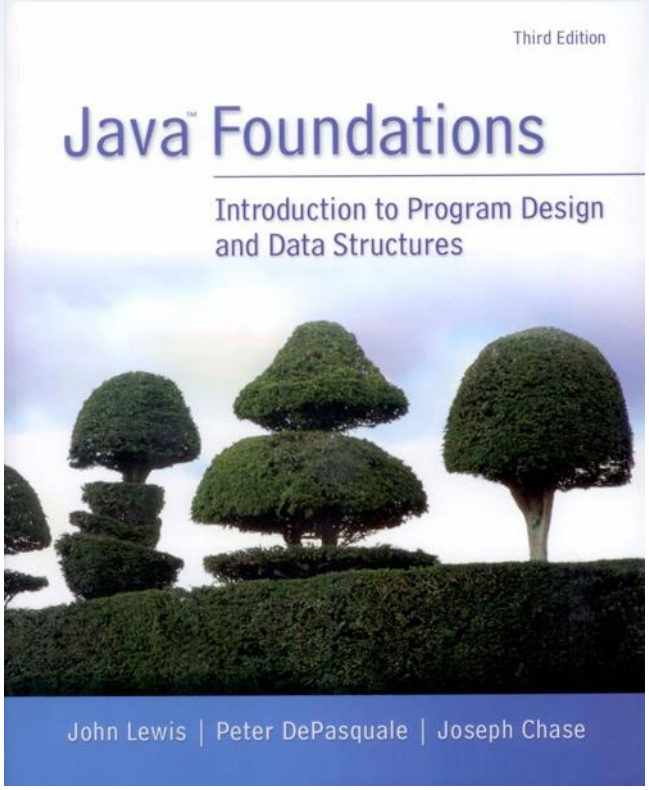

Chapter 22

Sets and Maps

Chapter Scope

- The Java API set and map collections
- Using sets and maps to solve problems
- How the Java API implements sets and maps
- Hashing

Sets

- A *set* is a collection of elements with no duplicates
- No other relationship among the elements should be assumed
- A primary purpose of a set is to determine whether a particular element is a member
- Other collections (such as lists) can test for containment, but if that operation is important, consider using sets

- Operations on a set:

Method Summary	
`boolean`	`add(E e)` Adds the specified element to this set if it is not already present (optional operation).
`boolean`	`addAll(Collection<? extends E> c)` Adds all of the elements in the specified collection to this set if they are not already present (optional operation).
`void`	`clear()` Removes all of the elements from this set (optional operation).
`boolean`	`contains(Object o)` Returns `true` if this set contains the specified element.
`boolean`	`containsAll(Collection<?> c)` Returns `true` if this set contains all of the elements of the specified collection.
`boolean`	`equals(Object o)` Compares the specified object with this set for equality.
`int`	`hashCode()` Returns the hash code value for this set.
`boolean`	`isEmpty()` Returns `true` if this set contains no elements.
`Iterator`	`iterator()` Returns an iterator over the elements in this set.
`boolean`	`remove(Object o)` Removes the specified element from this set if it is present (optional operation).
`boolean`	`removeAll(Collection<?> c)` Removes from this set all of its elements that are contained in the specified collection (optional operation).
`boolean`	`retainAll(Collection<?> c)` Retains only the elements in this set that are contained in the specified collection (optional operation).
`int`	`size()` Returns the number of elements in this set (its cardinality).

Sets

- Operations on a set, continued:

Method Summary (*continued*)	
`Object[]`	`toArray()` Returns an array containing all of the elements in this set.
`<T> T[]`	`toArray(T[] a)` Returns an array containing all of the elements in this set; the run-time type of the returned array is that of the specified array.

Maps

- A map is a collection that establishes a relationship between keys and values
- The goal is to have an efficient way of obtaining a value given its key
- Keys of a map must be unique, but multiple keys could map to the same object
- For example, a membership id (a `String`) could be the key used to retrieve a member (a `Member` object)

Maps

- Operations on a map:

Method Summary	
`void`	`clear()` Removes all of the mappings from this map (optional operation).
`boolean`	`containsKey(Object key)` Returns `true` if this map contains a mapping for the specified key.
`boolean`	`containsValue(Object value)` Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	`entrySet()` Returns a `Set` view of the mappings contained in this map.
`boolean`	`equals(Object o)` Compares the specified object with this map for equality.
`V`	`get(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`int`	`hashCode()` Returns the hash code value for this map.
`boolean`	`isEmpty()` Returns `true` if this map contains no key-value mappings.
`Set<K>`	`keySet()` Returns a `Set` view of the keys contained in this map.

Maps

- Operations on a map, continued:

Method Summary (*continued*)	
`V`	`put(K key, V value)` Associates the specified value with the specified key in this map (optional operation).
`void`	`putAll(Map<? extends K,? extends V> m)` Copies all of the mappings from the specified map to this map (optional operation).
`V`	`remove(Object key)` Removes the mapping for a key from this map if it is present (optional operation).
`int`	`size()` Returns the number of key-value mappings in this map.
`Collection<V>`	`values()` Returns a Collection view of the values contained in this map.

Using Sets

- Let's use a set to test for containment
- In particular, we'll create a set of web domains that will be blocked
- The blocked domains will be held in an input file
- The set is represented using a `TreeSet` object from the Java API

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.TreeSet;

/**
 * A URL domain blocker.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class DomainBlocker
{
    private TreeSet<String> blockedSet;

    /**
     * Sets up the domain blocker by reading in the blocked domain names from
     * a file and storing them in a TreeSet.
     * @throws FileNotFoundException
     */
    public DomainBlocker() throws FileNotFoundException
    {
        blockedSet = new TreeSet<String>();

        File inputFile = new File("blockedDomains.txt");
        Scanner scan = new Scanner(inputFile);

        while (scan.hasNextLine())
        {
            blockedSet.add(scan.nextLine());
        }
    }
```

```
    /**
     * Checks to see if the specified domain has been blocked.
     *
     * @param domain the domain to be checked
     * @return true if the domain is blocked and false otherwise
     */
    public boolean domainIsBlocked(String domain)
    {
        return blockedSet.contains(domain);
    }
}
```

```
import java.io.FileNotFoundException;
import java.util.Scanner;

/**
 * Domain checking driver.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class DomainChecker
{
    /**
     * Repeatedly reads a domain interactively from the user and checks to
     * see if that domain has been blocked.
     */
    public static void main(String[] args) throws FileNotFoundException
    {
        DomainBlocker blocker = new DomainBlocker();
        Scanner scan = new Scanner(System.in);

        String domain;
```

```
        do
        {
            System.out.print("Enter a domain (DONE to quit): ");
            domain = scan.nextLine();

            if (!domain.equalsIgnoreCase("DONE"))
            {
                if (blocker.domainIsBlocked(domain))
                    System.out.println("That domain is blocked.");
                else
                    System.out.println("That domain is fine.");
            }
        } while (!domain.equalsIgnoreCase("DONE"));
    }
}
```

Using Maps

- Now let's look at an example using maps to keep track of product sales
- Suppose that each time a product is sold, its product code is entered into a sales file
- Our example will read the sales file and update product sales summary
- A `TreeMap` will be used to map the product code to the `Product` object

```java
/**
 * Represents a product for sale.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class Product implements Comparable<Product>
{
    private String productCode;
    private int sales;

    /**
     * Creates the product with the specified code.
     *
     * @param productCode a unique code for this product
     */
    public Product(String productCode)
    {
        this.productCode = productCode;
        this.sales = 0;
    }
```

```
/**
 * Returns the product code for this product.
 *
 * @return the product code
 */
public String getProductCode()
{
    return productCode;
}

/**
 * Increments the sales of this product.
 */
public void incrementSales()
{
    sales++;
}
```

```
    /**
     * Compares this product to the specified product based on the product
     * code.
     *
     * @param other the other product
     * @return an integer code result
     */
    public int compareTo(Product obj)
    {
        return productCode.compareTo(obj.getProductCode());
    }

    /**
     * Returns a string representation of this product.
     *
     * @return a string representation of the product
     */
    public String toString()
    {
        return productCode + "\t(" + sales + ")";
    }
}
```

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;
import java.util.TreeMap;

/**
 * Demonstrates the use of a TreeMap to store a sorted group of Product
 * objects.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class ProductSales
{
    /**
     * Processes product sales data and prints a summary sorted by
     * product code.
     */
    public static void main(String[] args) throws IOException
    {
        TreeMap<String, Product> sales = new TreeMap<String, Product>();

        Scanner scan = new Scanner(new File("salesData.txt"));

        String code;
        Product product;
```

```
        while (scan.hasNext())
        {
            code = scan.nextLine();
            product = sales.get(code);
            if (product == null)
                sales.put(code, new Product(code));
            else
                product.incrementSales();
        }

        System.out.println("Products sold this period:");
        for (Product prod : sales.values())
            System.out.println(prod);
    }
}
```

Using Maps

- Another example using maps stores user information which can be retrieved by a user id
- This time, a `HashMap` is used to store the users

```java
/**
 * Represents a user with a userid.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class User
{
    private String userId;
    private String firstName;
    private String lastName;

    /**
     * Sets up this user with the specified information.
     *
     * @param userId a user identification string
     * @param firstName the user's first name
     * @param lastName the user's last name
     */
    public User(String userId, String firstName, String lastName)
    {
        this.userId = userId;
        this.firstName = firstName;
        this.lastName = lastName;
    }
```

```
    /**
     * Returns the user id of this user.
     *
     * @return the user id of the user
     */
    public String getUserId()
    {
        return userId;
    }

    /**
     * Returns a string representation of this user.
     *
     * @return a string representation of the user
     */
    public String toString()
    {
        return userId + ":\t" + lastName + ", " + firstName;
    }
}
```

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

/**
 * Stores and manages a map of users.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class Users
{
    private HashMap<String, User> userMap;

    /**
     * Creates a user map to track users.
     */
    public Users()
    {
        userMap = new HashMap<String, User>();
    }

    /**
     * Adds a new user to the user map.
     *
     * @param user the user to add
     */
    public void addUser(User user)
    {
        userMap.put(user.getUserId(), user);
    }
```

```
    /**
     * Retrieves and returns the specified user.
     *
     * @param userId the user id of the target user
     * @return the target user, or null if not found
     */
    public User getUser(String userId)
    {
        return userMap.get(userId);
    }

    /**
     * Returns a set of all user ids.
     *
     * @return a set of all user ids in the map
     */
    public Set<String> getUserIds()
    {
        return userMap.keySet();
    }
}
```

```
import java.io.IOException;
import java.util.Scanner;

/**
 * Demonstrates the use of a map to manage a set of objects.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class UserManagement
{
    /**
     * Creates and populates a group of users. Then prompts for interactive
     * searches, and finally prints all users.
     */
    public static void main(String[] args) throws IOException
    {
        Users users = new Users();

        users.addUser(new User("fziffle", "Fred", "Ziffle"));
        users.addUser(new User("geoman57", "Marco", "Kane"));
        users.addUser(new User("rover322", "Kathy", "Shear"));
        users.addUser(new User("appleseed", "Sam", "Geary"));
        users.addUser(new User("mon2016", "Monica", "Blankenship"));

        Scanner scan = new Scanner(System.in);
        String uid;
        User user;
```

```
        do
        {
            System.out.print("Enter User Id (DONE to quit): ");
            uid = scan.nextLine();
            if (!uid.equalsIgnoreCase("DONE"))
            {
                user = users.getUser(uid);
                if (user == null)
                    System.out.println("User not found.");
                else
                    System.out.println(user);
            }
        } while (!uid.equalsIgnoreCase("DONE"));

        // print all users
        System.out.println("\nAll Users:\n");
        for (String userId : users.getUserIds())
            System.out.println(users.getUser(userId));
    }
}
```

Implementing Sets and Maps

- The `TreeSet` and `TreeMap` classes use an underlying tree to hold the elements of the set or map
- These trees are binary search trees, and in particular are red-black balanced trees
- All of the basic operations are performed with O(log n) efficiency

Implementing Sets and Maps

- The `HashSet` and `HashMap` classes are implemented using a technique called *hashing*
- An element in a *hash table* is stored in a location based on a *hash function*
- A hash function makes some calculation based on the element itself
- Multiple elements may be stored in the same location (which is called a *collision*)
- A function that maps each element to a unique location is called a *perfect hashing function*

Hashing

- A hash table with a simple hash function based on the first letter of the string:
- Collisions would be managed using a list of A names, a list of B names, etc.