Grouping objects
Lecture 7

Waterford Institute of Technology

April 24, 2014

John Fitzgerald

Abstraction & object interaction

Abstraction

- Simplify problem by identifying discrete components
- Details hidden behind public interface

Object interaction

- Assemble component set to act as unit
- Use component public interface

```
public class BIABank
{
    private Person manager;
    private Person customer;
    private Account account;
    public BIABank(int accountNmr) {
        account.set(accountNmr);
    }
}
```

Using library class

Class libraries major aid to abstraction & modularization

- Java libraries called packages
- Recall: package grouping related types
- Example *java.util* package
- Includes *ArrayList* class
- Known as *collection* class
- Import statement grants access to class

```
import java.util.ArrayList;
public class Notebook
{
    private ArrayList<String> notes;
}
```

ArrayList

ArrayList example *flexible* collection class

- Can store arbitrary number elements
- Stored object type determined at instantiation
- Cannot directly store primitive types
- *Diamond* notation : < >
- new ArrayList<String>()
- Each element of *notes* is String object

```
import java.util.ArrayList;
public class Notebook
{
    private ArrayList<String> notes;
    public Notebook() {
        notes = new ArrayList<String>();
    }
}
```

ArrayList methods

Storing primitives

Wrap primitive in Number class.
Example

- Integer
- Double

```
import java.util.ArrayList;

ArrayList<Integer> list = new ArrayList<>();
list.add(100);
System.out.println(list.get(0);
//Output is 100
```

ArrayList methods

ArrayList method

- `size` : returns number list elements

```
import java.util.ArrayList;
public class Notebook
{
    private ArrayList<String> notes;
    ...
    public int numberOfNotes() {
        return notes.size();
    }
}
```

ArrayList methods

ArrayList method

- `get` : returns element at specified index position

```
import java.util.ArrayList;
public class Notebook
{
    private ArrayList<String> notes;
    ...
    public String showNote(int noteNumber) {
        return notes.get(noteNumber);
    }
}
```

ArrayList methods

ArrayList method

- `remove` : removes element at specified position in list

```
import java.util.ArrayList;
public class Notebook
{
    private ArrayList<String> notes;
    ...
    public void removeNote(int noteNumber)
        notes.remove(noteNumber);
    }
}
```

Generic classes

Generic classes potentially define many types

`ArrayList<String> notes;`

- Specifies an ArrayList of String types

String Java class but could equally define user-defined types

`ArrayList<House> houses;`

- Specifies an ArrayList of House types

Generic classes

Numbering within collections

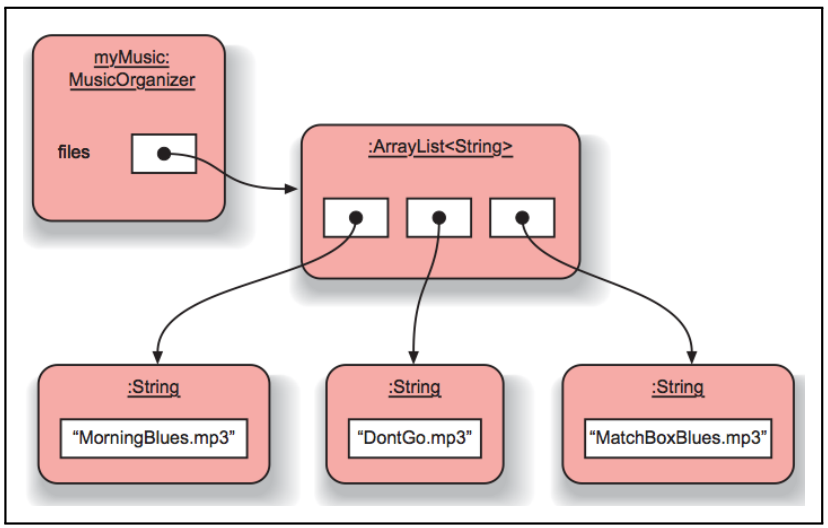
- Zero index based
- Index of next added element is *size*

Example

- Assume *String notes* has 3 elements, i.e. size is 3
- Its indices: $0, 1, 2$
- Add new element: its index 3 i.e. former size

Generic classes

```
public class MusicOrganizer
{
    ArrayList<String> files;
    ...
}
...
MusicOrganizer organizer = new MusicOrganizer();
organizer.add("MorningBlues.mp3");
organizer.add("DontGo.mp3");
organizer.add("MatchBoxBlues.mp3");
```



Collection traversal

Processing a collection

Three techniques to traverse a collection

- *for-each* loop
  - Standard technique to process all elements
- *while* loop
  - Use when unsure at outset how many elements for processing
- *iterate* over collection
  - A more general approach than *for-each* or *while*

for-each

Using *for-each* to process collection

```
for(Object o : collection)
{
    statement(s)
}
```

```
//Print all notes in list
public void listNotes()
{
    for(String note: notes)
    {
        System.out.println(note);
    }
}
```

while

Using *while* to process all or part collection

```
while (expresion)
{
    statement(s)
}
```

```
//Print all notes in list
public void listNotes()
{
    int index = 0;
    while(index < notes.size())
    {
        System.out.println(notes.get(index));
        index = index + 1;
    }
}
```

do while

Used where a loop will be traversed at least once

Differs from *while*

- expression evaluated at bottom of loop

```
do
{
    statement(s)
} while (expression)
```

```
//Print numbers in range [0, n] using do-while
public void printNumbers(int n)
{
    int count = 0;
    do
    {
        System.out.println(count);
        count += 1;
    } while (count <= n);
}
```

iterator

Using *iterator* to process collection

Iterator a Java class defined in *java.util* package

```
ArrayList<Object> collection;
Iterator<Object> it = collection.iterator();
while(it.hasNext())
{
    Object o = it.next();
}
```

```
//Print all notes in list
public void listNotes()
{
    Iterator<String> it = notes.iterator();
    while(it.hasNext())
    {
        System.out.println(it.next());
    }
}
```

for

for statement precedes *for-each*

Could be used to traverse a collection

However, *for-each* preferrable

- Unless individual element access required

```
for (initialization; termination; increment) {
        statement(s)
}
```

```
//Print all notes in list
public void listNotes()
{
    for (int i=0; i < notes.size(); i = i + 1)
    {
        System.out.println(notes.get(i));
    }
}
```

Arrays

Arrays are fixed-size collections

Have advantages over flexible collections

- Java's oldest collection structure
- Access to elements often more efficient
- Can store objects of primitive types
- Flexible types can store objects only

```
public LogAnalyser
{
    private int[ ] hours;
    public Analyzer() {
        hours = new int[24];
    }
}
```

Declaring arrays

Declaration array variable

Example: int[ ] hours

Two components

- Type: int[ ]
- Name: hours

```
String[ ] name;
float[ ] cost;
double [ ] amount;
boolean[ ] results;
```

Create, initialize, access arrays

Create *int* array

- int[ ] `hours` = new int[2]

Initialize array

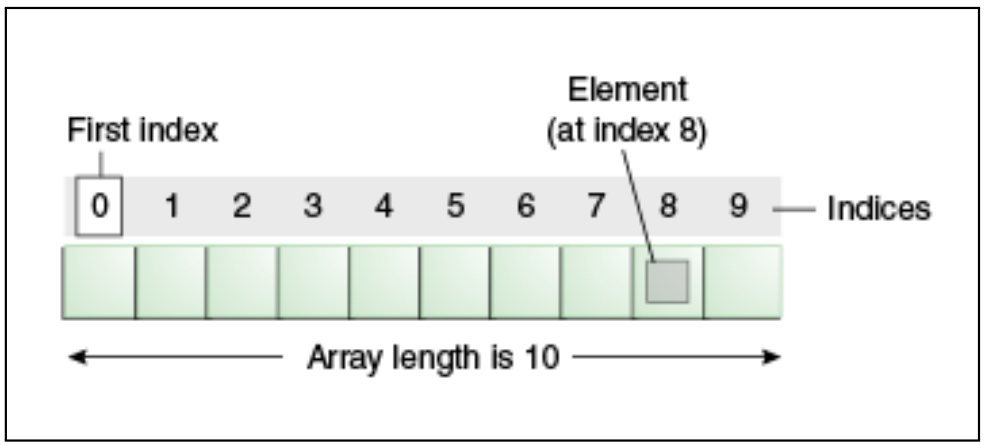
- `hours`[0] = 2;
- `hours`[1] = 6;

Declare, create & initialize

int[ ] `hours` = {2, 6}

Access 2nd element

- int `timeNow` = `hours`[1];



Copying arrays

Java *System* class has *arraycopy()* method

- Efficiently copies one array to another

```
/**
 * copies src array to dest array
 * begins copy from at srcPos
 * begins paste at destPos
 */
public void copyArray()
{
    int[ ] src = {1,2,4,6,8};
    int length = 5;
    int[] dest = new int[length];
    int srcPos = 0;
    int destPos = 0;
    System.arraycopy(src, srcPos, dest, destPos, length);
}
```

Using arrays

Individual array elements accessible by

- Using *for* to *index* array
  - `int time = hours[4];`
- Using *for-each*

```
/**
 * use for
 */
public void print(int[] ar) {
    for(int i=0; i<ar.size; i++) {
        System.out.println(ar[i]);
    }
}
```

```
/**
 * use for-each
 */
public void print(int[] ar) {
    for(int val : ar) {
        System.out.println(val);
    }
}
```

Array indices

Array indices

- Begin at 0
- End at one less than array size

Common mistakes

- Begin at 1
- End at size array

```
for(int i = 1; i <= intArray.size; i = i+1)
{
    ...
}
```

Incrementing and decrementing

Mostly to date we have used naive method to increment

- `val = val + 1;`

Commonly practiced methods increment & decrement

- `val++` increments val by 1
- `val += x` increments val by x
- `val--` decrements val by 1
- `val -= x` decrements val by x

```
for(int i = 1; i <= intArray.size; i++)
{
    ...
}
```

Selecting loop method

How to choose between *for-each*, *for*, *while* and *iterator*?

for-each

- Concisely traverse collection
- Can be used on arrays
- Cannot be used to remove an element

for

- Good if number iterations known at outset

while

- Interchangeable with *for*

iterator

- Can traverse entire collection
- Can remove elements

Java String class

Has several constructors and methods

- `compareTo(String anotherString)`
  - Compares 2 strings character-wise
- `toUpperCase()`
  - Changes invoking string to upper case
- `equalsIgnoreCase(String anotherString`
  - Compares 2 strings for equality ignoring case
- `length()`
  - Calculate and returns length of string
- `valueOf(int i)`
  - Returns string representation of integer argument

```
String str = new String("textwidth");
//creates new string
int length = str.length();
//obtains length of string (9)
str.toUpperCase();
//converts textwidth to TEXTWIDTH
```

```
String str = String.valueOf(100);
//creates string "100" referenced by str
String str2 = "online";
//creates string "online"
str.equalsIgnoreCase(str2);
//compares str and str2, returning false
      unless both strings equivalent
```

Java String class

Manipulate characters in String object

- String an ordered collection characters
- Zero-indexed array
- `charAt(int pos)` accesses character at position pos

Concatenate string objects

- `concat(String str)` concatenates str to existing string
- Overloaded + operator achieves same effect

```
String str = "TEXT"
str.charAt(0) is "T"
str.charAt(1) is "E"
str.charAt(2) is "X"
str.charAt(3) is "T"
```

```
String s3 = "why ";
String s4 = "not?";
String s5 = s3 + s4;
System.out.println("s3 + s4 "+s5);
System.out.println("s3 + s4 "+s3.concat(s4));
//both statements output: s3 + s4 why not?
```

String objects immutable

Health warning: String objects once created cannot be modified

```
String str = new String("java");
//str is reference pointing to string object "java"
str.toUpperCase();//has no effect. Produces no warning.
System.out.println(str);//outputs java, not JAVA
str = "JAVA";//reference str points to different object
System.out.println(str);//outputs JAVA
```