

Chapter 9 : Holding the objects

To solve the general programming problem, you need to be able to create **any number** of objects, **anytime**, **anywhere**.

Java has several ways to hold objects : **array** & **container classes**.

9.1 Arrays

Features:

1. The array is **the most efficient** way that Java provides to store and **randomly** access a sequence of objects. ---- Its **size is fixed** and cannot be changed.

2. When you create an array, you create it to **hold a specific type**.

For efficiency and type checking it's always worth **trying to use an array if you can**.

9.1.1 Arrays are first-class objects

The array identifier is actually **a reference** to a true object that's created on the heap. ---- **the object** that holds the references to other objects.

Arrays of **objects** and arrays of **primitives** are almost **identical** in their use.

Sample : **ArraySize.java**

1. You can't find out how many elements are **actually** in the array ---- **length** tells you only how many elements **can** be placed in the array.
2. You can see whether a particular array slot has an object in it by checking whether it's **null**.

Dynamic aggregate initialization



```
hide(new Weeble[] { new Weeble(), new Weeble() });
```

Note : **Container** classes can hold only references to **objects**. It is possible to use the “**wrapper**” classes to place primitive values inside a container.

However, it's much more **efficient** to create and access an **array of primitives** than a **container of wrapped** primitives.

9.1.2 Returning an array

Actually, you're returning a **reference** to an array . And it will be around **as long as you need it**.

Sample : **IceCream.java** *

9.2 The Arrays class

`java.util.Arrays`

1. **equals()** ---- compare two arrays for equality.

2. **fill()** ---- fill an array with a value.

3. **sort()** ---- sort the array.

4. **binarySearch()** ---- find an element in a sorted array.

All of these methods are **overloaded** for all the primitive types and **Objects**.

While useful, the **Arrays** class **isn't fully functional**.

Supplemental utilities to the **Arrays** class



C++ templates

Sample : Generator.java
BooleanGenerator.java
Arrays2.java

Random data generators are useful for testing. ---- Inner classes implementation.

9.2.1 Filling an array

Sample : FillingArrays.java

9.2.2 Copying an array

System.arraycopy()

Sample : CopyingArrays.java

Both **primitive** arrays and **object** arrays can be copied. However, if you copy arrays of objects then **only the references** get copied.

9.2.3 Comparing arrays

Arrays.equals() compares entire arrays for equality, using the **equals()** for each element. For primitives, that primitive's **wrapper** class **equals()** is used.

9.2.4 Array element comparisons

Writing generic sorting code ---- the code that **stays the same** is the **general sort algorithm**, and the thing that **changes** is **the way objects are compared**.

Java 2 provides **two** ways for comparison functionality.

1. With the *natural comparison method* ---- implement the `java.lang.Comparable` interface (`compareTo()`).

Sample : `CompType.java` *

If **Comparable** hasn't been implemented, then you'll get a compile-time error message when you try to call `sort()`.

2. Create a separate class that implements the **Comparator** interface. (`compare()` and `equals()`)

Sample : `ComparatorTest.java` *

See Sample : `AlphabeticSorting.java` after class. (P436)

9.2.5 Searching a sorted array

With the **built-in sorting** methods, you can sort **any** array of primitives, and any array of objects that either implements **Comparable** or has an associated **Comparator**.

Once an array is **sorted**, you can perform a fast search for a particular item using **Arrays.binarySearch()**.

Sample : **ArraySearching.java**

Arrays.binarySearch() produces a value (≥ 0) if the search item is found. Otherwise, it produces a negative value ($-(\text{insertion point}) - 1$).

If the array contains duplicate elements, there is **no guarantee** which one will be found.

If you have sorted an object array using a **Comparator**, you must include **that same Comparator** when you perform a **binarySearch()**.

Sample : **AlphabeticSearch.java**

Array summary :

Your **first and most efficient** choice to hold a group of objects should be an array, and you're forced into this choice if you want to hold a group of **primitives**.

9.3 Introduction to containers

Java container classes will **automatically resize** themselves.

Two kinds of container :

1. **Collection**: a group of individual elements ---- A **List** and a **Queue** must hold the elements in a particular sequence, and a **Set** cannot have any duplicate elements.
2. **Map**: a group of key-value object pairs. ---- A **Map** can return a **Set** of its keys, a **Collection** of its values, or a **Set** of its pairs.

9.3.1 Printing containers

Unlike arrays, the containers print nicely **without any help**.

Sample : **PrintingContainers.java**

A **Collection** is printed surrounded by “[]”, with each element separated by a “,”. A **Map** is surrounded by “{ }”, with each key and value associated with an “=”.

1. The **List** holds the objects exactly as they are entered.
2. The **Set** only accepts one of each object and it uses its own ordering method.
3. The **Map** also only accepts one of each type of item, based on the key, and it also has its own ordering.

9.3.2 Filling containers

Collections.fill() just duplicates a single object reference throughout the container, and only works for **List** objects and not **Sets** or **Maps**.

Supplemental utilities to the **Collections** class



Sample : Pair.java MapGenerator.java
Collections2.java CountryCapitals.java

Sample : FillTest.java

9.4 Container disadvantage: unknown type

You **lose type information** when you put an object into a container. The container holds references to **Object**.

1. There's **no restriction on the type** of object that can be put into a container. (not including primitives)

2. You must perform a **cast to the correct type** before you use a reference in a container.

You can think of **ArrayList** as “an array that **automatically expands** itself.”

Sample : **CatsAndDogs.java**



It's something that can create difficult-to-find bugs.

Making a type-conscious ArrayList :

A more ironclad solution is to create a new class using the **ArrayList**, such that it will **accept and produce only a specific type**.

Sample : MouseList

Note :If **MouseList** had instead been *inherited* from **ArrayList**, the **add(Mouse)** method would simply **overload** the existing **add(Object)** and there would still be **no restriction on type**.

```
mice.add(new Pigeon());
```



9.5 Iterators

In any **container** class, you must have a way to put things in and a way to get things out.

An **iterator** is an object whose job is to **move through** a sequence of objects and **select** each object **without** the client programmer **knowing the underlying structure** of that sequence.

An iterator is usually a “light-weight” object: one that’s cheap to create.

Java **Iterator** :

1. Ask a container to hand you an **Iterator** using **iterator()**.
2. Get the next object in the sequence with **next()**.
3. See if there *are* any more objects in the sequence with **hasNext()**.
4. Remove the last element returned by the iterator with **remove()**.

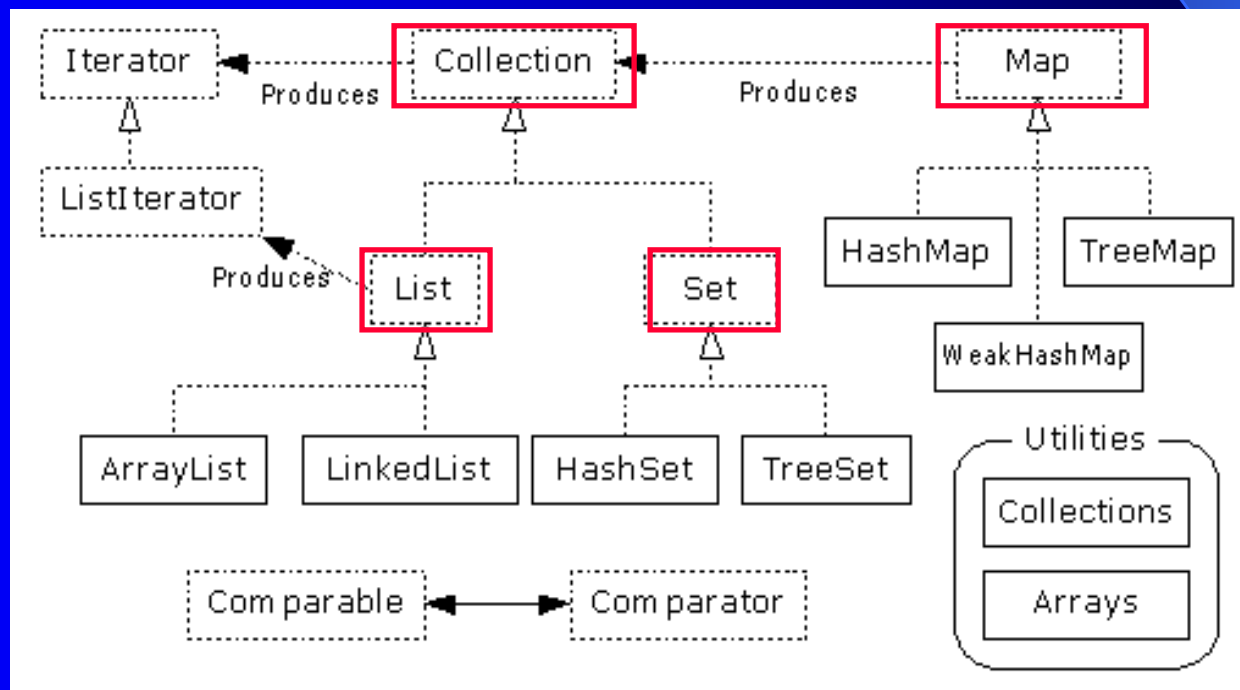
```
public class CatsAndDogs2 {  
    public static void main(String[] args) {  
        ArrayList cats = new ArrayList();  
        for(int i = 0; i < 7; i++)  
            cats.add(new Cat(i));  
        Iterator e = cats.iterator();  
        while(e.hasNext())  
            ((Cat)e.next()).print();  
    }  
} ///:~
```


With the **Iterator**, you don't need to worry about the **number** of elements in the container.

Sample : **HamsterMaze.java**

Hint : Improvements in JDK 1.5

9.6 Container taxonomy



Generics

```
/**
 * Remove the four-letter words from the specified
 * collection, which must contain only strings.
 */
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        String s = (String) i.next();
        if (s.length() == 4)
            i.remove();
    }
}
```



```
/**
 * Remove the four-letter words from the specified collection of strings.
 */
static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
        if (i.next().length() == 4)
            i.remove();
}
```

- Enhanced for loop

```
void cancelAll(Collection c) {  
    for (Iterator i = c.iterator(); i.hasNext(); ) {  
        TimerTask tt = (TimerTask) i.next();  
        tt.cancel();  
    }  
}
```



```
void cancelAll(Collection c) {  
    for (Object o : c)  
        ((TimerTask)o).cancel();  
}
```



```
void cancelAll(Collection<TimerTask> c) {  
    for (TimerTask task : c)  
        task.cancel();  
}
```

•Autoboxing/ unboxing

```
public class Freq {
    private static final Integer ONE = new Integer(1);

    public static void main(String args[]) {
        // Maps word (String) to frequency (Integer)
        Map m = new TreeMap();

        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE :
                new Integer(freq.intValue() + 1)));
        }
        System.out.println(m);
    }
}
```



```
public class Frequency {
    public static void main(String[] args) {
        Map<String, Integer> m = new TreeMap<String, Integer>();
        for (String word : args) {
            Integer freq = m.get(word);
            m.put(word, (freq == null ? 1 : freq + 1));
        }
        System.out.println(m);
    }
}
```

You'll typically make an object of a **concrete class**, **upcast** it to the corresponding **interface**, and then use the **interface** throughout the rest of your code.

Sample : SimpleCollection.java

9.7 Collection functionality

See the table in textbook, P463

If you want to examine all the elements of a **Collection** you must use an **iterator**.

9.8 List functionality

See the table in textbook, P467

Sample : List1.java

9.8.1 Making a stack from a LinkedList

Sample : StackL.java

If you want only stack behavior, **inheritance is inappropriate** because it would produce a class with all the rest of the **LinkedList** methods.

9.8.2 Making a queue from a LinkedList

Sample : Queue.java

9.9 Set functionality

Set has exactly the **same interface** as **Collection**. A Set **refuses** to hold **more than one** instance of each object value.

See the table in textbook, P473

A **Set** needs a way to maintain **a storage order**, so you must implement the **Comparable** interface when creating your own classes.

Sample : **Set2.java**

The **order** maintained by the **HashSet** is **different** from **TreeSet**.

You must define an **equals()** in **both** cases, but the **hashCode()** is absolutely necessary only if the class will be placed in a **HashSet**.

A **HashSet** implementation should generally be your **first choice**.

9.10 Map functionality

It seems like an **ArrayList**, but instead of looking up objects using a number, you look them up using *another object*.

See the table in textbook, P477

A **HashMap** takes the **hashCode()** of the object and uses it to **quickly hunt** for the key.

The **keySet()** method produces a **Set** backed by the keys in the **Map**.
The **values()** method produces a **Collection** containing all the values in the **Map**.

See Sample : Map1.java

Note :keys must be **unique**, while values may contain duplicates.

Sample : Statistics.java *

In **main()**, each time a random number is generated it is **wrapped** inside an **Integer** object so that **reference** can be used with the **HashMap**.

9.10.1 Hashing and hash codes

1. **Object's hashCode()** method is used to generate the hash code for each object, and **by default** it just uses the **address** of the object.

2. The default **Object.equals()** simply compares object **addresses**.

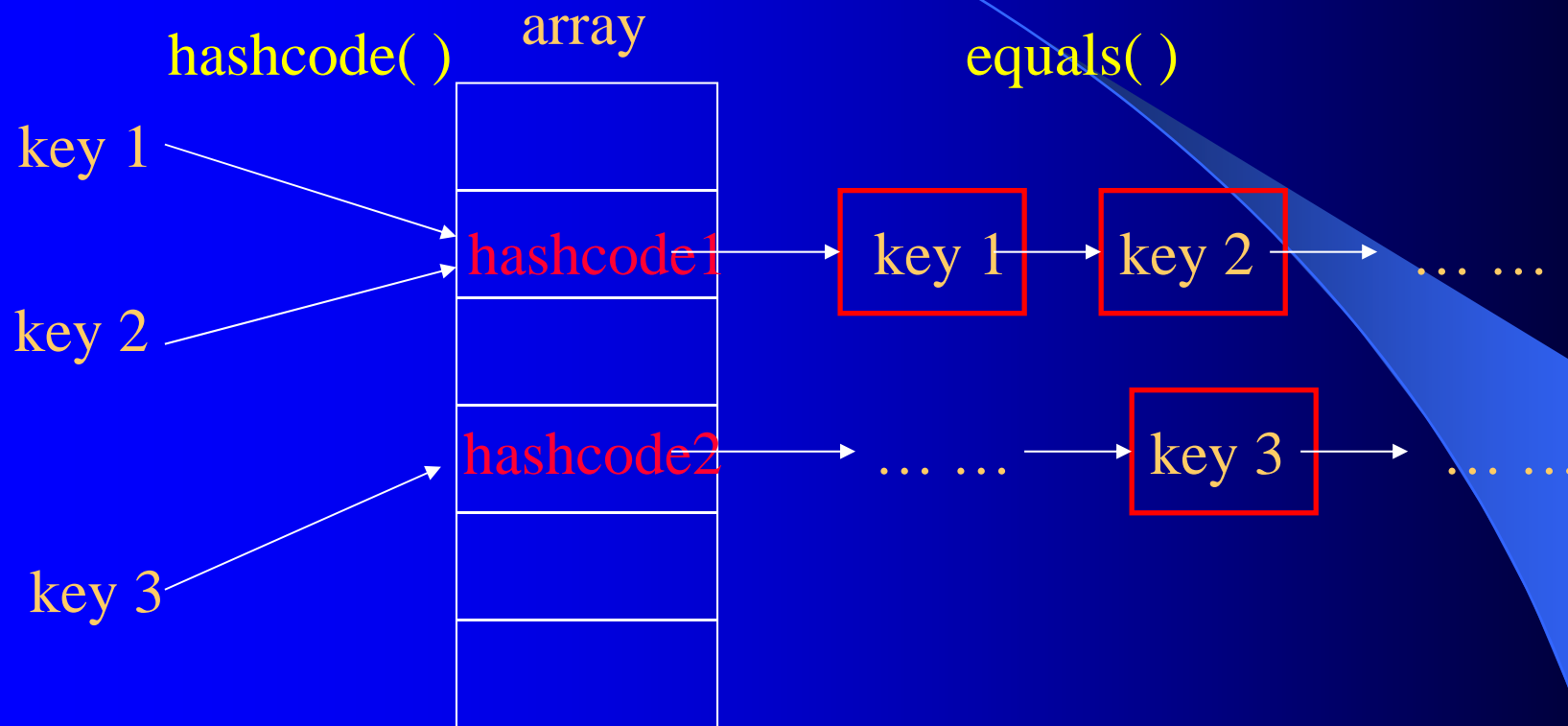
Summary : To use your **own classes as keys** in a **HashMap**, you **must** override both **hashCode()** and **equals()**.

Sample : **SpringDetector2.java**

The **hashCode()** is not required to return a unique identifier, but the **equals()** must be able to **strictly determine** whether two objects are equivalent.

9.10.2 Understanding hashCode()

The whole point of **hashing** is **speed**: hashing allows the **lookup** to happen quickly.



Sample :
SlowMap.java



Sample :
SimpleHashMap.java

9.10.3 Overriding hashCode()

The most important factor is, regardless of when **hashCode()** is called, it produces **the same value** for a particular object.

For a **hashCode()** to be effective, it must be **fast and meaningful**.

A good **hashCode()** should result in an **even distribution** of values.

Sample : **CountedString.java**

9.11 Choosing an implementation

There are really **only four** container components: **Map**, **List**, **Queue** and **Set**, and only **two or three** implementations of each interface.

Each different implementation has its own **features**, **strengths**, and **weaknesses**.

1. If you want to do many **insertions and removals in the middle** of a list, a **LinkedList** is the appropriate choice. If not, an **ArrayList** is typically **faster**.

2. The **performance of HashSet** is generally **superior** to **TreeSet** for all operations. The only reason **TreeSet** exists is that it maintains its elements in **sorted order**.

3. when you're using a **Map** your **first choice** should be **HashMap**, and only if you need a **constantly sorted Map** will you need **TreeMap**.

There's **no need** to use the legacy classes **Vector**, **Hashtable** and **Stack** in new code.