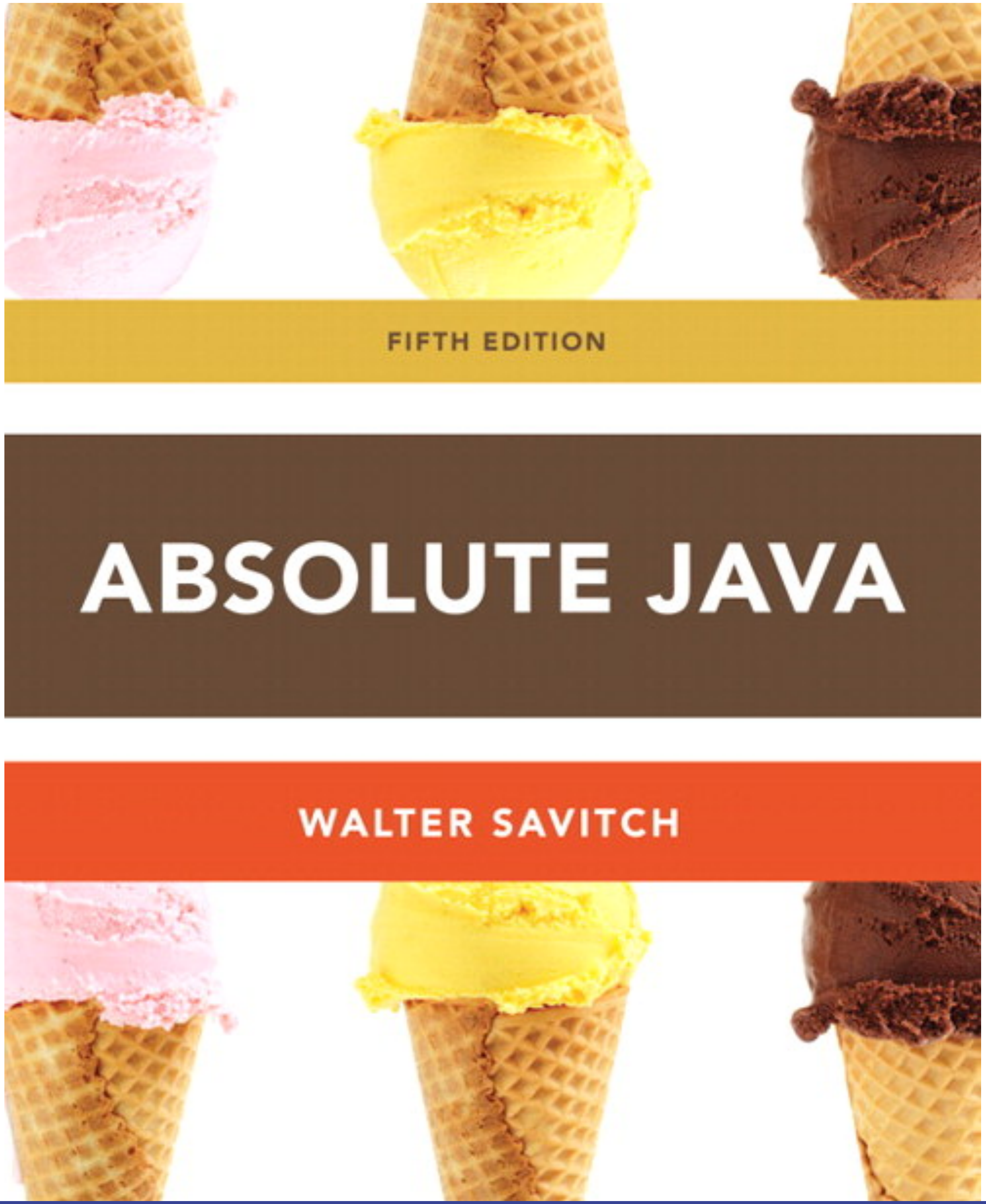


Chapter 12

UML and Patterns

Slides prepared by Rose Williams,
Binghamton University

Kenrick Mock, *University of Alaska Anchorage*

Patterns

- *Patterns* are design outlines that apply across a variety of software applications
  - To be useful, a pattern must apply across a variety of situations
  - To be substantive, a pattern must make some assumptions about the domain of applications to which it applies

Container-Iterator Pattern

- A *container* is a class or other construct whose objects hold multiple pieces of data
  - An array is a container
  - Vectors and linked lists are containers
  - A String value can be viewed as a container that contains the characters in the string
- Any construct that can be used to cycle through all the items in a container is an *iterator*
  - An array index is an iterator for an array
- The *Container-Iterator* pattern describes how an iterator is used on a container

Adaptor Pattern

- The Adaptor pattern transforms one class into a different class without changing the underlying class, but by merely adding a new interface
  - For example, one way to create a stack data structure is to start with an array, then add the stack interface

The Model-View-Controller Pattern

- The *Model-View-Controller* pattern is a way of separating the I/O task of an application from the rest of the application
  - The Model part of the pattern performs the heart of the application
  - The View part displays (outputs) a picture of the Model's state
  - The Controller is the input part: It relays commands from the user to the Model

The Model-View-Controller Pattern

- Each of the three interacting parts is normally realized as an object with responsibilities for its own tasks
- The Model-View-Controller pattern is an example of a divide-and-conquer strategy
  - One big task is divided into three smaller tasks with well-defined responsibilities

The Model-View-Controller Pattern

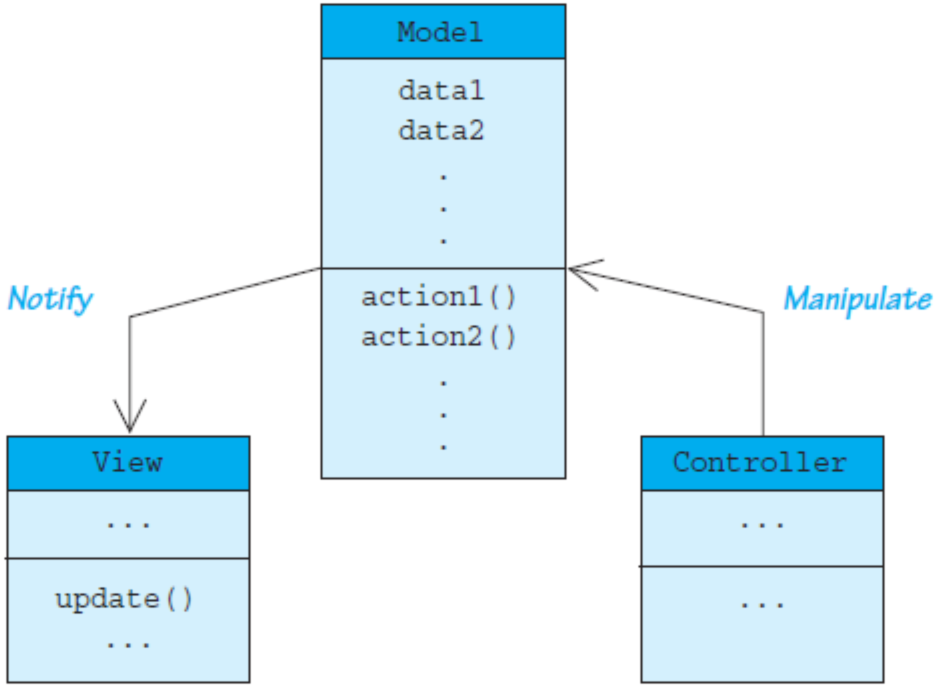
- As an example, the Model might be a container class, such as an array.
- The View might display one element of the array
- The Controller would give commands to display the element at a specified index
- The Model would notify the View to display a new element whenever the array contents changed or a different index location was given

The Model-View-Controller Pattern

- Any application can be made to fit the Model-View-Controller pattern, but it is particularly well suited to GUI (Graphical User Interface) design projects
  - The View can then be a visualization of the state of the Model

Model-View-Controller Pattern

Display 12.4 Model-View-Controller Pattern



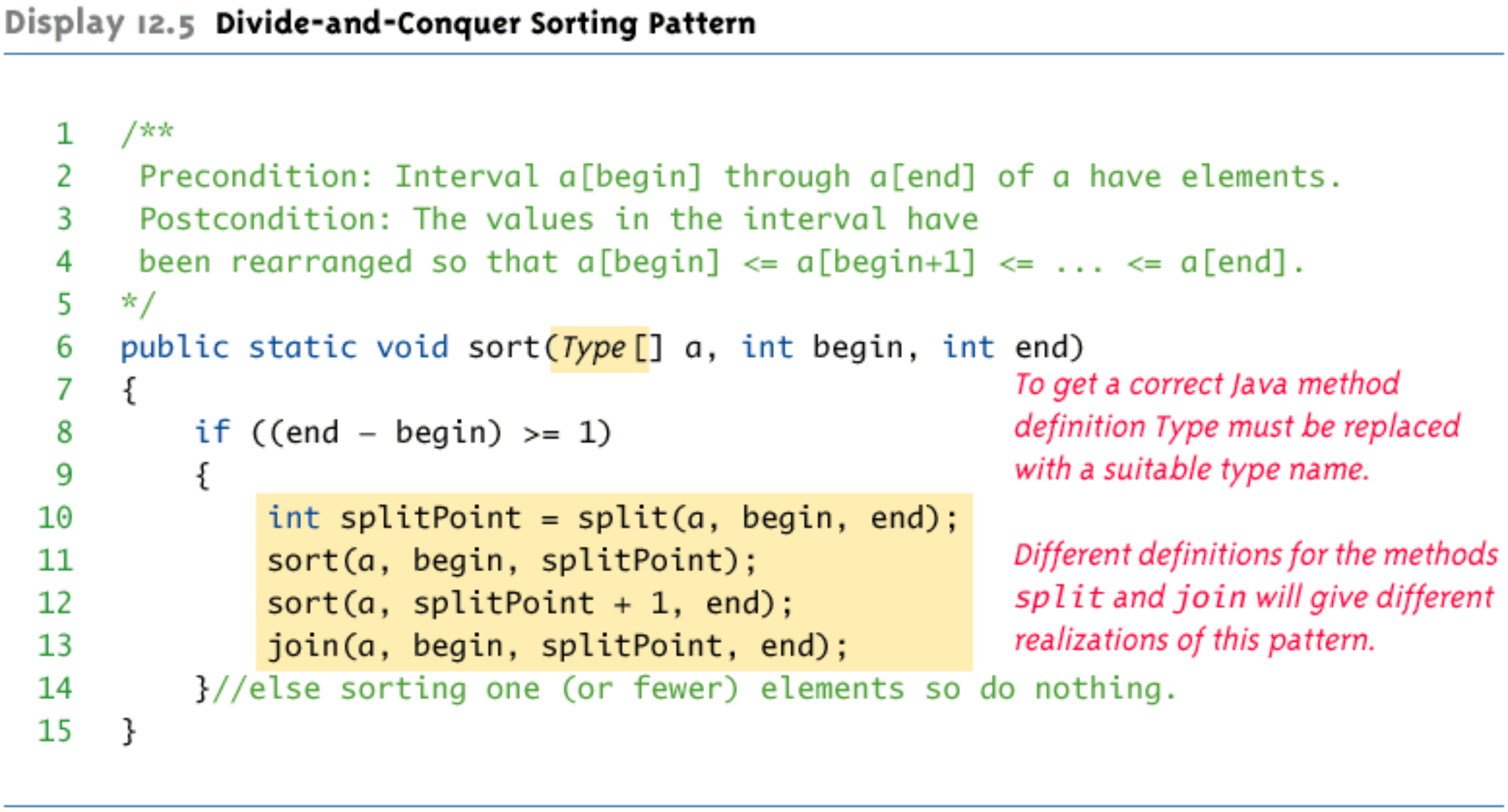
A Sorting Pattern

- The most efficient sorting algorithms all seem to follow a divide-and-conquer strategy
- Given an array `a`, and using the `<` operator, these sorting algorithms:
  - Divide the list of elements to be sorted into two smaller lists (`split`)
  - Recursively sort the two smaller lists (`sort`)
  - Then recombine the two sorted lists (`join`) to obtain the final sorted list

A Sorting Pattern

- The method **`split`** rearranges the elements in the interval **`a[begin]`** through **`a[end]`** and divides the rearranged interval at **`splitPoint`**
- The two smaller intervals are then sorted by a recursive call to the method **`sort`**
- After the two smaller intervals are sorted, the method **`join`** combines them to obtain the final sorted version of the entire larger interval
- Note that the pattern does not say exactly how the methods **`split`** and **`join`** are defined
  - Different definitions of **`split`** and **`join`** will yield different sorting algorithms

Divide-and-Conquer Sorting Pattern



Display 12.5 **Divide-and-Conquer Sorting Pattern**

```
/**
 Precondition: Interval a[begin] through a[end] of a have elements.
 Postcondition: The values in the interval have
 been rearranged so that a[begin] <= a[begin+1] <= ... <= a[end].
*/
public static void sort(Type[] a, int begin, int end)
{
    if ((end - begin) >= 1)
    {
        int splitPoint = split(a, begin, end);
        sort(a, begin, splitPoint);
        sort(a, splitPoint + 1, end);
        join(a, begin, splitPoint, end);
    }//else sorting one (or fewer) elements so do nothing.
}
```

To get a correct Java method definition Type must be replaced with a suitable type name.

Different definitions for the methods `split` and `join` will give different realizations of this pattern.

Merge Sort

- The simplest realization of this sorting pattern is the *merge sort*
- The definition of `split` is very simple
  - It divides the array into two intervals without rearranging the elements
- The definition of `join` is more complicated
- Note: There is a trade-off between the complexity of the methods `split` and `join`
  - Either one can be made simpler at the expense of making the other more complicated

Merge Sort: the `join` method

- The merging starts by comparing the smallest elements in each smaller sorted interval
- The smaller of these two elements is the smallest of all the elements in either subinterval
- The method `join` makes use of a temporary array, and it is to this array that the smaller element is moved
- The process is repeated with the remaining elements in the two smaller sorted intervals to find the next smallest element, and so forth

Merge Sort Code (1 of 3)

```
/**
 Class that realizes the divide-and-conquer sorting pattern and
 uses the merge sort algorithm.
*/
public class MergeSort
{

    /**
     Precondition: Interval a[begin] through a[end] of a have elements.
     Postcondition: The values in the interval have
     been rearranged so that a[begin] <= a[begin+1] <= ... <= a[end].
    */
    public static void sort(double[] a, int begin, int end)
    {
        if ((end - begin) >= 1)
        {
            int splitPoint = split(a, begin, end);
            sort(a, begin, splitPoint);
            sort(a, splitPoint + 1, end);
            join(a, begin, splitPoint, end);
        }//else sorting one (or fewer) elements so do nothing.
    }
```

Merge Sort Code (2 of 3)

```
private static int split(double[] a, int begin, int end)
{
    return ((begin + end)/2);
}

private static void join(double[] a, int begin, int splitPoint, int end)
{
    double[] temp;
    int intervalSize = (end - begin + 1);
    temp = new double[intervalSize];
    int nextLeft = begin; //index for first chunk
    int nextRight = splitPoint + 1; //index for second chunk
    int i = 0; //index for temp

    //Merge till one side is exhausted:
    while ((nextLeft <= splitPoint) && (nextRight <= end))
    {
        if (a[nextLeft] < a[nextRight])
        {
            temp[i] = a[nextLeft];
            i++; nextLeft++;
        }

        else
        {
            temp[i] = a[nextRight];
            i++; nextRight++;
        }
    }
```

Merge Sort Code (3 of 3)

```
     while (nextLeft <= splitPoint)//Copy rest of left chunk, if any.
        {
            temp[i] = a[nextLeft];
            i++; nextLeft++;
        }

        while (nextRight <= end) //Copy rest of right chunk, if any.
        {
            temp[i] = a[nextRight];
            i++; nextRight++;
        }

        for (i = 0; i < intervalSize; i++)
            a[begin + i] = temp[i];
    }

}
```

Merge Sort Demo

```
public class MergeSortDemo
{
    public static void main(String[] args)
    {
        double[] b = {7.7, 5.5, 11, 3, 16, 4.4, 20, 14, 13, 42};

        System.out.println("Array contents before sorting:");
        int i;
        for (i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println( );

        MergeSort.sort(b, 0, b.length-1);
        System.out.println("Sorted array values:");
        for (i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println( );
    }
}
```

Quick Sort

- In the *quick sort* realization of the sorting pattern, the definition of **`split`** is quite sophisticated, while **`join`** is utterly simple
  - First, a value called the *splitting value* is chosen
    - We do this arbitrarily but other methods to select this value may be employed
  - The elements in the array are rearranged:
    - All elements less than or equal to the splitting value are placed at the front of the array
    - All elements greater than the splitting value are placed at the back of the array
    - The splitting value is placed in between the two

Quick Sort

- Note that the smaller elements are not sorted, and the larger elements are not sorted
  - However, all the elements before the splitting value are smaller than any of the elements after the splitting value
- The smaller elements are then sorted by a recursive call, as are the larger elements
- Then these two sorted segments are combined
  - The `join` method actually does nothing

Quick Sort Code (1 of 3)

```
public class QuickSort
{
    /**
     Precondition: Interval a[begin] through a[end] of a have elements.
     Postcondition: The values in the interval have
     been rearranged so that a[begin] <= a[begin+1] <= ... <= a[end].
    */
    public static void sort(double[] a, int begin, int end)
    {
        if ((end - begin) >= 1)
        {
            int splitPoint = split(a, begin, end);
            sort(a, begin, splitPoint);
            sort(a, splitPoint + 1, end);
            join(a, begin, splitPoint, end);
        }//else sorting one (or fewer) elements so do nothing.
    }

    private static int split(double[] a, int begin, int end)
    {
        double[] temp;
        int size = (end - begin + 1);
        temp = new double[size];

        double splitValue = a[begin];
        int up = 0;
        int down = size - 1;
```

Quick Sort Code (2 of 3)

```
    //Note that a[begin] = splitValue is skipped.
    for (int i = begin + 1; i <= end; i++)
    {
        if (a[i] <= splitValue)
        {
            temp[up] = a[i];
            up++;
        }
        else
        {
            temp[down] = a[i];
            down--;
        }
    }

    //0 <= up = down < size

    temp[up] = a[begin]; //Positions the split value, spliV.

    //temp[i] <= splitValue for i < up
     // temp[up] = splitValue
     // temp[i] > splitValue for i > up

    for (int i = 0; i < size; i++)
        a[begin + i] = temp[i];

    return (begin + up);
}
```

Quick Sort Code (3 of 3)

```
 private static void join(double[] a, int begin,
                          int splitPoint, int end)
    {
        //Nothing to do.
    }

}

public class QuickSortDemo
{
    public static void main(String[] args)
    {
        double[] b = {7.7, 5.5, 11, 3, 16, 4.4, 20, 14, 13, 42};

        System.out.println("Array contents before sorting:");
        int i;
        for (i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println( );

        QuickSort.sort(b, 0, b.length-1);
        System.out.println("Sorted array values:");
        for (i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println( );
    }
}
```

Restrictions on the Sorting Pattern

- Like all patterns, the sorting pattern has some restrictions on where it applies
  - It applies only to types for which the **`<`** operator is defined
  - It applies only to sorting into increasing order
- The pattern can be made more general, however
  - The **`<`** operator can be replaced with a **`boolean`** valued method called **`compare`**
  - The **`compare`** method would take two arguments of the base type of the array, and return **`true`** or **`false`** based on the comparison criteria

Efficiency of the Sorting Pattern

- The most efficient implementations of the sorting pattern are those for which the **`split`** method divides the array into two substantial size chunks
  - The merge sort **`split`** divides the array into two roughly equal parts, and is very efficient
  - The quick sort **`split`** may or may not divide the array into two roughly equal parts
    - When it does not, its worst-case running time is not as fast as that of merge sort

Efficiency of the Sorting Pattern

- The selection sort algorithm (from Chapter 5) divides the array into two pieces: one with a single element, and one with the rest of the array interval
  - Because of this uneven division, selection sort has a poor running time
  - However, it is simple

Pragmatics and Patterns

- Patterns are guides, not requirements
  - It is not necessary to follow all the fine details
- For example, quick sort was described by following the sorting pattern exactly
  - Notice that, despite the fact that method calls incur overhead, the quick sort `join` method does nothing
  - In practice calls to `join` would be eliminated
  - Other optimizations can also be done once the general pattern of an algorithm is clear