Grouping objects
Lecture 7

Waterford Institute of Technology

February 2, 2016

John Fitzgerald

Presentation outline

Estimated duration presentation

Questions at end presentation

Topics discussed:

- Abstraction & object interaction
- Using library classes
- Packaging
- Generics such as ArrayList
- Traversal - looping techniques
- Native arrays

Abstraction & object interaction

Abstraction

- 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

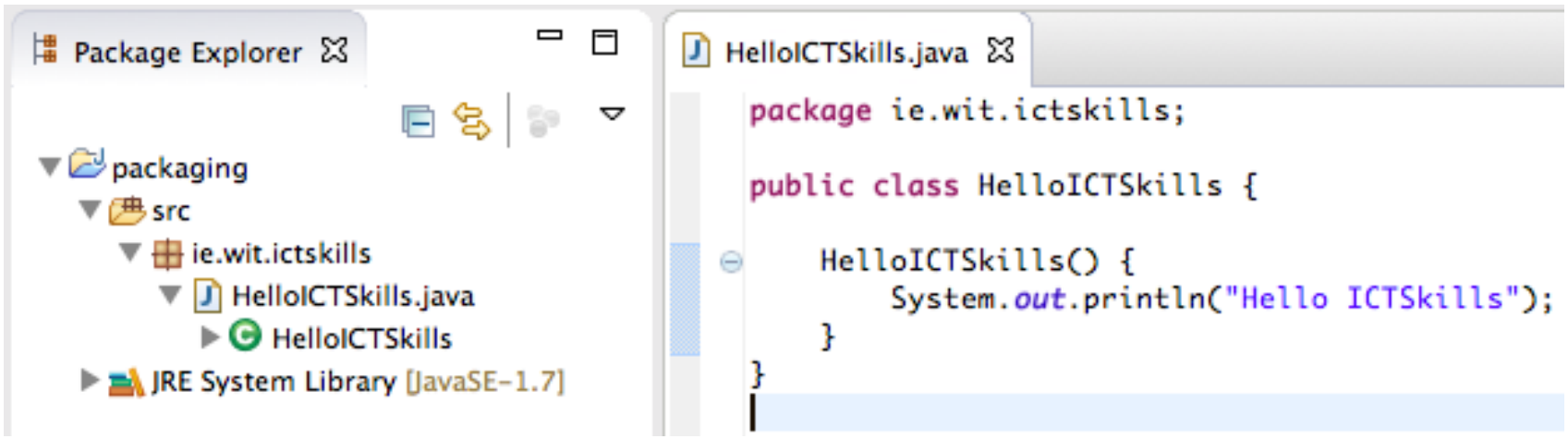
major aid to abstraction & modularization

- 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;
}
```

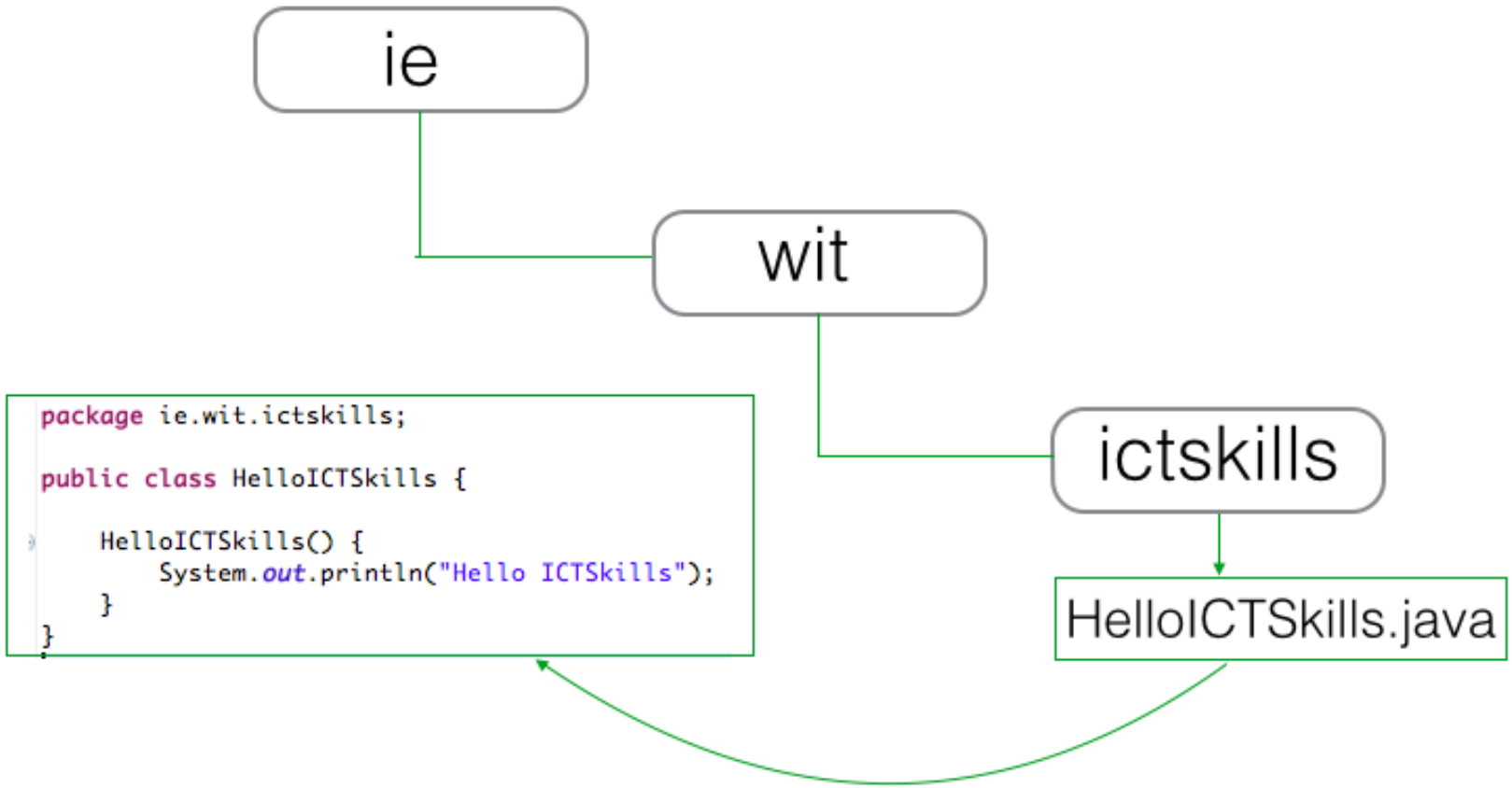
Packaging

Facilitates unique naming



Packaging

Facilitates unique naming



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

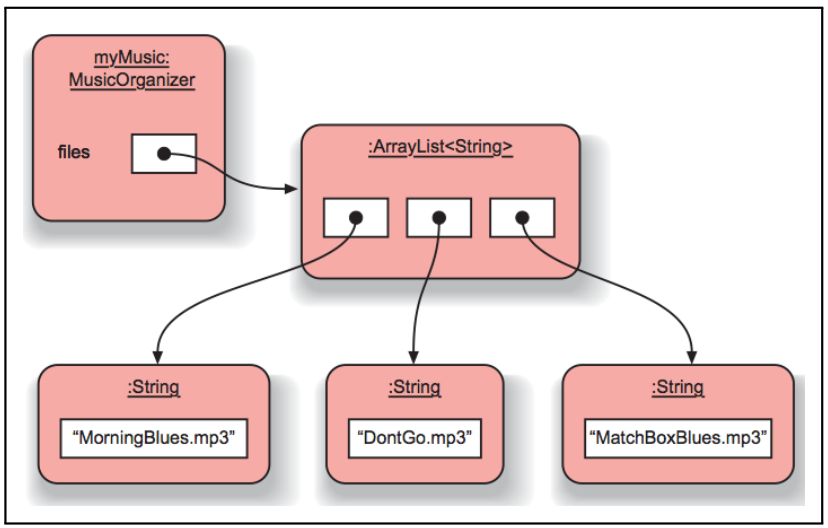
- Zero index based
- Index of next added element is *size*

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> organizer;
    ...
}
...
MusicOrganizer organizer = new MusicOrganizer();
organizer.add("MorningBlues.mp3");
organizer.add("DontGo.mp3");
organizer.add("MatchBoxBlues.mp3");
```



Generic and non-generic classes

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Collection traversal

Processing a collection

Three techniques to traverse a collection

- *for-each* or *for* 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

Could be used to traverse a collection

However, *for-each* preferrable

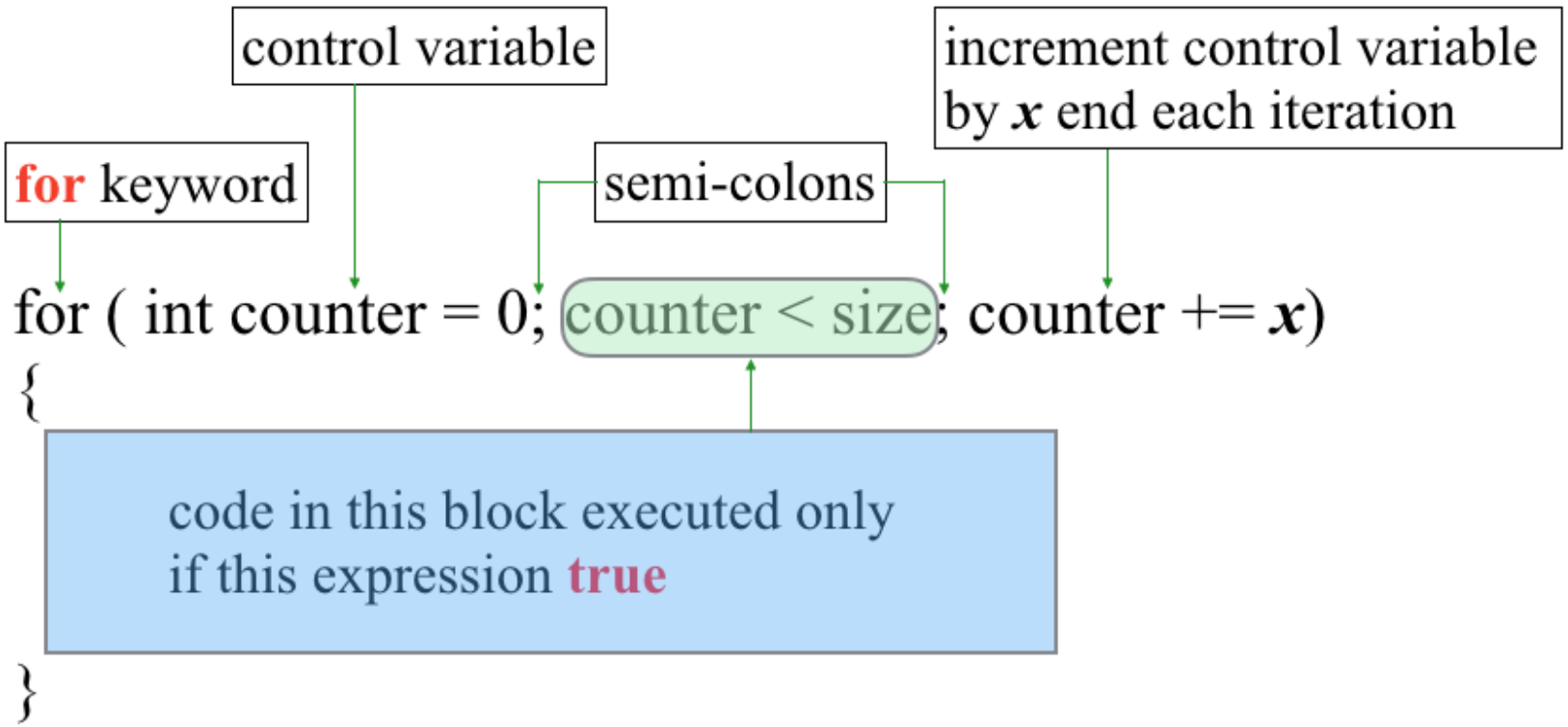
- Unless individual element access required

```
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));
    }
}
```

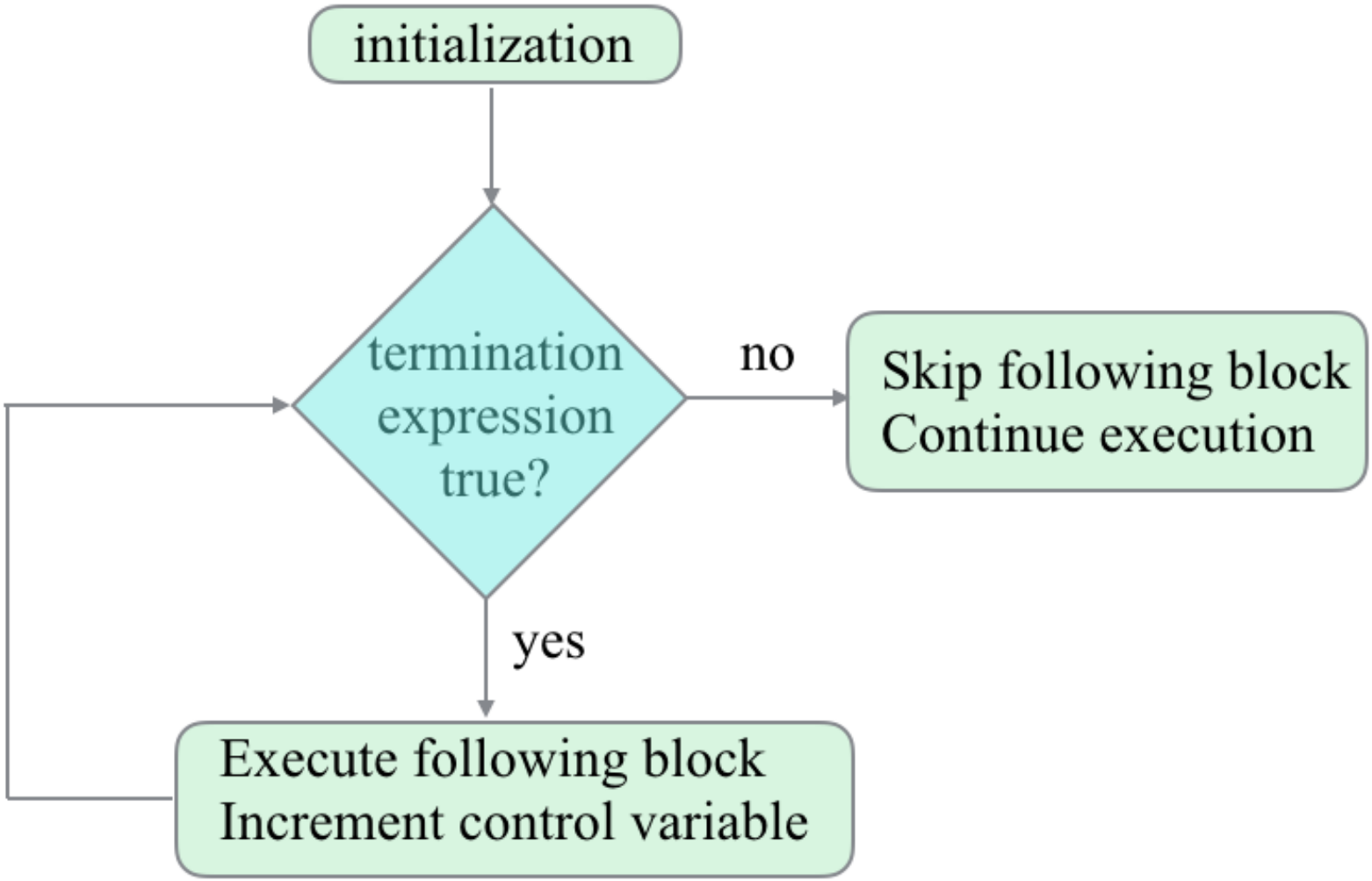
for

Syntax


control variable
increment control variable by x end each iteration
for keyword
semi-colons
for (int counter = 0; counter < size; counter += x)
{
code in this block executed only if this expression true
}

for

Logical flow



for-each

for-each

introduced in Java 5

Using *for-each* to process collection

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

```
//Print all notes in list
ArrayList<String> notes = ...;
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 type 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());
    }
}
```

iterator

Comparison with classical for loop

Using *iterator* to process ArrayList<Integer>

```
ArrayList<Integer> list;
Iterator<Integer> iter = list.iterator(); // iterator() ArrayList method
while(iter.hasNext()) {
    Integer val = it.next();
    if ((val % 2) ==0) {// val is even number
     iter.remove(); // removes all even numbers
    }
}
```

Using classical *for* to process ArrayList<Integer>

```
for (int i = 0; i < list.size(); i++) {
    Integer val = list.get(i);
    if ((val % 2) ==0) // val is even number
     list.remove(val); // removes all even numbers
}
```

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

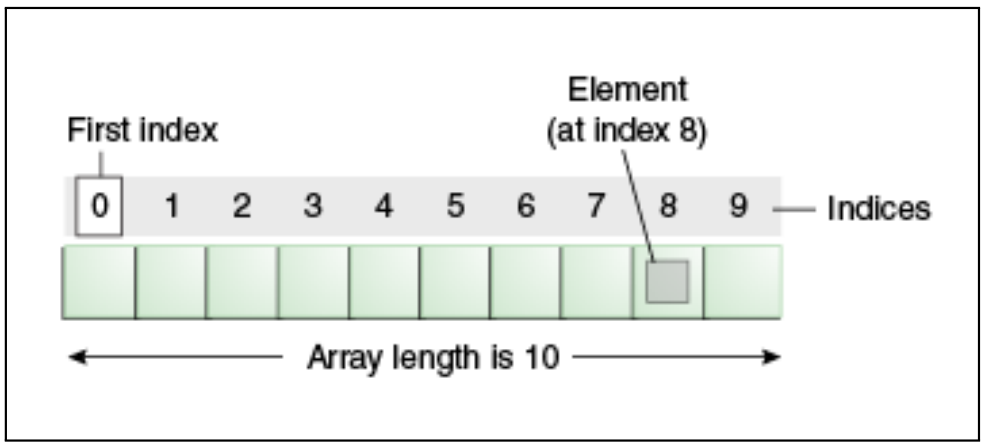
- `hours`[0] = 2;
- `hours`[1] = 6;

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* loop and array *index*
  - `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

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

Incrementing and decrementing

Example naive method to increment:

- `val = val + 1;`

Commonly practiced methods increment & decrement

- `val++` increments val by 1
- `++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

Summary

- Abstraction
  - hide the details
- Object interaction
  - how objects create other objects
  - objects call or invoke each other's methods

  Class libraries
  - such as java.util package
  - includes collection classes, date, time, random number generator
- ArrayList class
  - arbitrary number elements
  - can store different element types

Summary

- Generic classes
  - potentially define many types
  - an example is ArrayList class
- packaging
  - avoids name clash
  - unique identification class names
- Zero-based indexing
  - the norm in Java and many computer languages
  - some languages such as Fortran use base one

Summary

- Collection traversal
  - for, for-each, while, do-while, iterator.
  - while: number cycles determined at runtime.
  - do-while: at least one cycle.
  - for-each (enhanced for): use when possible.
  - iterator: use has important benefits.

Referenced Material

Summary of Java operators

`http://docs.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html`

[Accessed 2015-02-18]

Benefits of using iterator

`http://stackoverflow.com/questions/3595772/what-are-the-benefits-of-using-an-iterator-in-java`

[Accessed 2016-01-28]