Grouping objects

Introduction to collections – Part 1

5.0

Main concepts to be covered

- Collections (especially **`ArrayList`**)
- Builds on the *abstraction* theme from the last chapter.

The requirement to group objects

- Many applications involve collections of objects:
  - Personal organizers.
  - Library catalogs.
  - Student-record system.
- The number of items to be stored varies.
  - Items added.
  - Items deleted.

An organizer for music files

- Track files may be added.
- There is no pre-defined limit to the number of files.
- It will tell how many file names are stored in the collection.
- It will list individual file names.
- Explore the *music-organizer-v1* project.

Class libraries

- Collections of useful classes.
- We don't have to write everything from scratch.
- Java calls its libraries, *packages*.
- Grouping objects is a recurring requirement.
  - The **`java.util`** package contains classes for doing this.

```
import java.util.ArrayList;

/**
 * ...
 */
public class MusicOrganizer
{
    // Storage for an arbitrary number of file names.
    private ArrayList<String> files;

    /**
     * Perform any initialization required for the
     * organizer.
     */
    public MusicOrganizer()
    {
        files = new ArrayList<String>();
    }

    ...
}
```

Collections

- We specify:
  - the type of collection: **`ArrayList`**
  - the type of objects it will contain: **`<String>`**
  - **`private ArrayList<String> files;`**
- We say, “ArrayList of String”.

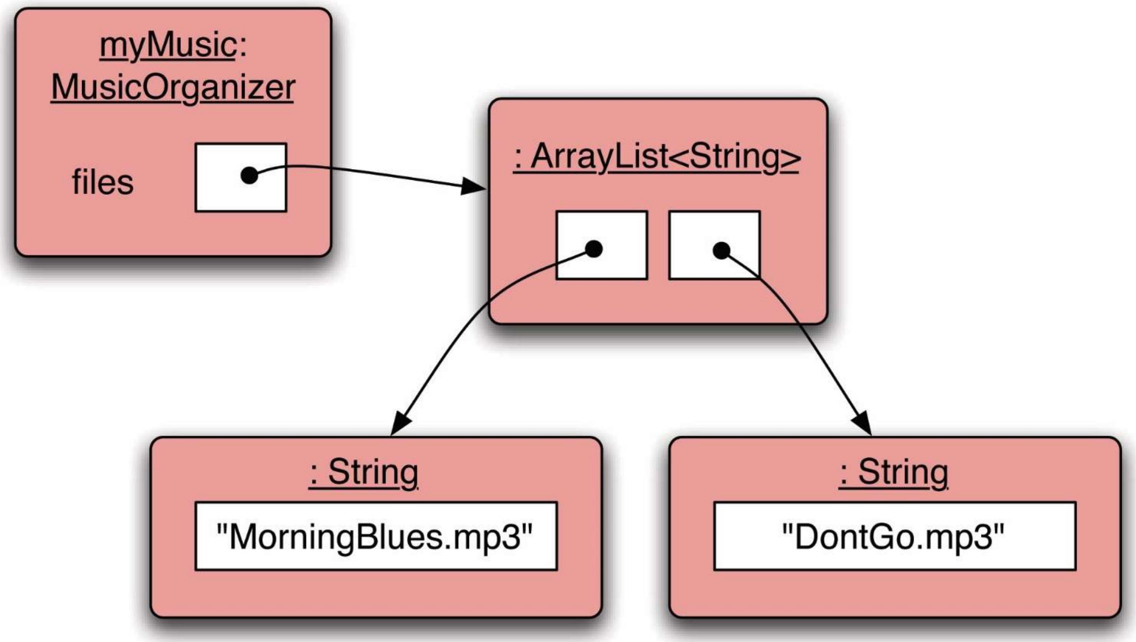
Generic classes

- Collections are known as *parameterized* or *generic* types.
- **`ArrayList`** implements list functionality:
  - **`add`**, **`get`**, **`size`**, etc.
- The type parameter says what we want a list of:
  - **`ArrayList<Person>`**
  - **`ArrayList<TicketMachine>`**
  - etc.

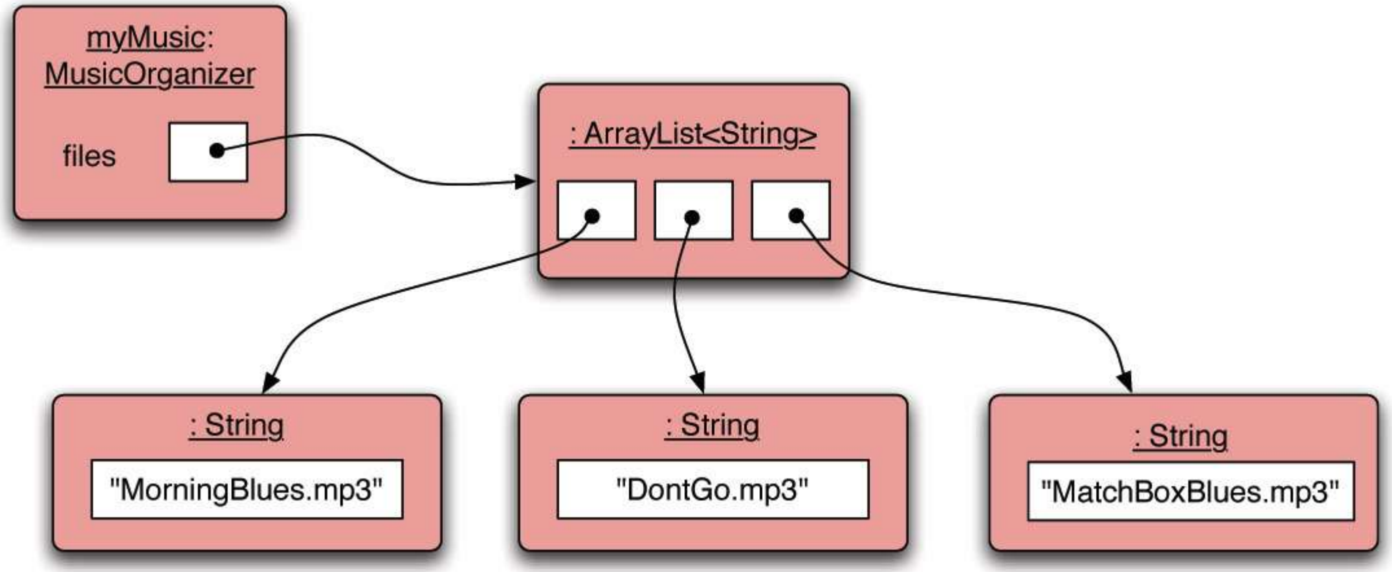
Creating an ArrayList object

- In versions of Java prior to version 7:
  - `files = new ArrayList<String>();`
- Java 7 introduced 'diamond notation'
  - `files = new ArrayList<>();`
- The type parameter can be inferred from the variable being assigned to.
  - A convenience.

Object structures with collections



Adding a third file



Features of the collection

- It increases its capacity as necessary.
- It keeps a private count:
  - `size()` accessor.
- It keeps the objects in order.
- Details of how all this is done are hidden.
  - Does that matter? Does not knowing how prevent us from using it?

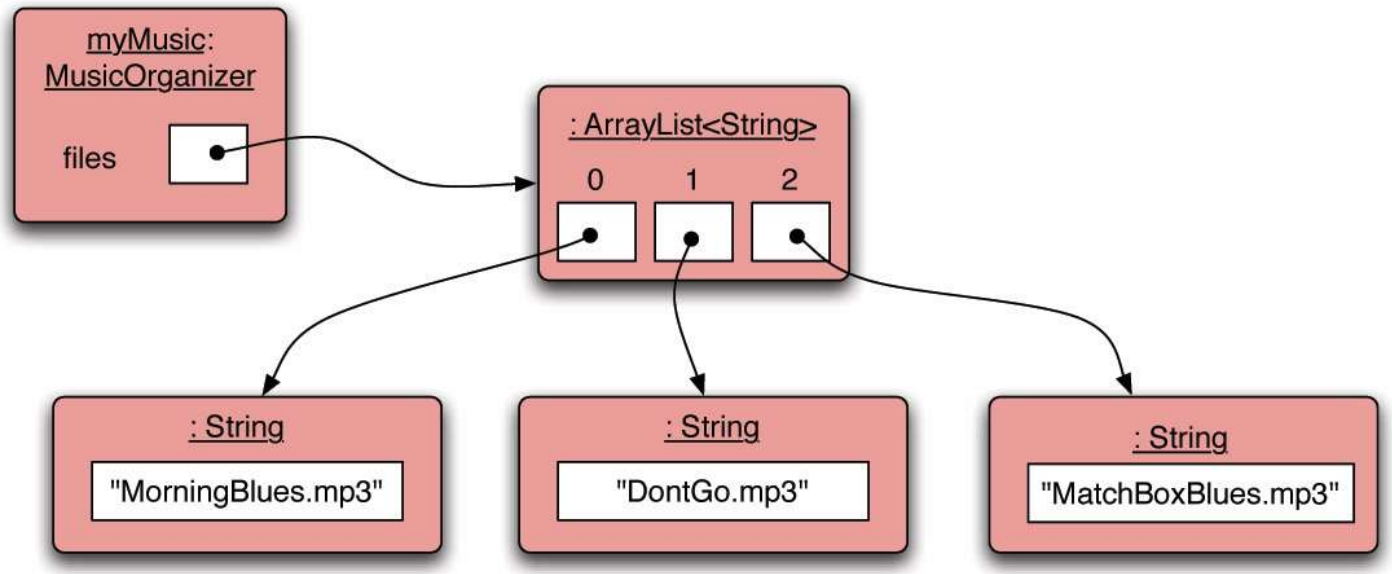
Using the collection

```
public class MusicOrganizer
{
    private ArrayList<String> files;

    ...

    public void addFile(String filename)
    {
        files.add(filename);
    }

    public int getNumberOfFiles()
    {
        return files.size();
    }

    ...
}
```

Adding a new file

Returning the number of files *(delegation)*

Index numbering



Retrieving an object

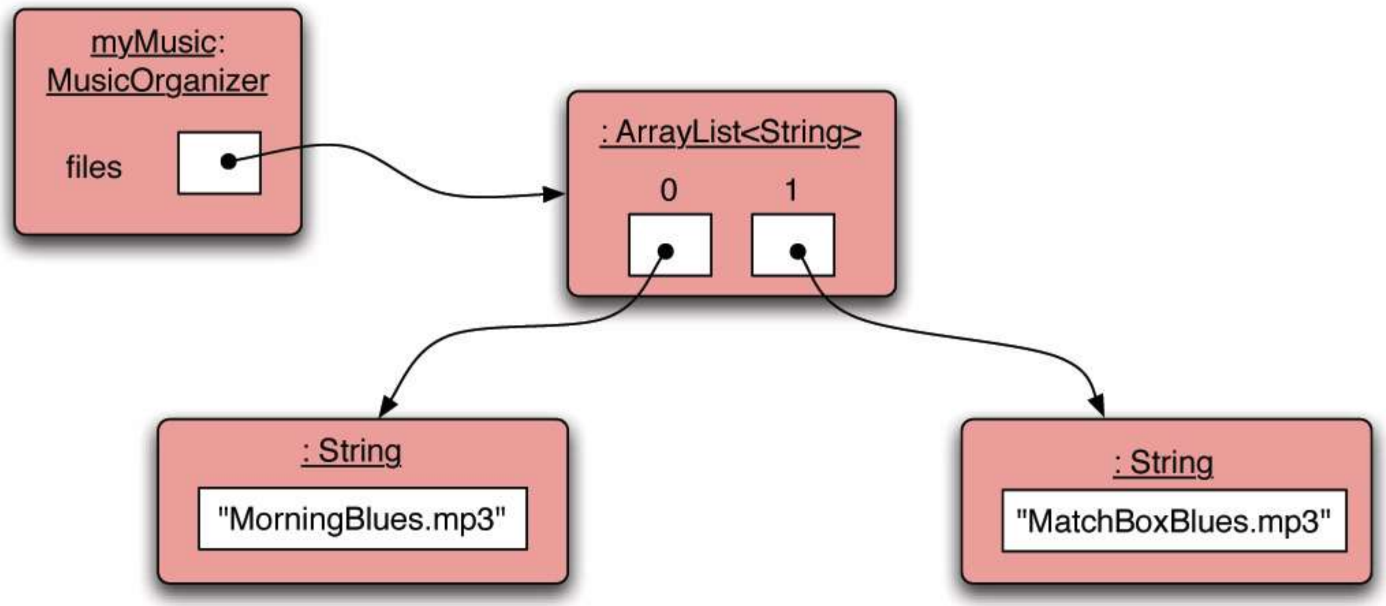
```
public void listFile(int index)
{
    if(index >= 0 &&
            index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
    }
    else {
        // This is not a valid index.
    }
}
```

Index validity checks

Retrieve and print the file name

Needed? (Error message?)

Removal may affect numbering



The general utility of indices

- Using integers to index collections has a general utility:
  - ‘next’ is: `index + 1`
  - ‘previous’ is: `index – 1`
  - ‘last’ is: `list.size() – 1`
  - ‘the first three’ is: the items at indices `0, 1, 2`
- We could also think about accessing items in sequence: `0, 1, 2, …`

Review

- Collections allow an arbitrary number of objects to be stored.
- Class libraries usually contain tried-and-tested collection classes.
- Java's class libraries are called *packages*.
- We have used the **`ArrayList`** class from the **`java.util`** package.

Review

- Items may be added and removed.
- Each item has an index.
- Index values may change if items are removed (or further items added).
- The main **`ArrayList`** methods are **`add`**, **`get`**, **`remove`** and **`size`**.
- **`ArrayList`** is a parameterized or generic type.

Interlude:
Some popular errors...

What's wrong here?

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
    if(files.size() == 0); {
        System.out.println("Organizer is empty");
    }
    else {
        System.out.print("Organizer holds ");
        System.out.println(files.size() + " files");
    }
}
```

This is the same as before!

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(files.size() == 0);

   {
      System.out.println("Organizer is empty");
   }
   else {
      System.out.print("Organizer holds ");
      System.out.println(files.size() + "files");
   }
}
```

This is the same again

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(files.size() == 0)
      ;

   {
      System.out.println("Organizer is empty");
   }
   else {
      System.out.print("Organizer holds ");
      System.out.println(files.size() + "files");
   }
}
```

and the same again...

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(files.size() == 0) {
       ;
   }

   {
       System.out.println("Organizer is empty");
   }
   else {
       System.out.print("Organizer holds ");
       System.out.println(files.size() + "files");
   }
}
```

This time I have a boolean field called 'isEmpty' ...

What's wrong here?

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
    if(isEmpty = true) {
        System.out.println("Organizer is empty");
    }
    else {
        System.out.print("Organizer holds ");
        System.out.println(files.size() + "files");
    }
}
```

This time I have a boolean field called ʻisEmptyʼ ...

The correct version

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(isEmpty == true) {
      System.out.println("Organizer is empty");
   }
   else {
      System.out.print("Organizer holds ");
      System.out.println(files.size() + "files");
   }
}
```

What's wrong here?

```
/**
 * Store a new file in the organizer. If the
 * organizer is full, save it and start a new one.
 */
public void addFile(String filename)
{
   if(files.size() == 100)
       files.save();
       // starting new list
       files = new ArrayList<String>();

   files.add(filename);
}
```

This is the same.

```
/**
 * Store a new file in the organizer. If the
 * organizer is full, save it and start a new one.
 */
public void addFile(String filename)
{
   if(files.size == 100)
       files.save();

   // starting new list
   files = new ArrayList<String>();

   files.add(filename);
}
```

The correct version

```
/**
 * Store a new file in the organizer. If the
 * organizer is full, save it and start a new one.
 */
public void addFile(String filename)
{
    if(files.size == 100) {
        files.save();
        // starting new list
        files = new ArrayList<String>();
    }

    files.add(filename);
}
```