Chapter 10: Arrays
After studying this chapter, you should be able to:

- Store data in an array, access a single element, process all elements, search for a particular element, and put the elements in order
- Declare, allocate, and initialize an array
- Handle changing numbers of elements in an array, including inserting a new element and deleting an existing one
- Enlarge or shrink the size of an array
- Manipulate data stored in a multi-dimensional array
10.1 Using Arrays (1/2)
Wanted:

- A way to easily work with 1,500 (or many more!) persons collected from [www.myspace.com](http://www.myspace.com).
- Each person represented as an instance of the **Person** class:

| Person | 
|---|---|
| - String name  
- int id  
- Gender gender  
- int age  
- String city  
- String province  
- String country  
- DateTime lastLogon  
- ??? friends  
+ Person(Scanner in)  
+ String getName()  
+ int getID()  
+ Gender getGender()  
... |
### 10.1.1: Visualizing An Array

```plaintext
<table>
<thead>
<tr>
<th>persons</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
</tr>
<tr>
<td>[0]</td>
</tr>
<tr>
<td>[1]</td>
</tr>
<tr>
<td>[2]</td>
</tr>
<tr>
<td>[3]</td>
</tr>
<tr>
<td>[4]</td>
</tr>
<tr>
<td>[5]</td>
</tr>
<tr>
<td>[6]</td>
</tr>
<tr>
<td>[7]</td>
</tr>
</tbody>
</table>
```

- Steve, MALE, 16, St. Cat
- Ken, MALE, 18, Niagara-on
- Beth, FEMALE, 17, St. Cat
- Kathleen, FEMALE, 23, Tim
- Roydyn, MALE, 21, Fonth
- Kala, FEMALE, 18, Niaga
- Ali, MALE, 19, Boston, MS
- Zaki, FEMALE, 21, Kitch
Person p3 = new Person(...);
... persons ...    // An array of Person objects

System.out.println(p3.getName());
System.out.println(persons[3].getName);

if (p3.getGender() == Gender.FEMALE)
{ System.out.println(p3.getName() + " is female.");
}
if (persons[3].getGender() == Gender.FEMALE)
{ System.out.println(persons[3].getName() + " is female.");
}

persons[8].addFriend(p3);
p3.addFriend(persons[8]);

p3 = new Person(...);
persons[3] = new Person(...);
public class PersonList extends Object
{
    ... persons ...

    /** Swap the person object at index a with the object at index b. */
    public void swap(int a, int b)
    {
        Person temp = this.persons[a];
        this.persons[a] = this.persons[b];
        this.persons[b] = temp;
    }
}

Assume that `swap(1, 2)` has been called.

Person temp = this.persons[a];
this.persons[a] = this.persons[b];

// After the swap method finishes
public class PersonList extends Object
{
    ... persons ...

    /* Print the name and number of friends for every person in the array. */
    public void printBasicInfo()
    {
        for(int i = 0; i < this.persons.length; i += 1)
        {
            Person p = this.persons[i];
            System.out.println(p.getName() + " has " +
                                p.getNumFriends + "friends");
        }
    }

    /** Calculate the average number of friends */
    public double calcAverageNumberOfFriends()
    {
        int sumFriends = 0;
        for( int i = 0; i < this.persons.length; i += 1)
        {
            Person p = this.persons[i];
            sumFriends = sumFriends + p.getNumFriends();
        }
        return (double) sumFriends / this.persons.length;
    }
}
public class PersonList extends Object
{
    ... persons ...

    /** Calculate the average number of friends */
    public double calcAverageNumberOfFriends()
    {
        int sumFriends = 0;

        for( int i = 0; i < this.persons.length; i += 1)
        {
            Person p = this.persons[i];
            sumFriends = sumFriends + p.getNumFriends();
        }

        return (double) sumFriends / this.persons.length;
    }

    /** Calculate the average number of friends using a “foreach loop” */
    public double calcAverageNumberOfFriends()
    {
        int sumFriends = 0;

        for( Person p : this.persons )
        {
            sumFriends = sumFriends + p.getNumFriends();
        }

        return (double) sumFriends / this.persons.length;
    }
}
for (each element in the array)
{
    if (the element meets some criteria)
    {
        process the element
    }
}

public class PersonList extends Object
{
    ... persons ...

    /** Count the number of minors (persons less than 18 years old). */
    public int countMinors()
    {
        int count = 0;
        for(int i = 0; i < this.persons.length; i += 1)
        {
            if (this.persons[i].getAge() < 18)
            {
                count += 1;
            }
        }
        return count;
    }
}
Searching uses some identifying information—name, telephone number, ID number—to find the corresponding object in the array. The identifying information is often called the key.

Find the person with ID 107733.

```java
public class PersonList extends Object
{
  … persons …

  /** Find the person with the given id; null if not found. */
  public Person search(int id)
  {
    for(int i=0; i<this.persons.length; i+= 1)
    {
      Person p = this.persons[i];
      if (p.getID() == id)
      {
        return p;           // success! Return, exiting loop.
      }
    }
    return null;          // not found
  }
}
```
One way of structuring the search is to ask when we’re done:

- Found the correct element (success!)
- Reached the end of the list (failure)

```java
public class PersonList extends Object {
    ... persons ...

    /** Find the person with the given id; null if not found. Use a traditional loop */
    public Person search2(int id)
    {
        int i = 0;
        while (i < this.persons.length && this.persons[i].getID() != id)
        {
            i += 1;
        }

        Person answer = null;
        if (i < this.persons.length)
        {
            answer = this.persons[i];
        }
        return answer;
    }
}
```
An extreme element has the most of something or the least of something. The most age, the longest time since the last login, the “smallest” name (first in dictionary order), etc.

remember the first element as the best seen so far
for (each remaining element in the array)
{ if (the current element is better than the best seen so far)
  { remember the current element as the best seen so far
  }
}
return best element seen so far
public class PersonList extends Object
  {
    ... persons ...

    /** Find the youngest person. */
    public Person findYoungestPerson()
    {
      Person youngestSoFar = this.persons[0];

      for(Person currentPerson : this.persons)
      {
        if (currentPerson.getAge() < youngestSoFar.getAge())
        {
          youngestSoFar = currentPerson;
        }
      }

      return youngestSoFar;
    }

  }

Quick Quiz

What does findYoungestPerson return for the array shown on the right?
Sorting an array puts all the elements in order by age, name, or some other criteria.

Selection Sort is one of many algorithms to sort an array. It builds on three patterns we’ve already seen:

- Process All Elements
- Find an Extreme
- Swap Two Elements

We’ll sort the **persons** array by name.

<table>
<thead>
<tr>
<th>persons</th>
<th>Faizel, MALE, 16, St. Ca</th>
<th>Ellen, FEMALE, 18, Niag</th>
<th>Ali, MALE, 19, Boston,</th>
<th>Greg, MALE, 21, Fonth</th>
<th>Beth, FEMALE, 17, St.</th>
<th>Doug, MALE, 18, Niagara-</th>
<th>Cathy, FEMALE, 23, Tim</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Persons[]</th>
<th>Faizel, MALE, 16, St. Ca</th>
<th>Ellen, FEMALE, 18, Niag</th>
<th>Ali, MALE, 19, Boston,</th>
<th>Greg, MALE, 21, Fonth</th>
<th>Beth, FEMALE, 17, St.</th>
<th>Doug, MALE, 18, Niagara-on</th>
<th>Cathy, FEMALE, 23, Tim</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Divide the array into the part that’s already sorted (dark background) and the part that isn’t (light background).

Repeatedly extend the sorted part of the array by:

- Finding the smallest element in the unsorted part of the array.
- Swapping it with the first element in the unsorted part of the array.
- Extending the sorted part of the array.
for (each position in the array except the last)
{
    find the element that should go in this position
    swap that element with the element currently there
}
```java
public class PersonList extends Object {
    ... persons ...

    /** Sort the list of persons in alphabetical order by name. */
    public void sort() {
        for (int firstUnsorted=0; firstUnsorted<this.persons.length-1;
             firstUnsorted++) {
            int extremeIndex = this.findExtreme(firstUnsorted);
            this.swap(firstUnsorted, extremeIndex);
        }
    }

    /** Swap the elements at indices a and b. */
    private void swap(int a, int b) {
        Person temp = this.persons[a];
        this.persons[a] = this.persons[b];
        this.persons[b] = temp;
    }
}
```
/** Find the extreme element in the unsorted portion of the array.  
* @param indexToStart the smallest index in the unsorted portion of the array  */
private int findExtreme(int indexToStart)  
{  
    int indexBestSoFar = indexToStart;
    String nameBestSoFar = this.persons[indexBestSoFar].getName();

    for (int i=indexToStart+1; i<this.persons.length; i++)
    {  
        String currPersonName = this.persons[i].getName();

        if (currPersonName.compareTo(nameBestSoFar) < 0)
        {  
            indexBestSoFar = i;
            nameBestSoFar = this.persons[i].getName();
        }
    }

    return indexBestSoFar;
}
public class PersonList extends Object
{
   … persons …

   /** Sort the persons array in increasing order by age */
   public void sortByAge()
   {
      for (int firstUnsorted=0; firstUnsorted<this.persons.length-1;
         firstUnsorted++)
      {
         // find the youngest unsorted person
         int extremelIndex = firstUnsorted;
         for (int i=firstUnsorted + 1; i<this.persons.length; i++)
         {
            if (this.persons[i].getAge() < this.persons[extremelIndex].getAge())
            {
               extremelIndex = i;
            }
         }

         // swap the youngest unsorted person with the person at firstUnsorted
         Person temp = this.persons[extremelIndex];
         this.persons[extremelIndex] = this.persons[firstUnsorted];
         this.persons[firstUnsorted] = temp;
      }
   }
}
## Comparing Arrays and Files

<table>
<thead>
<tr>
<th></th>
<th>Arrays</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store many records?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Where is the data?</td>
<td>In memory</td>
<td>On a disk drive</td>
</tr>
<tr>
<td>Access speed?</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Random access?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cost?</td>
<td>More expensive</td>
<td>Really cheap</td>
</tr>
<tr>
<td>When power is off?</td>
<td>Information lost</td>
<td>Information retained</td>
</tr>
</tbody>
</table>
Step 1: Declare the array

Step 2: Allocate space

Step 3: Initialize each element
**Declaration**

```java
public class PersonList extends Object {
    private Person[] persons;
}
```

**Allocation**

```java
... this.persons = new Person[8];
```

Decide how large when the program runs:

```java
public class PersonList extends Object {
    private Person[] persons;

    private void createArray() {
        Scanner in = new Scanner(System.in);
        System.out.print("How many persons: ");
        int numPersons = in.nextInt();
        this.persons = new Person[numPersons];
    }
}
```
public class PersonList extends Object
{
  private Person[ ] persons = new Person[3];

  public PersonList()
  {
    super();
    this.persons[0] = new Person(10059, "Jacob", Gender.MALE, ...);
    this.persons[1] = new Person(10060, "Emily", Gender.FEMALE, ...);
    this.persons[2] = new Person(10061, "Pat", Gender.FEMALE, ...);
  }
}
If the records are stored in a file…

```java
public class PersonList extends Object {
    private Person[] persons;

    public PersonList(String fileName) {
        super();
        int count = 0;
        Scanner in = this.openFile(fileName);
        while (in.hasNextLine()) {
            Person p = new Person(in);
            count++;
        }
        in.close();
        this.persons = new Person[count];

        in = this.openFile(fileName);
        for (int i=0; i<count; i++) {
            this.persons[i] = new Person(in);
        }
        in.close();
    }
}
```

Count the number of records in the file so we know how much space to allocate for the array.

Allocate the array.

Read the data again, creating the objects, and initializing the array with them.
public class PersonList extends Object {
    private Person[] persons;

    public PersonList(String fileName) {
        super();
        Scanner in = this.openFile(fileName);

        int count = in.nextInt();
        in.nextLine();

        this.persons = new Person[count];

        for (int i=0; i<count; i++) {
            this.persons[i] = new Person(in);
        }

        in.close();
    }
}

Date file:
3
10005 Joshua MALE …
10007 Ellie FEMALE …
10008 Pat FEMALE …

Find out how many records.
Allocate the array.
Read the data again, creating the objects, and initializing the array with them.
Suppose we want all the persons who live in a particular city, such as St. Catharines.

```java
public class PersonList extends Object {
    private Person[] persons;

    public Person[] extractSubset(String city) {
        size = count number of elements in the subset
        subset = a new array to store size elements
        fill subset with the appropriate objects
        return subset
    }
}
```

Usage:
```java
Person[] stCath = pList.extractSubset("St. Catharines");
System.out.println("There are "+ stCath.length + " people living in St. Catharines");
for (Person p : stCath) {
    System.out.println(p);
}
```
public class PersonList extends Object
{
    private Person[ ] persons;

    public Person[ ] extractSubset(String city)
    {
        int ssSize = this.countSubset(city);
        Person[ ] subset = new Person[ssSize];
        this.fillSubset(subset, city);
        return subset;
    }
}
Person[]

<table>
<thead>
<tr>
<th>length</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td></td>
</tr>
</tbody>
</table>

Person[]

<table>
<thead>
<tr>
<th>length</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td></td>
</tr>
<tr>
<td>[5]</td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td></td>
</tr>
<tr>
<td>[7]</td>
<td></td>
</tr>
</tbody>
</table>

ssPos 1

arrPos 2

Person

Steve, MALE, 16, Niagara
Beth, FEMALE, 22, Timmons
Kathleen, FEMALE, 22, St. Catharines
Roydyn, MALE, 19, Boston
Kala, FEMALE, 23, Kitchener
Ali, MALE, 19, Niagara
Zaki, MALE, 21, St. Catharines
Susan, FEMALE, 18, St. Catharines
public class PersonList extends Object
{
    private Person[ ] persons;

    public Person[ ] extractSubset(String city)
    {
        ...
    }

    /** Fill the subset array with Person objects matching the given city */
    /** @param subset The array to fill with elements belonging to the subset. */
    /** @param city The city from which all members of the subset come */

    private void fillSubset(Person[ ] ss, String city)
    {
        int ssPos = 0; // position within the subset
        int arrPos = 0; // position within the array
        while (ssPos < ss.length)
        {
            Person p = this.persons[arrPos];
            if (p.getCity().equalsIgnoreCase(city))
            {
                ss[ssPos] = p;
                ssPos++;
            }
            arrPos++;
        }
    }
}
A partially filled array:

- is an array with two areas, one that has elements in it and one that does not.
- has an auxiliary variable to say how many of the elements are valid.
Modifying the Process All Pattern:

```java
for (int i = 0; i < this.size; i += 1)
    { Person p = this.persons[i];
      System.out.println(p.getName());
    }
```

Adding a person at the end of a partially filled array:

```java
public void add(Person p)
{ this.persons[this.size] = p;
  this.size += 1;
}
```

**Quick Quiz**

You can’t add elements to a partially filled array indefinitely. Eventually, it will fill up.

Modify `add` so that it will detect when the array is full and print an error message instead of adding the element.
10.4.1: Inserting into a Sorted Array

Original, sorted array.

Make a “hole” for the new element.

Insert the new element.
We can also add an element to an array with the following four steps:

Step 1: Allocate a new, larger array.

Step 2: Copy the contents of the old array to the larger array.
Step 3: Reassign the array reference (allowing the old array to be garbage collected).

Step 4: Add the new element.
public class PersonList extends Object
{
    private Person[ ] persons;

    public void add(Person p)
    {
        // Step 1: Allocate a larger array.
        Person[ ] larger = new Person[this.persons.length + 1];

        // Step 2: Copy element from the old array to the new, larger array.
        for (int i = 0; i < this.persons.length; i += 1)
        {
            larger[i] = this.persons[i];
        }

        // Step 3: Reassign the array reference.
        this.person = larger;

        // Step 4: Add the new element.
        this.persons[this.persons.length-1] = p;
    }
}
Comparing the time (in seconds) to insert 150,000 elements into an array that grows vs. inserting the same elements into a sufficiently large partially filled array.
public class PersonList extends Object {
    private Person[ ] persons;
    private int size = 0;

    public void add(Person p) {
        if (this.persons.length == this.size) {
            // The array is full – grow it.
            Person[ ] larger = new Person[this.persons.length * 2];

            for (int i = 0; i < this.persons.length; i += 1) {
                larger[i] = this.persons[i];
            }

            this.person = larger;
        }

        // Add the new element to the end of the array and increase the size.
        this.persons[this.size] = p;
        this.size += 1;
    }
}
<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Partially Filled</strong></td>
<td>• Allows add &amp; delete.</td>
<td>• Fixed maximum size.</td>
</tr>
<tr>
<td><strong>Arrays</strong></td>
<td>• Fast.</td>
<td>• Wasted space if a large array is sometimes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>needed.</td>
</tr>
<tr>
<td><strong>Growing</strong></td>
<td>• Allows add &amp; delete.</td>
<td>• Too much time to insert many elements.</td>
</tr>
<tr>
<td><strong>Arrays</strong></td>
<td>• No wasted space.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• No maximum size (other than that imposed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>by the computer).</td>
<td></td>
</tr>
<tr>
<td><strong>Combined</strong></td>
<td>• Allows add &amp; delete.</td>
<td>• Some wasted space.</td>
</tr>
<tr>
<td><strong>Approach</strong></td>
<td>• No maximum size.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Less wasted space.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Fast</td>
<td></td>
</tr>
</tbody>
</table>
Java also allows arrays of `int`, `double`, `boolean`, etc.

For example, each MySpace `Person` object has a list of the ID numbers of their friends.

```java
public class Person extends Object {
  private String name;
  ...
  private int[] friendIDs; // Declare

  public Person(Scanner in) {
    // Allocate
    int numFriends = in.nextInt();
    this.friendIDs = new int[numFriends];
    // Initialize
    for(int i=0; i<numFriends; i+=1) {
      this.friendIDs[i] = in.nextInt();
    }
  }
}
```

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>- String name</td>
</tr>
<tr>
<td>- int id</td>
</tr>
<tr>
<td>- Gender gender</td>
</tr>
<tr>
<td>- int age</td>
</tr>
<tr>
<td>- String city</td>
</tr>
<tr>
<td>- String province</td>
</tr>
<tr>
<td>- String country</td>
</tr>
<tr>
<td>- DateTime lastLogon</td>
</tr>
<tr>
<td>- int[] friendIDs</td>
</tr>
</tbody>
</table>

+Person(Scanner in)
+String getName()
+int getID()
+Gender getGender()
+int[] getFriendIDs()
Write a method that returns the number of friends that two people have in common.

```java
public class Person extends Object
{
    private final int id;
    private final String name;
    ...
    private int[] friendIDs; // Filled array.

    public int numFriendsInCommon(Person other)
    {
        int inCommon = 0;

        for (each friend in my list)
        {
            if (my friend is also a friend of other)
            {
                inCommon += 1;
            }
        }

        return inCommon;
    }
}
```
Case Study 1: # of Friends in Common (2/2)

```java
public class Person extends Object {
  private int[] friendIDs;  // Filled array.

  // Filled array.

  /** Return the number of friends other has in common with this person. */
  public int numFriendsInCommon(Person other) {
    int inCommon = 0;

    for (int i = 0; i < this.friendIDs.length; i += 1) {
      int myFriend = this.friendIDs[i];
      if (other.hasFriend(myFriend)) {
        inCommon += 1;
      }
    }

    return inCommon;
  }

  /** Return true if personID appears in the list of this person's friends; false otherwise. */
  private boolean hasFriend(int personID) {
    for (int i = 0; i < this.friendIDs.length; i += 1) {
      if (this.friendIDs[i] == personID) {
        return true;
      }
    }

    return false;
  }

  ... 
}
```
Suppose you wanted to know the distribution of friends – how many people have 0 friends, how many have one friend, two friends, etc. Such information could be used to make a bar chart such as the following:
public class PersonList extends Object
{
  private Person[ ] persons;
  
  public int[ ] friendDistribution()
  {
    int[ ] numFriends = new int[100];

    for(Person p : this.persons)
    {
      if (p.getNumFriends() < numFriends.length)
      {
        numFriends[p.getNumFriends()] += 1;
      }
      else
      {
        // grow the array instead?!?
        throw new Error(p.getName() + " has too many friends (" +
                       p.getNumFriends() + ").");
      }
    }
    
    return numFriends;
  }

  Usage:
  int[ ] friendDist = pList.friendDistribution();
  for(int i=0; i<friendDist.length; i++)
  {
    System.out.println((friendDist[i] + 
                         " people have " + i + " friends.");
  }
}
<table>
<thead>
<tr>
<th></th>
<th>United Way</th>
<th>Corporate Donations</th>
<th>Individual Donations</th>
<th>Fundraising</th>
<th>Govt. Grants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>0</td>
<td>3,000</td>
<td>6,915</td>
<td>0</td>
<td>15,500</td>
</tr>
<tr>
<td>Feb</td>
<td>0</td>
<td>2,125</td>
<td>4,606</td>
<td>0</td>
<td>5,500</td>
</tr>
<tr>
<td>Mar</td>
<td>0</td>
<td>2,000</td>
<td>5,448</td>
<td>0</td>
<td>5,500</td>
</tr>
<tr>
<td>Apr</td>
<td>0</td>
<td>3,000</td>
<td>4,833</td>
<td>13,983</td>
<td>15,500</td>
</tr>
<tr>
<td>May</td>
<td>20,569</td>
<td>2,000</td>
<td>6,091</td>
<td>0</td>
<td>5,500</td>
</tr>
<tr>
<td>Jun</td>
<td>0</td>
<td>8,000</td>
<td>4,867</td>
<td>0</td>
<td>5,500</td>
</tr>
<tr>
<td>Jul</td>
<td>0</td>
<td>3,000</td>
<td>4,196</td>
<td>0</td>
<td>15,500</td>
</tr>
<tr>
<td>Aug</td>
<td>0</td>
<td>2,550</td>
<td>4,736</td>
<td>0</td>
<td>5,500</td>
</tr>
<tr>
<td>Sep</td>
<td>0</td>
<td>2,000</td>
<td>4,305</td>
<td>0</td>
<td>5,500</td>
</tr>
<tr>
<td>Oct</td>
<td>0</td>
<td>3,000</td>
<td>5,286</td>
<td>32,254</td>
<td>15,500</td>
</tr>
<tr>
<td>Nov</td>
<td>0</td>
<td>2,000</td>
<td>6,834</td>
<td>0</td>
<td>5,500</td>
</tr>
<tr>
<td>Dec</td>
<td>9,351</td>
<td>2,000</td>
<td>7,459</td>
<td>0</td>
<td>5,500</td>
</tr>
</tbody>
</table>

```java
int[][] income = new int[12][5];
```
income[4][1] += 1000;
Printing Every Element

for ( each row in the array )
{
    for ( each column in the row )
    {
        print the data at the intersection of the row and column
    }
}

public class Income extends Object
{
    private int[][] income;

    ...

    /** Print the income chart. */
    public void printIncomeChart()
    {
        for (int row = 0; row < this.income.length; row += 1)
        {
            for (int col = 0; col < this.income[row].length; col += 1)
            {
                System.out.print(this.income[row][col] + "\t");
            }
            System.out.println();
        }
    }
}
**Sum Every Element**

```java
int total = 0;
for (each row in the array )
{ for (each column in the row )
    { if add the data at the intersection of the row and column to total
    }
}

public class Income extends Object
{ private int[ ][ ] income;
    ...

    /** Calculate the total income for the year. */
    public int getTotalIncome()
    { int total = 0;
      for (int row = 0; row < this.income.length; row += 1)
      { for (int col = 0; col < this.income[row].length; col += 1)
          { total += this.income[row][col];
          }
      }
      return total;
    }
}
Summing a Column

```java
int total = 0;
for (each row in the array )
{   add the data at intersection of the row and specified column to total
}
```

```java
public class Income extends Object
{  private int[ ][ ] income;

    **/ ** Calculate the total income for a given category for the year. */
    public int getTotalByCategory(int columnNum)
    {  int total = 0;
        for (int row = 0; row < this.income.length; row += 1)
        {  total = total + this.income[r][columnNum];
        }
        return total;
    }
}
```
public class Income extends Object
{
    private int[][] income;
    ...

    public Income(Scanner in)
    {
        super();
        // Get the size of the array.
        int rows = in.nextInt();
        int cols = in.nextInt();
        in.nextLine();

        // Allocate the array.
        this.income = new int[rows][cols];

        // Fill the array.
        for (int r = 0; r < this.income.length; r++)
        {
            for (int c = 0; c < this.income[r].length; c++)
            {
                this.income[r][c] = in.nextInt();
                in.nextLine();
            }
        }
    }
}
10.6.3: Arrays of Arrays

```
<table>
<thead>
<tr>
<th>int[]</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0</td>
</tr>
<tr>
<td>[1]</td>
<td>3,000</td>
</tr>
<tr>
<td>[2]</td>
<td>4,833</td>
</tr>
<tr>
<td>[3]</td>
<td>13,983</td>
</tr>
<tr>
<td>[4]</td>
<td>15,500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int[]</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>5</td>
</tr>
<tr>
<td>[1]</td>
<td>0</td>
</tr>
<tr>
<td>[2]</td>
<td>3,000</td>
</tr>
<tr>
<td>[3]</td>
<td>6,915</td>
</tr>
<tr>
<td>[4]</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int[]</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>5</td>
</tr>
<tr>
<td>[1]</td>
<td>0</td>
</tr>
<tr>
<td>[2]</td>
<td>2,125</td>
</tr>
<tr>
<td>[3]</td>
<td>4,606</td>
</tr>
<tr>
<td>[4]</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int[]</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>5</td>
</tr>
<tr>
<td>[1]</td>
<td>0</td>
</tr>
<tr>
<td>[2]</td>
<td>2,550</td>
</tr>
<tr>
<td>[3]</td>
<td>4,736</td>
</tr>
<tr>
<td>[4]</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int[]</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>5</td>
</tr>
<tr>
<td>[1]</td>
<td>0</td>
</tr>
<tr>
<td>[2]</td>
<td>9,351</td>
</tr>
<tr>
<td>[3]</td>
<td>2,000</td>
</tr>
<tr>
<td>[4]</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int[]</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>5</td>
</tr>
<tr>
<td>[1]</td>
<td>0</td>
</tr>
<tr>
<td>[2]</td>
<td>20,569</td>
</tr>
<tr>
<td>[3]</td>
<td>2,000</td>
</tr>
<tr>
<td>[4]</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int[]</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>5</td>
</tr>
<tr>
<td>[1]</td>
<td>0</td>
</tr>
<tr>
<td>[2]</td>
<td>15,500</td>
</tr>
<tr>
<td>[3]</td>
<td>0</td>
</tr>
<tr>
<td>[4]</td>
<td>5,500</td>
</tr>
</tbody>
</table>
```
Showing these images in quick succession will cause the eyes to appear to roll.

- Already know how to use `ImageComponent` to get an image from a file and display it (Section 9.8.2).
- Put images into an array so we can use an index to display them in sequence.
- But…
  - need to sleep for a bit (about 100ms) so the user can see the current image.
  - need to allow other things to happen (like selecting ‘quit’). Therefore… use a thread to update the index and call `repaint`. 

}````
import javax.swing.*;
import java.awt.*;

/** Instances of AnimateImage show a sequence of images to produce an animation.
 * @author Byron Weber Becker */
public class AnimateImage extends JComponent implements Runnable {
    private int NUM_IMAGES = 6;
    private ImageIcon[] images;
    private int currentImage = 0;

    /** Construct a new image animation component, loading all the images. */
    public AnimateImage() {
        super();
        // Allocate and initialize array of images
        this.images = new ImageIcon[NUM_IMAGES];
        for (int i = 0; i < NUM_IMAGES; i++) {
            String fileName = "Happy" + i + "\.gif";
            this.images[i] = new ImageIcon(fileName);
        }

        this.setPreferredSize(new Dimension(this.images[0].getIconWidth(),
            this.images[0].getIconHeight()));
    }

    public void run() {
        //滚动显示图像
        while (true) {
            currentImage = (currentImage + 1) % NUM_IMAGES;
            repaint();
        }
    }
}

10.7: GUI: Animation (2/4)
/*
  @author Byron Weber Becker */
public class AnimateImage extends JComponent implements Runnable {
    private int NUM_IMAGES = 6;
    private ImageIcon[] images;
    private int currentImage = 0;

    /** Construct a new image animation component, loading all the images. */
    public AnimateImage() {
        super();
        // Allocate and initialize array of images
        this.images = new ImageIcon[NUM_IMAGES];
        for (int i = 0; i < NUM_IMAGES; i++) {
            String fileName = "Happy" + i + "\.gif";
            this.images[i] = new ImageIcon(fileName);
        }

        this.setPreferredSize(new Dimension(this.images[0].getIconWidth(),
            this.images[0].getIconHeight()));
    }

    public void run() {
        //滚动显示图像
        while (true) {
            currentImage = (currentImage + 1) % NUM_IMAGES;
            repaint();
        }
    }
}
/** Paint the current image on the screen. */
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    Image img = this.images[this.currentImage].getImage();
    g.drawImage(img, 0, 0, null);
}

/** Run the animation. */
public void run()
{
    while (true)
    {
        // Advance to the next image in the sequence.
        this.currentImage = (this.currentImage + 1) % this.images.length;
        this.repaint();
        try
        {
            // Give the user a chance to see the image.
            Thread.sleep(100);
        } catch (InterruptedException ex)
        {
            // ignore
        }
    }
}
```java
import javax.swing.*;

public class Main extends Object
{
    public static void main(String[ ] args)
    {
        // Create two animated components.
        AnimateImage anim1 = new AnimateImage();
        AnimateImage anim2 = new AnimateImage();

        // Put the components in a panel and then in a frame.
        JPanel contents = new JPanel();
        contents.add(anim1);
        contents.add(anim2);

        JFrame f = new JFrame("Animations");
        f.setContentPane(contents);
        f.pack();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);

        // Run each animation in its own thread.
        Thread t1 = new Thread(anim1);
        t1.start();
        Thread t2 = new Thread(anim2);
        t2.start();
    }
}
```

Parameterize for increased flexibility (image name, # of images, speed, direction through the array, length of time to run, etc.).
Name: Process All Elements

Context: You have a collection of values stored in an array and need to perform the same operation on all of them.

Solution: Use a `for` loop to process each element of the array, one element with each iteration of the loop.

```java
for(int «index» = 0; «index» < «arrayName».length; «index» += 1) {
    «elementType» «elementName» = «arrayName»[«index»];
    «statements to process the element»
}
```

The `foreach` loop is somewhat more concise when the index is not needed for processing:

```java
for(«elementType» «elementName» : «arrayName») {
    «statements to process the element»
}
```

Consequences: Each element in the array is processed. A variation of this pattern must be used for partially filled arrays.

Related Patterns: Many patterns are specializations of this pattern. For example, Process Matching Elements, Find and Extreme, etc.
Name: Linear Search (version 1)

Context: You want to find one element in an array, based on an identifying key.

Solution: Step through the elements of the array, breaking out of the loop when the element is found. This assumes a partially filled array.

```java
public «typeName» «methodName»(«type» «criteria»)
{ for (int i = 0; i < «auxVar»; i += 1)
    { «typeName» «elem» = «arrayName»[i];
        if («elem» satisfies «criteria»)
            { return «elem»;
            }
    }
    return «failureValue»;
}
```

Consequences: If the element is found, it is returned; otherwise a designated value such as `null` is returned.

Related Patterns: An alternate pattern avoids the early return out of the `for` loop. There are many variations for returning the value, a Boolean, the element index, etc.
**Name:** Linear Search (version 2)

**Context:** You want to find one element in an array, based on an identifying key.

**Solution:**

```java
public «typeOfElement» «methodName»(«type» «criteria»)
{
    int i = 0;
    while (i < «auxVar» && ! («arrayName»[i] satisfies «criteria»))
    {
        i += 1;
    }
    if (i == «auxVar»)  { return «failureValue»;  }
    else          {  return «arrayName»[i];   }
}
```

The **while** loop’s test depends on short-circuit evaluation.

**Consequences:** If the element is found, it is returned; otherwise a designated value such as **null** is returned.

**Related Patterns:** There are many variations for returning the value, a Boolean, the element index, etc.
Arrays hold multiple values and are referenced with indices. Each has the same meaningful sequence numbers. Arrays may have multiple dimensions and may have additional auxiliary variables that track filled elements with an auxiliary variable.

Arrays may be partially filled or may be reallocated to change size. Algorithms are processed with such as sort, find an extreme, insert, and delete. Such as insert and delete algorithms may have additional steps to process all elements. Arrays are prepared for use by first declaring variables, second allocating space, and third initializing elements of each type, which are sometimes meaningful, sometimes sequence numbers.

Algorithms such as sorting algorithms can be used to prepare arrays for use, possibly after partially filled arrays are reallocated. The process of sorting may involve additional steps such as finding an extreme value to insert into the array.
We have learned:

- that an array stores many elements accessed with an integer index.
- how to access individual values.
- how to process all the elements of the array, including variations to process only selected elements, find an extreme, and sort the array.
- how to declare, allocate, and initialize an array.
- that array references can be passed as arguments and returned from methods.
- how to store collections that grow and shrink using partially filled arrays, arrays that are reallocated, and a combination of the two methods.
- that arrays may store primitive types.
- that sometimes the indices have meaning within the problem.
- how to declare and use multi-dimensional arrays.
- how to animate an array of images.