVillainSet.iterator();
while (it.hasNext()) {
  String aVillain = (String)it.next();
  if (cleverVillainSet.contains(aVillain))
    littleAndClever.add(aVillain);
}
System.out.println("Little and clever: " +
                   littleAndClever);
```

We expect the output to be something like

```
Little and clever: [pat, sam]
```

(c) We now eliminate *pat* from our enquiries, that is, we eliminate *pat* from our little and clever set.

We create the set *eliminated* containing just *pat*. We also create the set *suspect* which, initially, contains all the little and clever villains.

```
Set eliminated = new HashSet();
eliminated.add("pat");
Set suspect = new HashSet(littleAndClever);
```

We look at each element in *suspect* in turn. If a villain from *suspect* is also in *eliminated*, we remove it from *suspect*.

```
it = suspect.iterator();
while (it.hasNext()) {
  String aVillain = (String)it.next();
  if (eliminated.contains(aVillain))
    it.remove();
}
System.out.println("Sus: " + suspect);
```

We expect the output to be

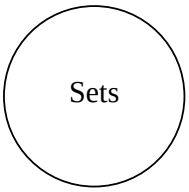
```
Sus: [sam]
```

We can use sets to model a surprisingly large range of data processing systems.

12.11 FURTHER READING

HORSTMANN & CORNELL *Core Java 2 Volume 1* pp 137
HORSTMANN & CORNELL *Core Java 2 Volume 2* pp 77, 93
NORCLIFFE & SLATER Mathematics of Software Construction pp24..41
MARRIS *Simple Z* pp1..25

12.12 REVIEW



a collection of objects (elements) of the same type

no duplicates, no order

implemented by HashSet with operations

add(Object):boolean

if the given object is not in this set,
adds the given object to this set & returns true

contains(Object):boolean

returns true if the given object is in this set

clone():Object

returns a duplicate copy of this set

size():int

returns the number of elements in this set

iterator():Iterator

returns an iterator over this set
to look at each element in turn

hasNext():boolean

returns true if this iterator has another object to look at

next():Object

returns the next object to be looked at (but only if there is one not yet looked at)

remove():void

removes the object just returned by next()

12.13 EXERCISES

1 Explain, with the aid of illustrative examples, the meaning of each of the following terms:

(a) set **(b)** cardinality **(c)** a HashSet **(d)** an Iterator

2 Write and test a program that performs the following tasks:

(a) creates the sets of employees

programmer = { bigJohn, littleJohn, may, pat, alice }

analyst = { tom, denise, may }

projectLeader = { alice, littleJohn }

(b) uses HashSet and Iterator methods to derive, from the sets given above,

(i) employee = { bigJohn, littleJohn, may, pat, alice, tom, denise, alice }

(ii) employees who are both programmers and analysts

(iii) employees who are programmers and not project leaders

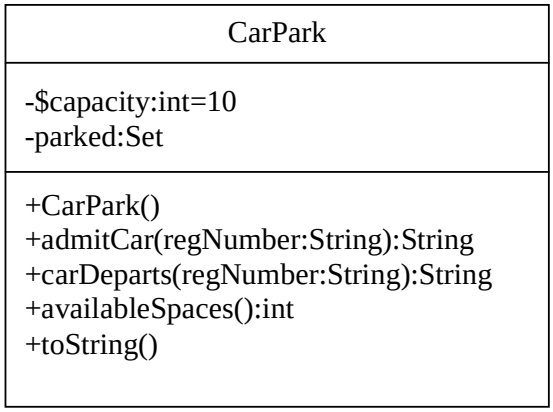
(iv) employees who are project leaders and not programmers

(v) employees who are programmers but not including *promoted = { may, alice }*

(vi) employees who are project leaders and including *promoted = { may, alice }*

3 Write a small test program to determine whether *null* is an element of a set.

4 A college has a car park. Unfortunately, the car park is too small for all its employees and students. Write and test the class *CarPark* specified below.



capacity is the number of cars a *CarPark* can contain and, therefore, the maximum cardinality of the set *parked.*

parked contains the registration numbers of cars currently in the car park.

When a car arrives, it is admitted to the car park provided the capacity of the car park has not been reached. It is an error for a car to be added if it is already in the park. *admitCar(String):String* returns "success" and adds the given car registration number to the set parked if the size of parked has not reached capacity, otherwise it returns failure.

When a car leaves the car park, it is no longer parked there. It is an error for a car to be removed if it was not there in the first place. *carDeparts(String):String* returns *"success"* and removes the given car registration number from the set *parked* if it is in the set, otherwise it returns *"failure - car not in park"*.

admitCar(String) adds the given car (represented by its registration number) to the set *parked* and returns *"success"* provided *parked* does not already contain a car with the given registration number, otherwise it returns *"failure - car already in park"*.

availableSpaces():int returns the number of cars parked, that is, the number of car registration numbers stored in the set *parked.*

toString():String returns a string representing all the cars currently in the car park, that is, the contents of the set *parked*.