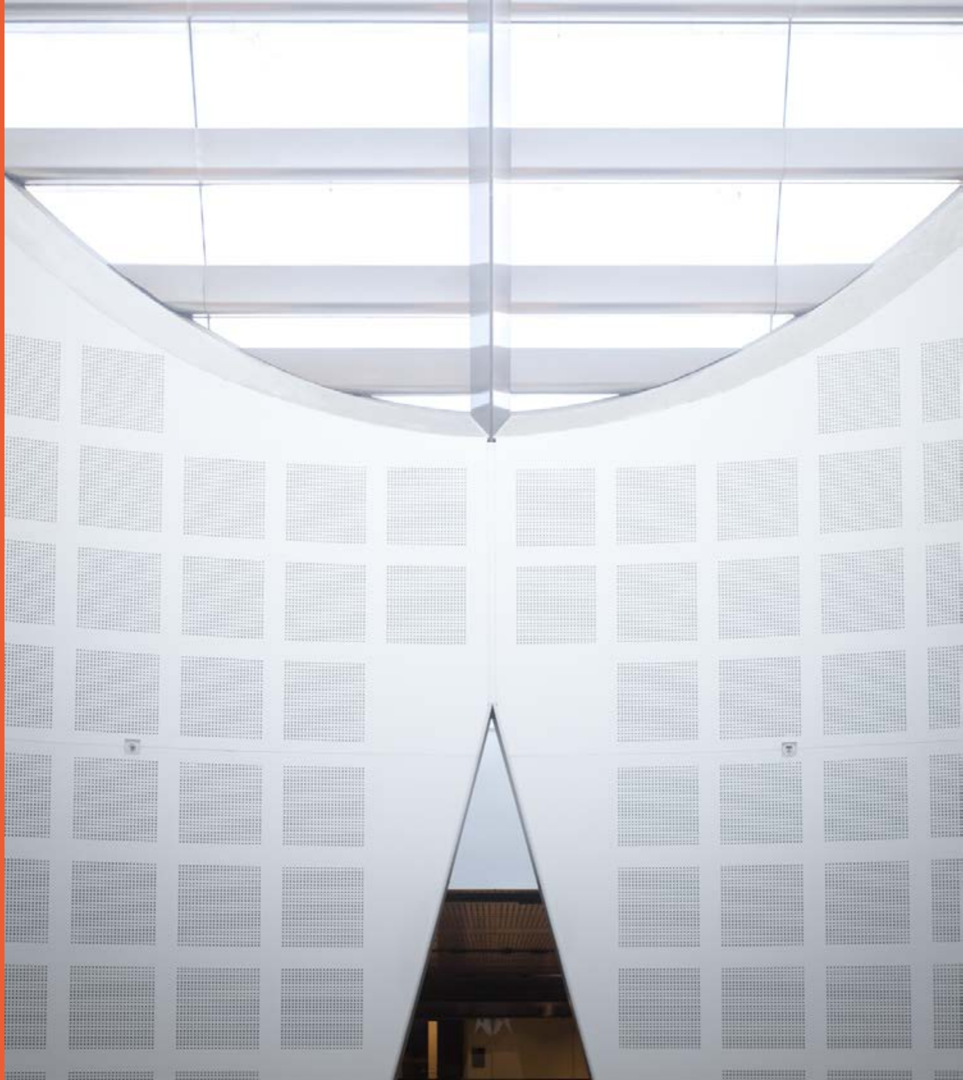
COMP9103: Software Development in Java

W5: Class Members & ArrayList

Presented by
Dr Ali Anaissi
School of Computer Science

THE UNIVERSITY OF SYDNEY



Review of Classes & Objects

Class Definition

- Class name should be noun
- Each word starts with capital letter

accessSpecifier **class** *ClassName*

{

fields/variables

constructors

methods

}

accessSpecifier fieldType fieldName;

Normally, **private**

When an instance variable is declared **private**, it is accessible only by methods/constructors of the class in which it is defined.

Each object of a class has its own set of instance fields

Class Definition

accessSpecifier **class** *ClassName*
{
fields/instance variables

constructors

methods
}

Constructors have **no return type and value**

Constructor name = class name

accessSpecifier ClassName(parameterType parameterName, …)
{
constructor body
}

Constructors contain instructions **to initialize the instance fields of an object**

Class Definition

accessSpecifier **class** *ClassName*
{
fields/instance variables

constructors

methods
}

specifiers are words to set characteristics of the method. E.g.,"**public**", "**static**"

Specify the type of value returned from the method. "**void**" indicates that the no value be returned from the method.

Represent information passed to the method from method invoker/user. Parameter-list can be empty, or consist of one or more parameter declarations in the format of **type parameter-name** for each parameter, separated by **commas**.

specifiers return-type methodName (parameter-list)

{

statements;

}

Method body (black box)

interface

Class Members

Look inside a class definition

```
public class ClassName {
  /*  static members: fields and methods specified with
   *  static modifier, including static fields and static
   *  methods
   */

/* instance members: fields and methods without static
   * modifier, including instance fields and instance
   * methods
   */


   //constructors
}
```

Instance Fields

- An object uses instance fields to store & specify its state
  - An **instance field** is a storage location that is present in **each object** of the class.
- The class declaration specifies the instance fields:

```
public class Customer {
  private double creditCardBalance;
  private double chequeAccountBalance;
  private String name;
…  …  …
```

- A class declares the type of an instance field, but does NOT reserve memory space for any instance fields

Instance Methods

- Every method must be in a class
- Instance methods are invoked via an object/instance

```
public class Customer {
        private double creditCardBalance;
        private double chequeAccountBalance;
        private String name;

        public double wealth() {
                return creditCardBalance + chequeAccountBalance;
        }

        public boolean inDebt() {
                return (wealth() < 0);
```

Instance methods

The object on which a method is invoked is called the implicit parameter

a

Customer	
creditCardbalance	-127
chequeAccountBalance	4351
name	Paul

Call the methods from the object
Eg., a.wealth()

The keyword *this*

- The keyword **this**, when used inside a method, refers to the receiver object.

- It has two main uses:
  - to return a reference to the receiver object from a method
  - to call constructors from other constructors.

The keyword *this*

– For example, we may add a method setName(String name) in the previous Customer class, it can be defined as follows:

```
public void setName(String name) {
  this.name = name;
}
```

this represents the implicit parameter, eg **this**.name to indicate name is an instance field

The keyword *this*

- The class Customer has two constructors as follows:

```
public Customer(String name, double ccb) {
        this.name = new String(name);
        this.creditCardBalance = ccb;
        this.chequeAccountBalance = 0;
}
```

```
public Customer(String name, double ccb, double cab) {
        this.name = new String(name);
        this.creditCardBalance = ccb;
        this.chequeAccountBalance = cab;
}
```

- The second can be defined in terms of the first one:

```
public Customer(String name, double ccb, double cab) {
        this(name, ccb);
        this.chequeAccountBalance = cab;
}
```

Static Fields

- static fields (with static as the specifier)

```
private static int customerNumber = 1000;
```

- A static field (also called class field) belongs to the class.
  - static fields can be used even when no object created
  - Only one copy and No duplication
  - Values in static fields are shared among all objects created from this class
  - Think of these as some kind of "global variable", where changes are visible to all instances

Static Fields

- Suppose that we need a customer number in the class Customer, so that the 1st customer created should be 1001, and then the 2nd should be 1002, the 3rd should be1003, … …
- We need to keep track of the last assigned number and increment it each time we create a new customer.

```
public class Customer {
        private double creditCardBalance;
        private double chequeAccountBalance;
        private String name;
        private int customerNumber;
        private static int lastCustomerNumber = 1000;
```

- If ***lastCustomerNumber*** was not *static*, each instance of ***Customer*** would have its own value of ***lastCustomerNumber***

Constructors & Static Fields

– During construction, each constructor in the Customer class needs to increment the lastCustomerNumber and assign the value of lastCustomerNumber to customerNumber in the current object.

```
public Customer(String name){
        this.name = name;
        creditCardBalance = 0;
        chequeAccountBalance = 0;
        lastCustomerNumber++;          //Updates the static field
        customerNumber = lastCustomerNumber;
        // Assigns field to account number of this new customer
}

public Customer(String name, int ccb, int cab){
        this.name = name;
        creditCardBalance = ccb;
        chequeAccountBalance = cab;
        lastCustomerNumber++;
        customerNumber = lastCustomerNumber;
}
```

Static methods

- The main() method we have defined follows the method definition syntax:

  public static void main (String [ ] args) {……}

- The method max() from Math class also follows the method definition

  public static int max (int a, int b) { ……}

- *static* methods belong to a class (NOT to a specific object) and are invoked via the class name

  E.g.: Math.max(3, 5);

Static methods

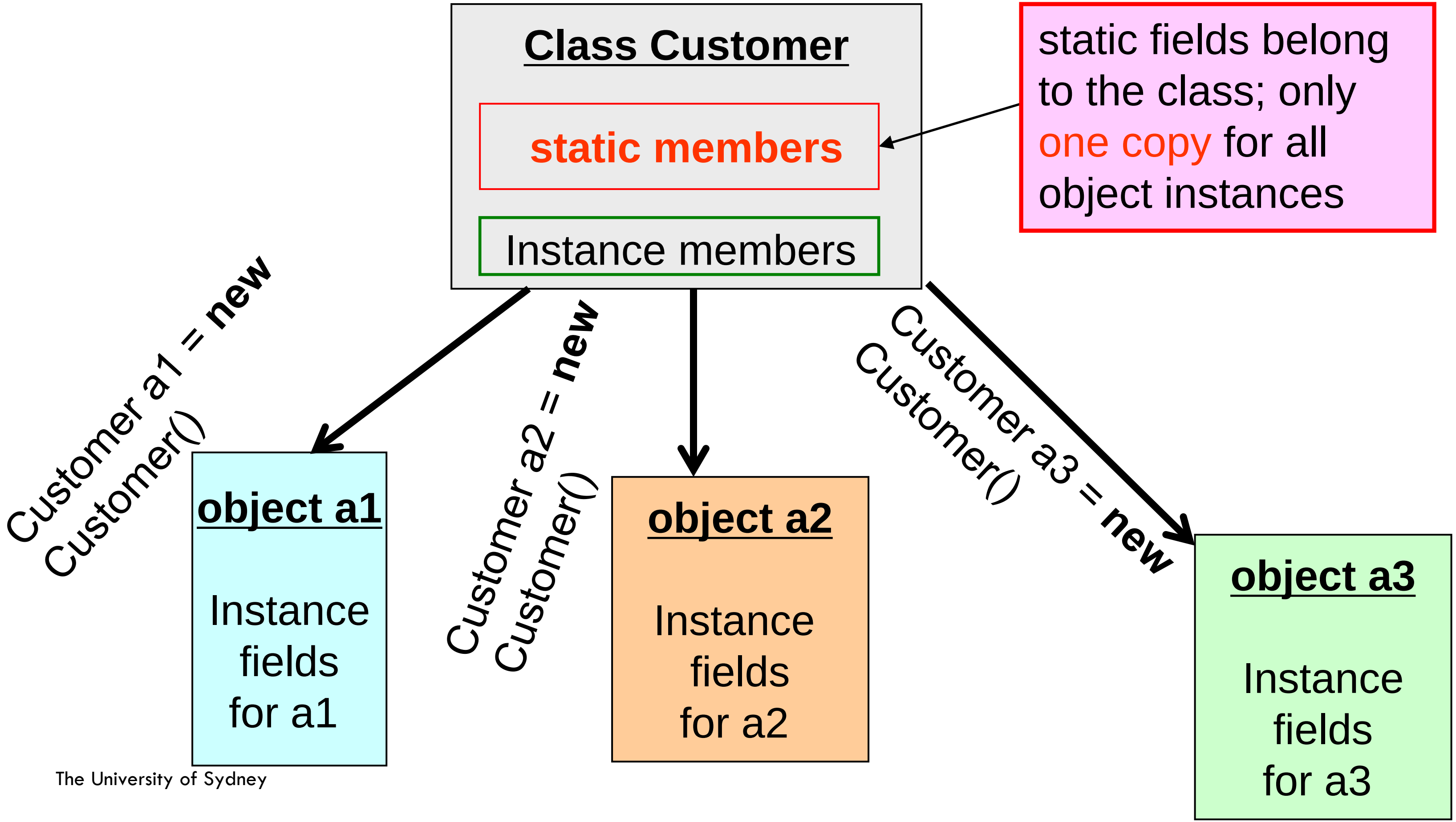
```java
public class Customer {
....
private double creditCardBalance;
private double chequeAccountBalance;
private String name;
private int customerNumber;
private static int lastCustomerNumber = 1000;
private static int transactionFee = 1;
....
public static void incrementTransactionFee() {
        transactionFee += 2;
}
public static int getTransactionFee() {
        return transactionFee;
}
```

```java
public class CustomerTester {
...
    System.out.println(Customer.getTransactionFee());
    Customer.incrementTransactionFee();
    System.out.println(Customer.getTransactionFee());
…
}
```

Class name

Prints
1
3

Schematic View of Static vs Instance Members

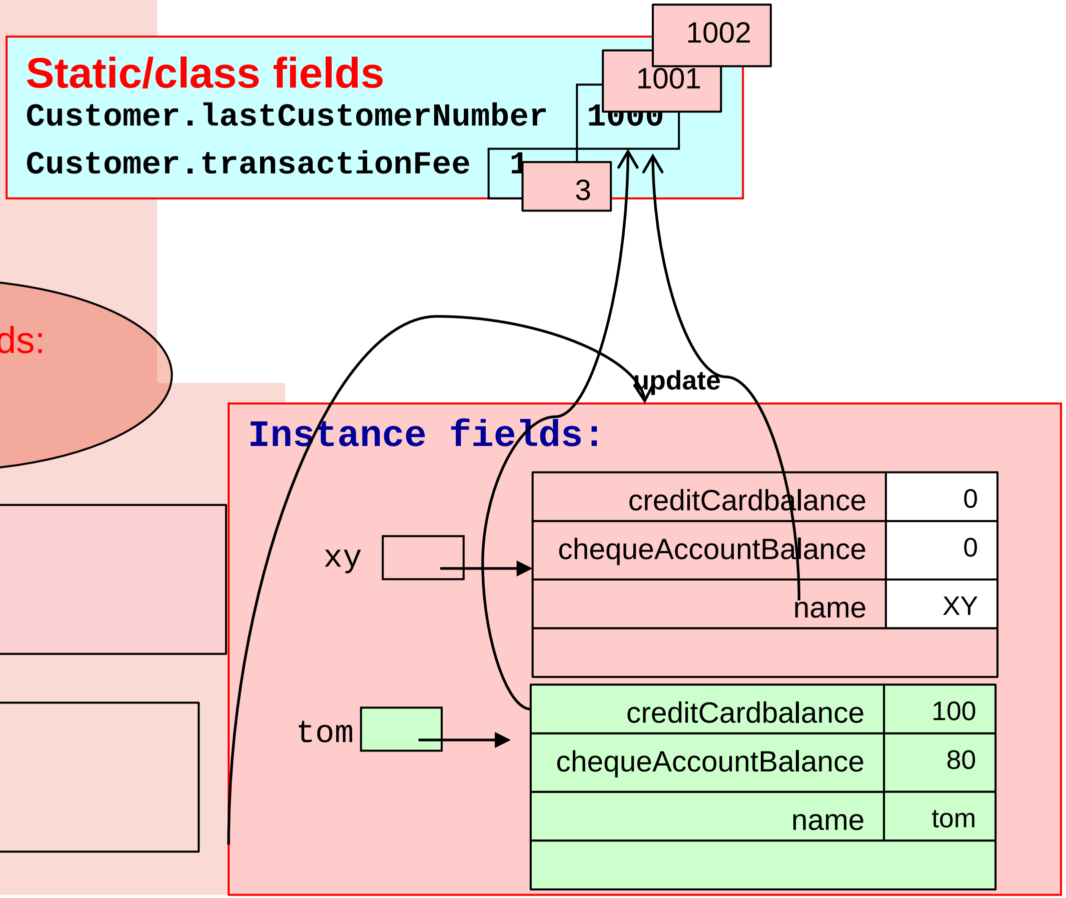


```
public class Customer {
//....
        private double creditCardBalance;
        private double chequeAccountBalance;
        private String name;
        private int customerNumber;
        private static int lastCustomerNumber = 1000;
        private static int transactionFee = 1;
        //....
        public static void incrementTransactionFee() {
            transactionFee += 2;
        }
        public static int getTransactionFee() {
            return transactionFee;
        }
//
```

Class methods: invoked from class name

```
public class CustomerClient {
  public static void main(String[] args) {
    System.out.println(Customer.getTransactionFee());
    Customer.incrementTransactionFee();
    System.out.println(Customer.getTransactionFee());

    Customer xy = new Customer("XY");
    Customer tom = new Customer("tom", 100, 80);
}
```



Static Members vs Instance Members

static (class) members

There is **only one copy** of class fields regardless of the number of instances

Class fields are initialized **during compilation**

Class methods can **be invoked even without any instance being created**

Invoked by: ***ClassName.methodName(…)***

Instance members

Multiple copies of instance fields depending on the number of instances

Instance fields are **initialized when an instance is created at run time**

Instance methods can only be invoked after an instance is created

Invoked by: ***objectName.methodName(…)***

Access (Visibility) Modifiers

Access Modifiers

- Encapsulation is an important mechanism in OOP and it includes two relevant aspects:
  - Bundle the data with methods in a single unit
  - Control/Restrict access to some of the objects' members
- In Java, we accomplish encapsulation through the appropriate use of access (or visibility) modifiers
- Java has four visibility modifiers: public, protected, (no modifier – default modifier) and private
- The protected modifier involves inheritance, which we will discuss later

Access Modifiers

– Members of a class that are declared with ***public*** can be referenced/accessed anywhere

– ***public*** variables violate encapsulation because they allow the client to "reach in" and modify the values directly

– Members of a class that are declared with ***private*** can be accessed only within that class.

private members have class scope

– Members declared without a visibility modifier have *package visibility* and can be referenced by any class in the same package (A java package is a set of related classes)

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
(no modifier)	Y	Y	N	N
private	Y	N	N	N

Encapsulation: Fields

- Fields are normally private to protect/hide the internal structure from users
  - Other user classes cannot access these fields directly
  - Can be accessed from outside the class by using methods (getters & setters)
- Getters (accessors) return the current value of an instance variable
- Setters (mutators) change the value of a variable
- The names of getter and setter methods take the form getX and setX, respectively, where X is the name of the field

Encapsulation: Getters & Setters for Fields

– ***Fields*** can be accessed from outside the class by using getters/setters methods

```
public class  ATM{
   …

…
public static void main (String[] args){
     Customer xy = new Customer("xy", 1, 2);
    double bl=xy.chequeAccountBalance;   Error!
     System.out.println();

     bl=xy.getChequeAccountBalance();   Yes!

…

}
```

Not accessible by a client!

```
public class Customer {
  private double creditCardBalance;
  private double chequeAccountBalance;
  private String name;
  // …

  public String getName() {
         return name;
  }
  public double getCreditCardBalance() {
         return creditCardBalance;
  }
 public double getChequeAccountBalance() {
         return chequeAccountBalance;
  }
 public void setName(String nm){
       name = nm;
} … …
```

Access Modifiers for Methods

- Methods that provide the object's services are declared with public visibility so that they can be invoked by clients

- public methods are also called ***service methods***

- A method created simply to assist a service method is called a ***support method***

- Since a support method is not intended to be called by a client, it should NOT be declared with public visibility

ArrayList

ArrayList Class

- ArrayList class manages a sequence of objects
  - Is a part of the ***java.util*** package (`import java.util.ArrayList;`)
  - Can dynamically grow and shrink as needed
  - Elements are accessed and stored with an index
  - Like primitive arrays, indexes start at 0.

Object 0	Object 1	…	…	Object n-1

  - ArrayList class provides methods for many common tasks, such as removing and adding elements

ArrayList

- The ArrayList class is a generic class:

  `ArrayList<TypeParameter>` //an array list type

- For example:
  - **ArrayList<Customer>** // an array list of **Customer** type

```
ArrayList<Customer> customers = new ArrayList< Customer >();
customerList.add(new Customer("Peter", -1276, 423));
customerList.add(new Customer("Mary", -254, 1765));
customerList.add(new Customer("Paul", -3124, 102));
```

  - You can replace *Customer* with any other **class** to get a different array list type
- When you construct an ArrayList object, it has an initial size of 0.
- An arraylist has a set of methods for common operations
  - You can use **add()** method to add an object to the end of the array list.
  - **size()** method returns the current size of the array list.

ArrayList

*The **`get(int i)`** method retrieves the object at location **i**.*

	creditCard Balance	chequeAccount Balance	name
0	-1276	423	Peter
1	-254	1765	Mary
2	-3124	102	Paul

```
ArrayList <Customer> customerList = new ArrayList <Customer> ();
customerList.add(new Customer("Peter", -1276, 423));
customerList.add(new Customer("Mary", -254, 1765));
customerList.add(new Customer("Paul", -3124, 102));
System.out.println(customerList.get(1).getName());
```

Prints → Mary

Some more methods

– ***set(int, object)*** overwrites an existing object at the given index with another specified object

-1276	423	Peter
-254	1765	Mary
-3124	102	Paul

changes the third element of ***customerList***

```
customerList.set(2, new Customer("Sam ", 10,100));
```

0	-1276	423	Peter
1	-254	1765	Mary
2	**10**	**100**	**Sam**

Some more methods

– *add(object)* adds the element to the end of the ArrayList

0	-1276	423	Peter
1	-254	1765	Mary
2	-3124	102	Paul

Adds to the *end* of ***customerList***

customerList.add(new Customer("Sam " , 10,100));

0	-1276	423	Peter
1	-254	1765	Mary
2	-3124	102	Paul
3	**10**	**100**	**Sam**

Note: Size increased

Some more methods

– *add(int, object)* adds the object at the index specified, shifting down all other entries

0	-1276	423	Peter
1	-254	1765	Mary
2	-3124	102	Paul
3	10	100	Sam

Adds at location 2

customerList.add(2, new Customer(“Juni ”, -23,263));

0	-1276	423	Peter
1	-254	1765	Mary
2	**-23**	**263**	**Juni**
3	-3124	102	Paul
4	10	100	Sam

Note:

- Size increased
- Records are moved down

Some more methods

– ***remove(int)*** removes an element

0	-1276	423	Peter
1	-254	1765	Mary
2	-23	263	Juni
3	-3124	102	Paul
4	10	100	Sam

Removes element at location 1

customerList.remove(1);

Note:

- Size decreased
- Records are moved up

0	-1276	423	Peter
1	-23	263	Juni
2	-3124	102	Paul
3	10	100	Sam

A code cliché: traversing a collection

– Traversing all elements of an ArrayList object:

```
ArrayList<Customer> customerList= . . . ;
int sum = 0;
for (Customer c : customerList) {
    sum = sum + c.getCreditCardBalance();
}
```

"For each Customer object ***c*** in the ***customerList***"

Finding the Maximum or Minimum

- Initialize a candidate with the starting element
- Compare candidate with remaining elements
- Update it if you find a larger or smaller value.

```
if (!customerList.isEmpty()){
    int max = customerList.get(0).wealth();
    String richestPerson = customerList.get(0).getName();
    for (Customer c : customerList ) {
      if (c.wealth() > max) {
            richestPerson = c.getName();
            max = c.wealth();
      }
    }
}

    System.out.println("Richest person is " + richestPerson);
```

Get the starting object in customerList

Invoke its instance method

Finding a Value

- Check all elements until you find the value or reach the end of the array list

```
public Customer find(String name)
  {
     for (Customer c : customerList )
     {
        if (c.getName().equals(name))
              return c; // Found a match return c;
     }
     return null; // No match in the entire array list
  }
```

Primitive Array vs ArrayList

Primitive Array	**ArrayList:** Class from java.util package, need to import the package by: **import java.util.*;**
type: primitive or class	type: class only, use wrappers for primitive
int[] myArray = new int[10];	**ArrayList<type> myArrayList = new ArrayList<type>();**
capacity is predetermined	Capacity can be decreased or increased dynamically
Size: myArray.length;	Size method: myArrayList.size();
Assignment: myArray[index]=anValue;	Assignment: set method myArrayList.set(index, anObject);

Inserting or removing:
by programmer (might use loops)
	Inserting or removing: add or remove methods
	myArrayList.add(anObject);
	myArrayList.remove(index);

Questions?