Chapter 15 -The Java Collections Framework

Working with Maps

Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.
`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

`for (String key : scores.keySet())`
`{`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

Map

- Sometimes you want to enumerate all keys in a map.
- The `keySet` method yields the set of keys.
- Ask the key set for an iterator and get all keys.
- For each key, you can find the associated value with the `get` method.
- To print all key/value pairs in a map `m`:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
    Color value = m.get(key);
    System.out.println(key + "->" + value);
}
```

MapDemo.java

```
1  import java.awt.Color;
2  import java.util.HashMap;
3  import java.util.Map;
4  import java.util.Set;
5
6  /**
7     This program demonstrates a map that maps names to colors.
8  */
9  public class MapDemo
10 {
11    public static void main(String[] args)
12    {
13       Map<String, Color> favoriteColors = new HashMap<>();
14       favoriteColors.put("Juliet", Color.BLUE);
15       favoriteColors.put("Romeo", Color.GREEN);
16       favoriteColors.put("Adam", Color.RED);
17       favoriteColors.put("Eve", Color.BLUE);
18
19       // Print all keys and values in the map
20
21       Set<String> keySet = favoriteColors.keySet();
22       for (String key : keySet)
23       {
24          Color value = favoriteColors.get(key);
25          System.out.println(key + " : " + value);
26       }
27    }
28 }
```

Program Run:

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

Self Check 15.16

What is the difference between a set and a map?

Answer: A set stores elements. A map stores associations between keys and values.

Self Check 15.19

Suppose you want to track how many times each word occurs in a document. Declare a suitable map variable.

Answer: `Map<String, Integer> wordFrequency;`

Self Check 15.20

What is a `Map<String, HashSet<String>>`? Give a possible use for such a structure.

Answer: It associates strings with sets of strings. One application would be a thesaurus that lists synonyms for a given word. For example, the key `"improve"` might have as its value the set `["ameliorate", "better", "enhance", "enrich", "perfect", "refine"]`.

Java 8 Note 15.1

- Updating Map Entries used to be tedious

```
Integer count = frequencies.get(word); // Get the old frequency count
// If there was none, put 1; otherwise, increment the count
if (count == null) { count = 1; }
else { count = count + 1; }
frequencies.put(word, count);
```

- New `merge` method in `Map` interface
- Specify:

  Key

  Value to be used if key not yet present

  Function to compute the updated value if key is present

- Simplified approach

```
Map<String, Integer> freq = new HashMap<>();
freq.put("Cat", 10); // initialization


// When updating the entry
freq.putIfAbsent("Cat", 0);
freq.put("Cat", freq.get("Cat") + 1);
System.out.println(freq.get("Cat"));
```

Java 8 Note 15.1

- Updating Map Entries used to be tedious

```
Integer count = frequencies.get(word); // Get the old frequency count
// If there was none, put 1; otherwise, increment the count
if (count == null) { count = 1; }
else { count = count + 1; }
frequencies.put(word, count);
```

- New `merge` method in `Map` interface
- Specify:

  Key

  Value to be used if key not yet present

  Function to compute the updated value if key is present

- Replace code above with single line using lambda expression:

```
frequencies.merge("key", 1, (Integer oldValue, Integer newValue) -> oldValue
+ newValue);
```

```
Map<String, Integer> frequencies = new HashMap<>();
frequencies.put("Cat", 1);
frequencies.merge("Cat", 1, (Integer oldValue, Integer newValue) -> oldValue + newValue);

count = frequencies.get("Cat");
System.out.println(count);
```

Choosing a Collection

1. Determine how you access the values.
2. Determine the element types or key/value types.
3. Determine whether element or key order matters.
4. For a collection, determine which operations must be efficient.
5. For hash sets and maps, decide whether you need to implement the hashCode and equals methods.
6. If you use a tree, decide whether to supply a comparator.

Hash Functions

- You may need to implement a hash function for your own classes.
- **A hash function**: a function that computes an integer value, the hash code, from an object in such a way that different objects are likely to yield different hash codes.
- `Object` class has a `hashCode` method

  you need to override it to use your class in a hash table
- A collision: two or more objects have the same hash code.
- The method used by the `String` class to compute the hash code.

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
{
   h = HASH_MULTIPLIER * h + s.charAt(i);
}
```

- This produces different hash codes for `"tea"` and `"eat"`.

Hash Functions

A good hash function produces different hash values for each object so that they are scattered about in a hash table.

Hash Functions

- Override `hashCode` methods in your own classes by combining the hash codes for the instance variables.
- A hash function for our `Country` class:

```
public class Country
{
   public int hashCode()
   {
      int h1 = name.hashCode();
      int h2 = new Double(area).hashCode();
      final int HASH_MULTIPLIER = 29;
      int h = HASH_MULTIPLIER * h1 + h2;
      return h;
   }
}
```

- Easier to use `Objects.hash()` method
  - Takes hashcode of all arguments and multiplies them:

```
public int hashCode()
{
   return Objects.hash(name, area);
}
```

- A class's `hashCode` method must be compatible with its `equals` method.

Stacks

- A stack lets you insert and remove elements only at one end:

  Called the top of the stack.

  Removes items in the opposite order than they were added

  Last-in, first-out or LIFO order

- Add and remove methods are called `push` and `pop`.
- Example

```
Stack<String> s = new Stack<>();
s.push("A"); s.push("B"); s.push("C");
while (s.size() > 0)
{
   System.out.print(s.pop() + " "); // Prints C B A
}
```

- The last pancake that has been added to this stack will be the first one that is consumed.

Stacks

- Many applications for stacks in computer science.
- Consider: Undo function of a word processor

  The issued commands are kept in a stack.

  When you select “Undo”, the **last** command is popped off the stack and undone.

- Run-time stack that a processor or virtual machine:

  Stores the values of variables in nested methods.

  When a new method is called, its parameter variables and local variables are pushed onto a stack.

  When the method exits, they are popped off again.

Stack in the Java Library

- `Stack` class provides `push`, `pop` and `peek` methods.

Table 7 Working with Stacks

`Stack<Integer> s = new Stack<>();`	Constructs an empty stack.

`s.push(1);`
`s.push(2);`
`s.push(3);`
	Adds to the top of the stack; `s` is now `[1, 2, 3]`. (Following the `toString` method of the `Stack` class, we show the top of the stack at the end.)

`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and `s` is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

Queue

- A queue

  Lets you add items to one end of the queue (the tail)

  Remove items from the other end of the queue (the head)

  Items are removed in the same order in which they were added

  First-in, first-out or FIFO order

- To visualize a queue, think of people lining up.

Photodisc/Punchstock.

- Typical application: a print queue.

Queue

- The `Queue` interface in the standard Java library has:

  an `add` method to add an element to the tail of the queue,

  a `remove` method to remove the head of the queue, and

  a `peek` method to get the head element of the queue without removing it.

- The `LinkedList` class implements the `Queue` interface.
- When you need a queue, initialize a `Queue` variable with a `LinkedList` object:

```
Queue<String> q = new LinkedList<>();
 q.add("A"); q.add("B"); q.add("C");
 while (q.size() > 0) { System.out.print(q.remove() + " "); } // Prints A B  C
```

Table 8 Working with Queues

`Queue<Integer> q = new LinkedList<>();`	The `LinkedList` class implements the `Queue` interface.

`q.add(1);`
`q.add(2);`
`q.add(3);`
	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.

`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.

Priority Queues

- A priority queue collects elements, each of which has a priority.
- Example: a collection of work requests, some of which may be more urgent than others.
- Does not maintain a first-in, first-out discipline.
- Elements are retrieved according to their priority.
- Priority 1 denotes the most urgent priority.

  Each removal extracts the minimum element.

- When you retrieve an item from a priority queue, you always get the most urgent one.

Priority Queues

- Example: objects of a class `WorkOrder` into a priority queue.

```
PriorityQueue<WorkOrder> q = new PriorityQueue<>();
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix broken sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

- When calling `q.remove()` for the first time, the work order with priority 1 is removed.
- Elements should belong to a class that implements the `Comparable` interface.

Table 9 Working with Priority Queues

`PriorityQueue<Integer> q = new PriorityQueue<>();`	This priority queue holds `Integer` objects. In practice, you would use objects that describe tasks.
`q.add(3); q.add(1); q.add(2);`	Adds values to the priority queue.
`int first = q.remove(); int second = q.remove();`	Each call to `remove` removes the most urgent item: `first` is set to 1, `second` to 2.
`int next = q.peek();`	Gets the smallest value in the priority queue without removing it.

Self Check 15.21

Why would you want to declare a variable as

```
Queue<String> q = new LinkedList<>();
```

instead of simply declaring it as a linked list?

Answer: This way, we can ensure that only queue operations can be invoked on the `q` object.

Self Check 15.22

Why wouldn't you want to use an array list for implementing a queue?

Answer: Depending on whether you consider the 0 position the head or the tail of the queue, you would either add or remove elements at that position. Both are inefficient operations because all other elements need to be moved.

Self Check 15.23

What does this code print?

```
Queue<String> q =
new LinkedList<>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0) { System.out.print(q.remove() + " "); }
```

Answer: A B C

Self Check 15.24

Why wouldn't you want to use a stack to manage print jobs?

Answer: Stacks use a “last-in, first-out” discipline. If you are the first one to submit a print job and lots of people add print jobs before the printer has a chance to deal with your job, they get their printouts first, and you have to wait until all other jobs are completed.

Self Check 15.25

In the sample code for a priority queue, we used a `WorkOrder` class. Could we have used strings instead?

```
PriorityQueue<String> q = new
PriorityQueue<>();
q.add("3 - Shampoo carpets");
q.add("1 - Fix broken sink");
q.add("2 - Order cleaning supplies");
```

Answer: Yes––the smallest string (in lexicographic ordering) is removed first. In the example, that is the string starting with 1, then the string starting with 2, and so on. However, the scheme breaks down if a priority value exceeds 9. For example, a string `"10 - Line up braces"` comes before `"2 - Order cleaning supplies"` in lexicographic order.

Stack and Queue Applications

- A stack can be used to check whether parentheses in an expression are balanced.

```
When you see an opening parenthesis, push it on the stack.
When you see a closing parenthesis, pop the stack.
If the opening and closing parentheses don't match
   The parentheses are unbalanced. Exit.
If at the end the stack is empty
   The parentheses are balanced.
Else
   The parentheses are not balanced.
```

- Walkthrough of the sample expression:

Stack	Unread expression	Comments
Empty	-{ [b * b - (4 * a * c)] / (2 * a) }	
{	[b * b - (4 * a * c)] / (2 * a) }	
{ [	b * b - (4 * a * c)] / (2 * a) }	
{ [(	4 * a * c)] / (2 * a) }	
{ [	] / (2 * a) }	(matches)
{	/ (2 * a) }	[matches]
{ (	2 * a) }	
{	}	(matches)
Empty	No more input	{ matches }
		The parentheses are balanced

Stack and Queue Applications

- Use a stack to evaluate expressions in reverse Polish notation.

```
If you read a number
   Push it on the stack.
Else if you read an operand
   Pop two values off the stack.
   Combine the values with the operand.
   Push the result back onto the stack.
Else if there is no more input
   Pop and display the result.
```

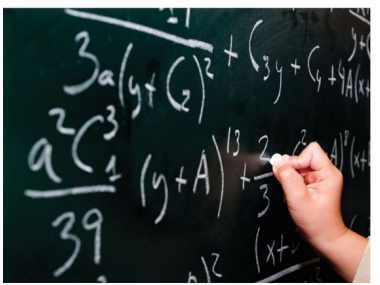
- Walkthrough of evaluating the expression 3 4 5 + ×:

Stack	Unread expression	Comments
Empty	3 4 5 + x	
3	4 5 + x	Numbers are pushed on the stack
3 4	5 + x	
3 4 5	+ x	
3 9	x	Pop 4 and 5, push 4 5 +
27	No more input	Pop 3 and 9, push 3 9 x
Empty		Pop and display the result, 27

Evaluating Algebraic Expressions with Two Stacks

- Using two stacks, you can evaluate expressions in standard algebraic notation.

  One stack for numbers, one for operators



© Jorge Delgado/iStockphoto.

- Evaluating the top: 3 + 4

	Number stack Empty	Operator stack Empty	Unprocessed input 3 + 4	Comments
1	3		+ 4	
2	3	+	4	
3	4 3	+	No more input	Evaluate the top.
4	7			The result is 7.

Evaluating Algebraic Expressions with Two Stacks

- Evaluate 3 x 4 + 5

Push until you get to the +

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × 4 + 5	Comments
1	3		× 4 + 5	
2	3	×	4 + 5	
3	4 3	×	+ 5	Evaluate × before +.

x (top of operator stack) has higher precedence than + , so evaluate the top

	Number stack	Operator stack		Comments
4	12	+	5	
5	5 12	+	No more input	Evaluate the top.
6	17			That is the result.

Evaluating Algebraic Expressions with Two Stacks

- Evaluate 3 + 4 × 5

Add x to the operator stack so we can get the next number

	Number stack Empty	Operator stack Empty	Unprocessed input 3 + 4 × 5	Comments
1	3		+ 4 × 5	
2	3	+	4 + 5	
3	4 3	+	× 5	Don't evaluate + yet.
4	4 3	× +	5	

Keep operators on the stack until they are ready to be evaluated

	Number stack	Operator stack		Comments
5	5 4 3	× +	No more input	Evaluate the top.
6	20 3	+		Evaluate top again.
7	23			That is the result.

Evaluating Algebraic Expressions with Two Stacks

- Evaluating parentheses: 3 × (4 + 5)

  Push (on the stack

  Keep pushing until we reach the)

  Evaluate until we find the matching (

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × (4 + 5)	Comments
1	3		× (4 + 5)	
2	3	×	(4 + 5)	
3	3	(×	4 + 5)	Don't evaluate × yet.
4	4 3	(×	+ 5)	
5	4 3	+ (×	5)	
6	5 4 3	+ (×	)	Evaluate the top.
7	9 3	(×	No more input	Pop (.
8	9 3	×		Evaluate top again.
9	27			That is the result.

Evaluating Algebraic Expressions with Two Stacks

- The algorithm

```
If you read a number
  Push it on the number stack.
Else if you read a (
  Push it on the operator stack.
Else if you read an operator op
  While the top of the stack has a higher precedence than op
    Evaluate the top.
  Push op on the operator stack.
Else if you read a )
  While the top of the stack is not a (
    Evaluate the top.
  Pop the (.
Else if there is no more input
  While the operator stack is not empty
    Evaluate the top.
```

At the end, the value on the number stack is the value of the expression.