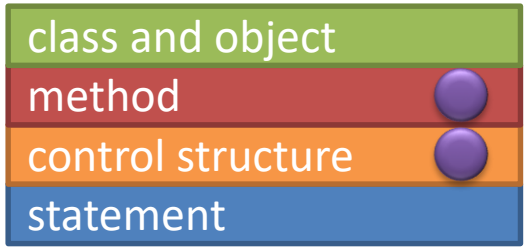
Sorting & searching arrays

Week 9



- Arrays
- Methods for working with arrays
- 10 Managing Collections with Arrays
  15 Sorting & Searching

Dr James Montgomery, james.montgomery@utas.edu.au

Tasks starting this week

9.1PP Divide and Conquer

- Given a program description, decide how to break down the design into methods and then implement it
- Draw a structure chart of your design
- The task includes lots of advice about implementing the functionality so you can focus on functional decomposition
  - The functionality has been chosen so that it is *not* algorithmically complex
  - Lots of library code can be used

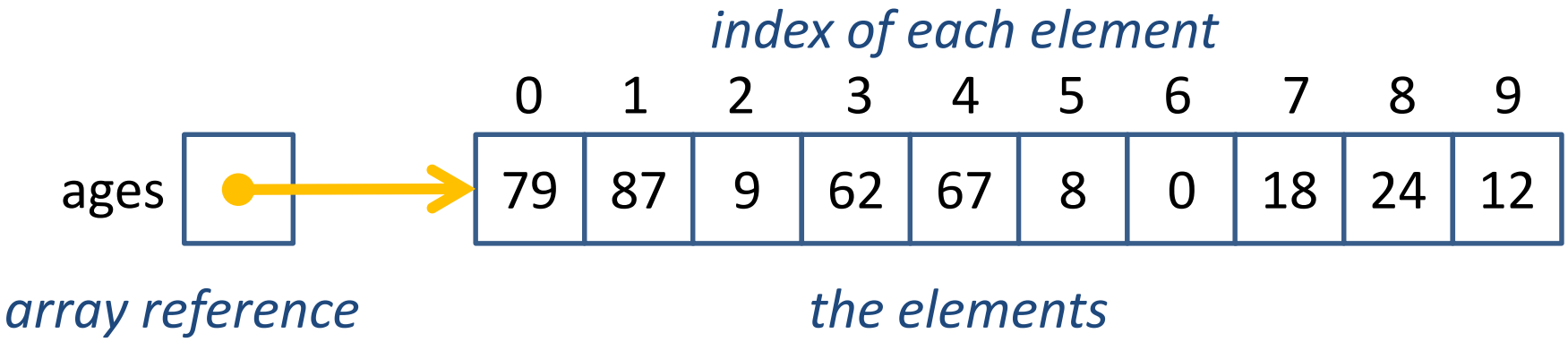
9.2CR Coding with Inheritance

- Discover how you can *extend* an existing class and build on its functionality

A reminder about arrays

An array is an ordered (and indexed) list of values of the same type (primitive or object)

Example: a list of 10 integer ages



A reminder about arrays

Declare an array reference

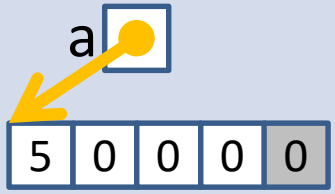
- syntax: *type*[] *identifier*;
- example: int[] a;

Allocate space

- syntax: *identifier* = new *type*[*size*];
- example: a = new int[5];

Access a specific element

- syntax: *identifier*[*index*];
- examples:
  - a[0] = 5;
  - int x = a[4];



And…

- *array*.length is length of array, as in a.length
- Array contents can be modified by methods

The good and bad of arrays

Advantages

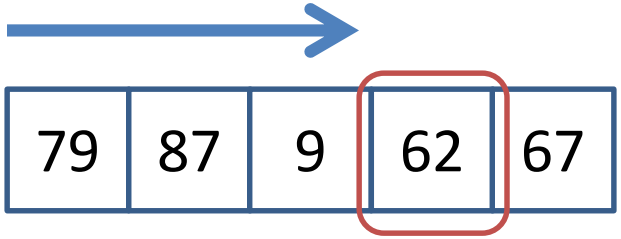
- Keep related items together
- Implicit ordering
  - Easy to traverse all elements
- Can pass as parameter & modify contents
- ‘random’/direct access to any element

Disadvantages

- Items must all be the same type
- Size fixed at instantiation
- Large arrays slow to traverse

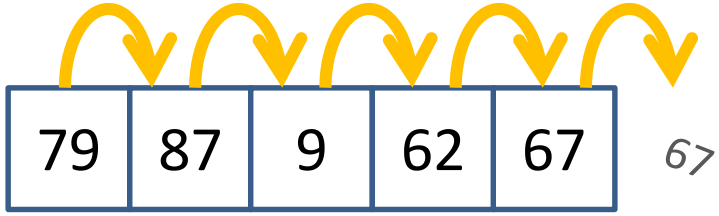
Common tasks with arrays

Process every element (e.g., display, fill, sum)

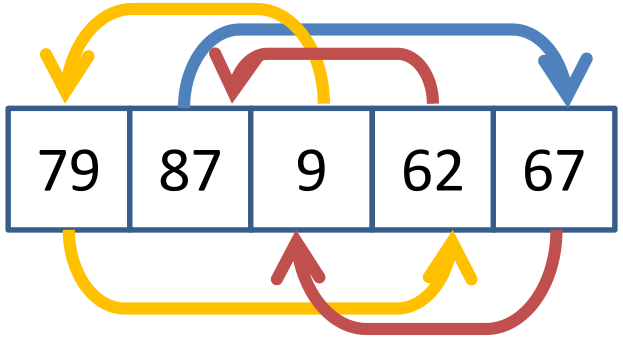


Rearrange

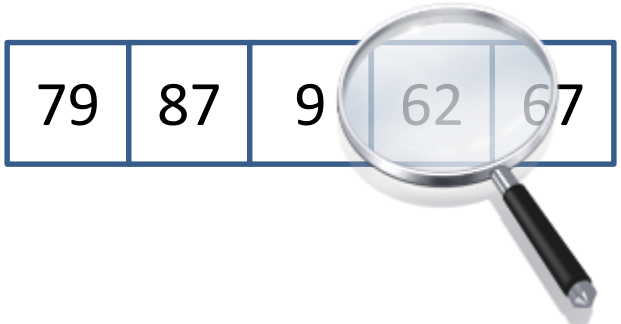
- move elements along



- into sorted order



Search



Sources of standard algorithms

Algorithm

- Standard text books
- Their references
- Wikipedia (actually quite good for this)

Implementation

- Write code each time
- Prepare a library (e.g. our ArrayRoutines class)
- Java standard library

Useful Java library methods

- System class


```
public static arraycopy(Object source, int srcPos,
                        Object dest, int destPos, int size)
```

(source and dest must be arrays to use this)

- Arrays class


```
public static int binarySearch(int[] array, int key)
public static char binarySearch(char[] array, char k)
public static boolean equals(int[] a1, int[] a2)
public static void sort(double[] array)
public static void fill(int[] a, int val)
public static void fill(int[] a,int from,int to,int val)
```

In the future, most of the time you will use the java.util.Arrays class

Sorting

The process of arranging a list of items into a particular order

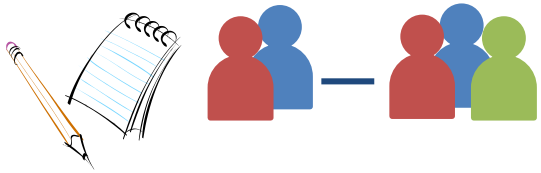
- must be some value on which the order is based

Many algorithms for sorting a list of items

- These vary in their efficiency

We will examine one specific algorithm: Insertion Sort

Insertion Sort



Basic approach

1. pick any item and insert it into its proper place in a sorted sublist
2. repeat until all items have been inserted

In more detail:

1. consider the *first item* to be a sorted sublist (of one item)
2. insert the *second item* into the sorted sublist, shifting items as necessary to make room to insert the new addition
3. repeat until all values are inserted into their proper position

Insertion Sort example

```
  original:
    3   9   6   1   2
"insert" 9:
    3   9   6   1   2
"insert" 6:
    3   6   9   1   2
"insert" 1:
    1   3   6   9   2
"insert" 2:
    1   2   3   6   9
```

Example code in `ArrayRoutines.java` as `insertionSort()`

Demonstration in `InsertSortTest.java`

Sorting Objects

Array elements can be primitive types *or* objects

Primitive types (except **boolean**) have a *natural* order

No guaranteed natural order for objects (must be defined by author of the class)

Strings have a **compareTo()** method

`str1.compareTo(str2)`

returns a value more than 0 if `str1` is alphabetically after `str2`

Some other classes also have compareTo()

Other sorting algorithms

Many different sorting algorithms

- vary in their **efficiency**
- efficiency could be measured as ‘number of comparisons’ or ‘number of swaps’

Example other algorithm: Selection Sort

1. find the smallest value in the list
2. switch it with the value in the first position
3. find the next smallest value in the list
4. switch it with the value in the second position
5. repeat until all values are placed

Code is in L&L (and in `ArrayRoutines.java`)

Searching

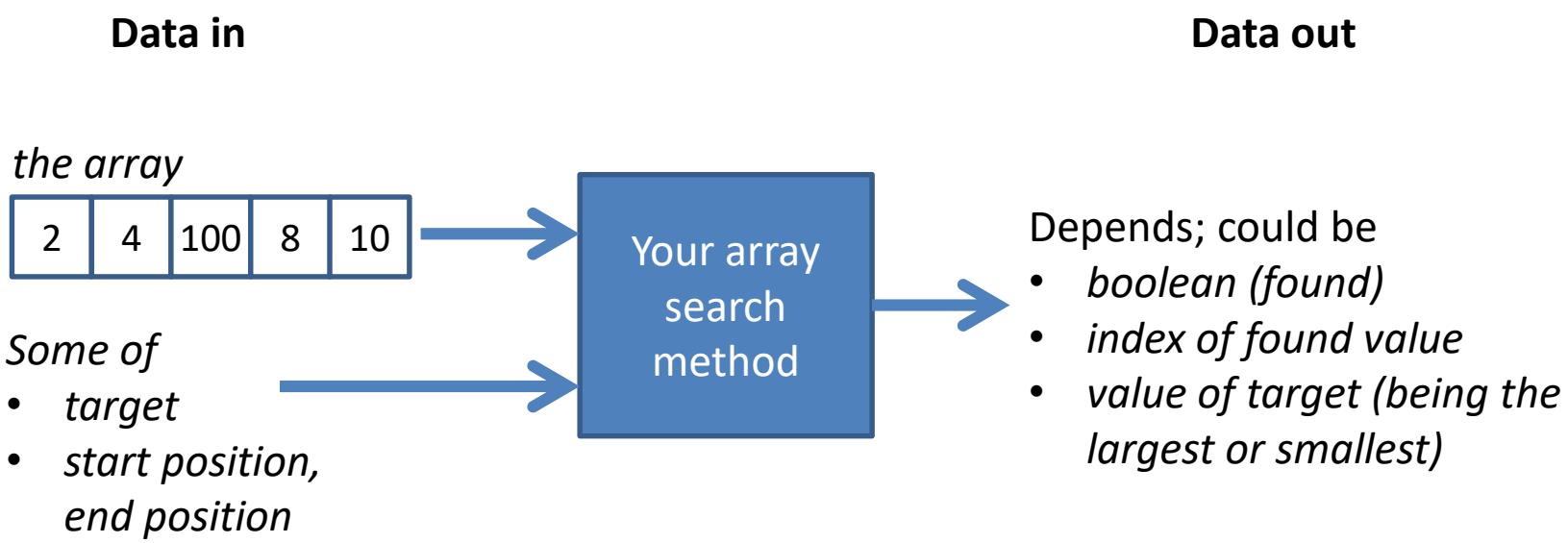
What are we searching for? (depends on problem)

- A particular value in the array
- The largest value
- The smallest value
- A value that matches some condition

What do we want to know? (depends on problem)

- **boolean** (yes/no answer)
- Actual value
- Position of value

Writing methods to search an array



The algorithm: Many alternatives; will consider two

An example search problem

Problem: Is a particular value found in a filled array?

Start state

- Have an array filled with values
- Have a ‘target’ value we are searching for

Goal state

- Know position (index) in the array of the target

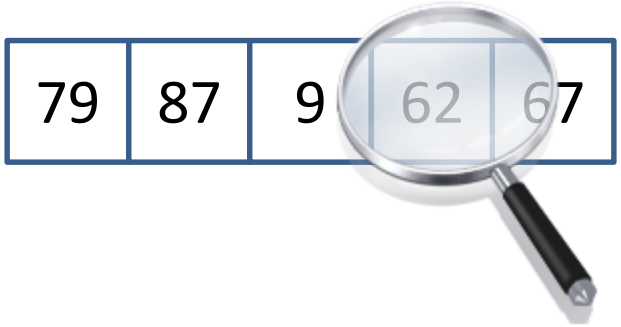
Issue: How to indicate that the target is not there?

- Return some impossible index value (–1 often used)

Linear Search

Base algorithm: traverse an array

- elements of array can be in *any* order
- can stop when target found

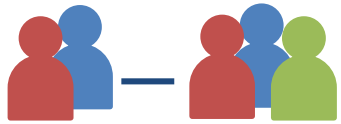


Linear Search Algorithm

1. set index to –1, found to false
2. start at start position (0 default)
3. while not at end and target not found
   - if current element same as target
     - set index to current
     - set found to true
   - else
     - move to next element
4. return index

See `linearSearch()` in `ArrayRoutines.java`

Binary Search (nothing to do with 0s & 1s)



Task: I have picked a number between 1 and 1,000,000. Can you guess it in 20 questions?

Constraints: Questions can only be

- is it smaller than __ ?
- is it __ ?

Answer will be **yes** or **no**

Let's play…

Binary Search

- ✓ Higher speed than linear search
- ✓ Eliminates half the elements being searched after each comparison

Issues

- ✘ array elements MUST be in sorted order
- – how to calculate index of middle element (calculate index of midpoint in array initially)
- – how to determine when the array can no longer be halved

Binary Search algorithm

1. set index to –1, found to false
2. while there are elements left to search and target not found
   - make current element middle of array to search
   - if current element is target
     - set found to true
     - set index to current index of current
   - else
     - search the half of the array that *might* contain the target

Question: how to determine whether there are elements left to search?

Implementation in ArrayRoutines.java

```
public int binarySearch(int[] a, int t) {
   boolean found = false;
   int low = 0;
   int high = a.length - 1;
   int middle;
   int index = -1;

   while (low <= high && !found) {
        middle = (low + high)/2;
        //is middle the target t?
        //if so
               //stop searching
               //index becomes middle
        //if not search likely part of array
        //(details next slide)
   }

   return index;
}
```

Binary search inner loop

```
while (low <= high && !found) {
   middle = (low + high)/2;

   if (a[middle] == t) { //is middle the target t?
       found = true;      //if so, stop searching
       index = middle;    //index becomes middle
   } else {
       //search likely part of array
       if (t < a[middle]) { //t can only be before middle
              high = middle - 1;
       } else {                //t can only be after middle
              low = middle + 1;
       }
    }
}
```