

ET0736

Lesson 4

Inheritance, Polymorphism and ArrayList

Topics

- Inheritance (extends)
- Class Object
- Method Overriding – toString(), equals()
- Array of Objects
- super()
- this()
- Polymorphism
- ArrayList and sorting
- Generic class and method

Scenario

Consider a real world example which would be able to take advantage of the inheritance structure in Java programming.

Example:

Singapore Polytechnic Library serves 2 types of members, **students** and **staff**. Assuming that in our program, we are interested in capturing the following information:

	Student	Staff
attributes	<ul style="list-style-type: none">- name- ID- department- quota- sClass	<ul style="list-style-type: none">- name- ID- department- quota- title
methods	<ul style="list-style-type: none">+ setQuota()+ getQuota()	<ul style="list-style-type: none">+ setQuota()+ getQuota()

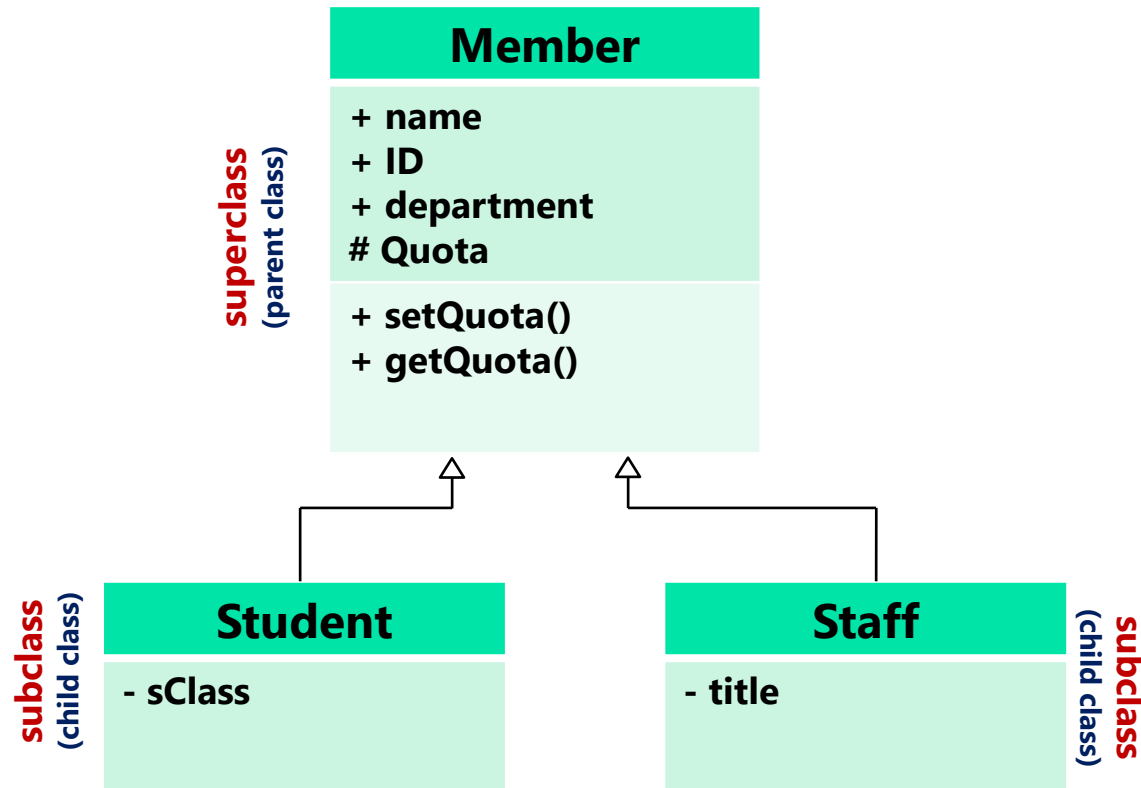
To model **students** and **staff** in the program, the most straight forward way is to create a class for **Student** and another class for **Staff**.

Scenario

However, one can easily observe that there would be many common characteristics between these 2 classes.

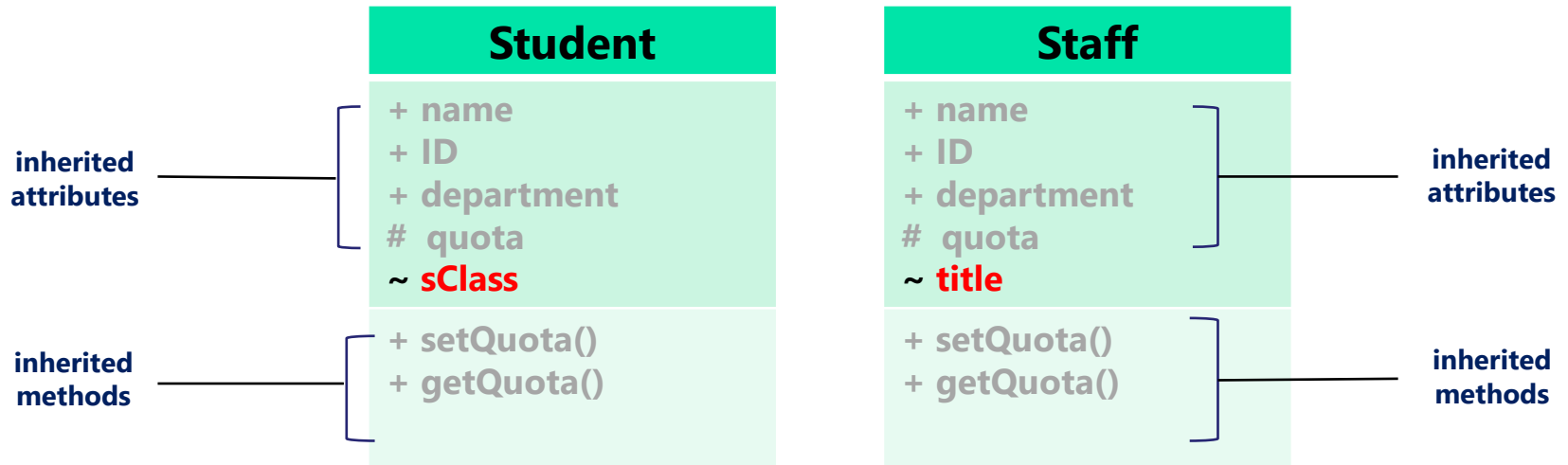
This is the more efficient way:


- * To define all these common characteristics in a **superclass**, called **Member**.
- * Next, to define class **Staff** and **Student** as **subclass** of **Member**



Scenario

- Once the class **Staff** and **Student** are defined as **subclass** of **Member**, these classes automatically **inherit** all the characteristics specified in **Member** without further specification of these characteristics.
- In addition, class **Staff** and **Student** can have their own individual unique characteristics defined within themselves.





```
class Member {
    public String name;
    public String ID;
    public String department;
    protected int quota;

    public void setQuota(int q) { quota = q; }
    public int getQuota() { return quota; }
}
```

class Member

```
class Student extends Member {
    private String sClass;
    public String getSClass() { return sClass; }
    public void setSClass(String sClass) {
        this.sClass = sClass;
    }
}
```

extends

```
class Staff extends Member {
    private String title;
    public String getTitle() { return title; }
    public void setTitle(String title) {
        this.title = title;
    }
}
```

**Student class and
Staff class are both
subclass/child
class Member**



Run the code from the previous slides together with the following:

```
public class Test {  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        s1.name="Koko";  
        s1.ID="p1234567";  
        s1.department="EEE";  
        s1.setSClass("DCPE/FT/1A/01");  
        s1.setQuota(10);  
        s1.setQuota(20);  
        System.out.println(s1.getSClass());  
        System.out.println(s1.getQuota());  
    }  
}
```

superclass : **Member**
subclasses: **Student** and **Staff**

S1 is an object of
Student class

S1 inherits
methods and
attributes from its
parent class, i.e.
Member class

Output:

DCPE/FT/1A/01

20

Inheritance

The 4 characteristics commonly associated with OOP:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Object Class

- The Object class provides many methods, including the *toString()* method and *equals()* method .
- The Object class is the parent class of all the classes in java by default.
- In other words, it is the topmost class of java. It is useful when there is a need to refer any object whose type is unknown.

toString()

```
package FirstPackage;

public class Test{
    public static void main(String[] args) {
        Member m1 = new Member();
        m1.name="Koko";
        m1.ID="p1234567";
        m1.department="EEE";
        m1.setQuota(10);

        System.out.println(m1);
        System.out.println(m1.toString());
    }
}
```

Printing the object by its reference is actually calling the inherited toString() from the Object class

Output:

```
FirstPackage.Member@5f184fc6
FirstPackage.Member@5f184fc6
```

Overriding toString()

Overriding is redefining the logic for an inherited.
Add an overriding **toString()** to class member.

```
class Member {  
    public String name;  
    public String ID;  
    public String department;  
    protected int quota;  
  
    public void setQuota(int q) { quota = q; }  
    public int getQuota() { return quota; }  
  
    public String toString() {  
        return("Name:"+name + "," +  
            "ID:"+ ID + "," +  
            "Dept:" + department +  
            "," + "Quota: " + quota);  
    }  
}
```

**Overriding
method toString()
in class Member**

Run this code again with the previous slides (modified class Member):

```
package FirstPackage;

public class Test{
    public static void main(String[] args) {
        Member m1 = new Member();
        m1.name="Koko";
        m1.ID="p1234567";
        m1.department="EEE";
        m1.setQuota(10);

        System.out.println(m1);
        System.out.println(m1.toString());
    }
}
```

Output:

Name:Koko,ID:p1234567,Dept:EEE,Quota: 10

Name:Koko,ID:p1234567,Dept:EEE,Quota: 10

Observe the output.

**The overriding
toString() in the class
Member is triggered**

equals()

- Commonly used method defined in Object class
- The **default** method checks if 2 references point to the same object
- Usually overridden in custom class to test if 2 objects are having the exact same content

```
public class Test {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(7);  
        Circle c2 = new Circle(7);  
        Circle c3 = c1;  
        System.out.println(c1.equals(c2));  
        System.out.println(c1.equals(c3));  
    }  
}
```

```
class Circle {  
    private int radius;  
  
    Circle (int radius) {  
        this.radius = radius;  
    }  
    public double getArea() {  
        return (radius*radius*22.0/7.0);  
    }  
}
```

Output:

false
true

Overriding equals()

```
public class Test {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(7);  
        Circle c2 = new Circle(7);  
        Circle c3 = c1;  
        System.out.println(c1.equals(c2));  
        System.out.println(c1.equals(c3));  
    }  
}
```

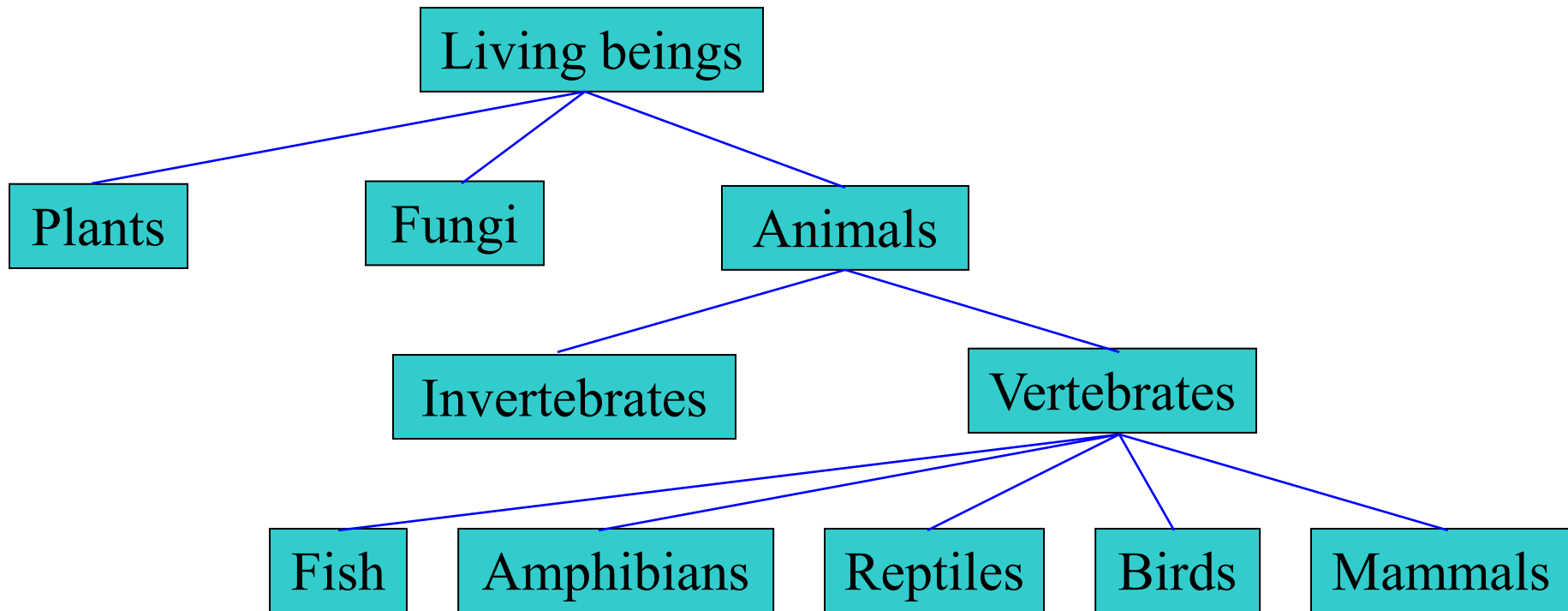
**Overriding
method equals()**

Output:
true
true

```
class Circle {  
    private int radius;  
  
    Circle (int radius) {  
        this.radius = radius;  
    }  
    public double getArea() {  
        return (radius*radius*22.0/7.0);  
    }  
  
    public boolean equals(Circle x) {  
        if (x.radius == radius)  
            return (true);  
        else  
            return (false);  
    }  
}
```

Inheritance Hierarchy

- Inheritance exists in the real world



Inheritance

The 4 characteristics commonly associated with OOP:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Inheritance allows creation of new classes by reusing existing classes and adding additional attributes and methods.

However, for proper OOP design, the relations between the parent and child class should be “is-a” instead of “has-a”.

For instance, a sub class SportsCar can be design to inherit from parent class Car as a sports car is also a type of car. However, a sub class hardDisk is not appropriate to be inherited from class Computer as a computer has a hard disk but a hard disk is not a computer.

Access Modifier

If a subclass is defined in a different package from the parent class, access to attributes or methods is limited to public or protected:

Access Levels				
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

* Default – no modifier

UML convention

+ public
- private
protected
~ default


super() call in Subclass

super()

- To be used in sub class's constructor
- To call the *no-arg* constructor in the immediate super class
- Can happen *automatically* in the sub class's *constructor*
- Can also be invoked explicitly

Auto super() call

Consider the superclass **Person** and the subclass **Employee** below.

```
public class Test {  
    public static void main(String[] args) {  
        Employee e1 = new Employee();  
    }  
}  
  
class Person {  
    Person() { System.out.println ("Person's no-arg constructor"); }  
}  
  
class Employee extends Person {  
    Employee() {  
          
        System.out.println("Employee's no-arg constructor");  
    }  
}
```

Output:

Person's no-arg constructor
Employee's no-arg constructor

Observe the output.

There is a hidden automatic **super()** call at the beginning of the sub class constructor before the 1st statement.

Auto super() call

```
public class Test {  
    public static void main(String[] args) {  
        Employee e1 = new Employee(33);  
    }  
}  
  
class Person {  
    Person() { System.out.println ("Person's no-arg constructor");}  
}  
  
class Employee extends Person {  
    Employee() {  
        System.out.println("Employee's no-arg constructor");  
    }  
    Employee(int x) {  
        System.out.println("Employee's 2nd constructor");  
    }  
}
```

**2nd
constructor**

Output:

Person's no-arg constructor
Employee's 2nd constructor

Observe the output.

Chain of super() call

Try out this program with 3 levels of inheritance and observe the chain effect of automatic **super()** call.

```
public class Test {  
    public static void main(String[] args) {  
        Faculty f1 = new Faculty();  
    }  
}  
  
class Person {  
    Person() { System.out.println ("Person's no-arg constructor"); }  
}  
  
class Employee extends Person {  
    Employee() {  
        System.out.println("Employee's no-arg constructor");  
    }  
}  
  
class Faculty extends Employee {  
    Faculty() {  
        System.out.println("Faculty's no-arg constructor");  
    } }  
}
```

By adding **super(10)** call statement (**must be 1st statement**) in Employee(), the automatic hidden super() is disabled.

```
public class Test{
    public static void main(String[] args) {
        Employee e1 = new Employee();
    }
}
class Person {
    Person() { System.out.println ("Person's no-arg constructor"); }
    Person(int a ) { System.out.println ("Person's 2nd constructor"); }
}

class Employee extends Person {
    Employee() {
        super(10);
        System.out.println("Employee's no-arg constructor");
    }
}
```

Matching super Call

An automatic or explicit ***super*** call from the *subclass* will activate the ***matching superclass*** constructors.

The match is determined by the number and data types of the arguments of the methods.

E.g. Person and Employee class

call from subclass Employee	Matching constructor in superclass Person
<code>super()</code>	<code>Person()</code>
<code>super(int)</code>	<code>Person(int)</code>
<code>super(int, String)</code>	<code>Person(int, String)</code>

An explicit ***super*** call from the *subclass* must be added as the 1st statement and the auto `super()` will be disabled.

Implicit super constructor is missing

Error!

```
public class Test {  
    public static void main(String[] args) {  
        Employee e1 = new Employee();  
    }  
}  
  
class Person {  
    Person(int a ) { System.out.println ("Person's only constructor"); }  
}  
  
class Employee extends Person {  
    Employee() {  
        System.out.println("Employee's no-arg constructor");  
    }  
}
```


this() call in class constructor

this()

- To call the *no-arg* constructor of its own class
- To be used in constructor only
- If added, it must also be the *1st statement*.
- If added, the *automatic super()* will also be *disabled*

If an explicit **this()** call statement (must be 1st statement) is added in Employee(int a), the automatic super() will be disabled.

```
public class Test {  
    public static void main(String[] args) {  
        Employee e2 = new Employee(4);  
    }  
}  
  
class Person {  
    Person() { System.out.println ("Person's no-arg constructor"); }  
    Person(int a ) { System.out.println ("Person's 2nd constructor"); }  
}  
  
class Employee extends Person {  
    Employee() {  
        System.out.println("Employee's no-arg constructor");  
    }  
    Employee(int a) {  
        this();    // calling Employee()  
        System.out.println("Employee's 2nd constructor");  
    } }  
}
```

Output:

```
Person's no-arg constructor  
Employee's no-arg constructor  
Employee's 2nd constructor
```

Polymorphism

The 4 characteristics commonly associated with OOP:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Polymorphism

- A Greek word meaning “*many forms*”.
- An object of a subclass can be used by any code designed to work with an object of its superclass.
- This is quite logical as a subclass inherited everything from the superclass. The subclass can do everything that is designed to be done by the superclass.

Polymorphism

Super class: **Animal**

Sub classes: **Fish** and **Dog**

```
public class Test {  
    public static void main(String[] args)  
    {  
        Animal a = new Animal();  
        a.move();  
  
        Fish f = new Fish();  
        Dog d = new Dog();  
        a = f;  
        a.move();  
        a = d;  
        a.move();  
    }  
}
```

```
class Animal {  
    void move()  
    {System.out.println("Somehow will  
move."); }  
}  
  
class Fish extends Animal {  
    void move() {System.out.println("I  
can swim!"); }  
}  
  
class Dog extends Animal {  
    void move() {System.out.println("I  
can walk, jump and run!"); }  
}
```

**Object f and d can be assigned to object a
as object a is Animal object (super class).**

f = a or d = a will cause error

Polymorphism (another example)

```
public class Test{
    public static void main(String[] args) {
        SportsCar sc1 = new SportsCar("BMW");
        System.out.println(sc1.getSpeed());
        stopVehicle(sc1);
        System.out.println(sc1.getSpeed());
    }

    public static void stopVehicle (Vehicle v) {
        v.setSpeed(0);
    }
}
```

Method stopVehicle is designed to take in a Vehicle object.,

Hence, it can also accepts SportsCar object.

Super class: **Vehicle**

Sub classes: **SportsCar**

```
class Vehicle {
    private int speed;
    Vehicle () { speed=50; }
    void setSpeed (int s) {
        speed = s;
    }

    int getSpeed () {
        return (speed);
    }
}

class SportsCar extends
    Vehicle {
    String brand;
    SportsCar(String b){
        brand = b;
        setSpeed(100);
    }
}
```

Summary - Polymorphism

- Polymorphism allows same method name be used in conjunction with different objects.
- Correct method chosen during running depends on the type of object it is associated with.

ArrayList

- Resides within Java Core Libraries (`import java.util.ArrayList`)
- Linear and dynamic
- Base on dynamic array (resizable)
- Can store different data types
- Stores data in consecutive memory location
- Supports randomly access for the elements (Index-based structure)

ArrayList

< > in ArrayList declaration indicates that the container is a generic class, i.e. able to take in different data types or objects.

Like other classes in Collection, ArrayList does not take primitive types.

To store primitive types in an ArrayList, the respective wrapper classes are used.

```
ArrayList<Integer> a1 = new ArrayList<Integer>();  
ArrayList<Double> a2 = new ArrayList<Double>();
```

New version of Java allows:

```
ArrayList<Integer> a1 = new ArrayList<>();  
ArrayList<Double> a2 = new ArrayList<>();
```

ArrayList – add new element

Example: Create an ArrayList to store only integers .

Method used: ***add()***

```
public static void main(String[] args) {  
    ArrayList<Integer> a1 = new ArrayList<>();  
  
    // insert elements into ArrayList  
    for (int i=1; i<=5; i++)  
        a1.add(i);  
  
    // a1.add("Hello");    // error for storing other data types  
  
    System.out.println(a1);  
}
```

Output:

```
[1, 2, 3, 4, 5]
```

ArrayList – remove element

Example: Remove an item.

Method used: ***remove(index)***, ***remove(object)***

```
public static void main(String[] args) {  
  
    ArrayList<Integer> a1 = new ArrayList<>();  
    for (int i=1; i<=5; i++)  
        a1.add(i*10);  
  
    a1.remove(2);  
    a1.remove(Integer.valueOf(50));  
    System.out.println(a1);  
}
```

Output:

```
[10, 20, 40]
```

ArrayList – insert new element

Example: Insert an item.

Method used: ***add(index,value)***

```
public static void main(String[] args) {  
  
    ArrayList<Integer> a1 = new ArrayList<Integer>();  
    for (int i=1; i<=5; i++)  
        a1.add(i);  
  
    a1.add(3,888);  
  
    System.out.println(a1);  
}
```

Output:

```
[1, 2, 3, 888, 4, 5]
```

ArrayList – retrieve element

Example: Retrieve an item.

Method used: ***get(index)***

```
public static void main(String[] args) {  
  
    ArrayList<Integer> a1 = new ArrayList<Integer>();  
    for (int i=1; i<=5; i++)  
        a1.add(i);  
  
    System.out.println(a1.get(3));  
}
```

Output:

4

ArrayList – update element

Example: Update an item.

Method used: ***set(index, Object)***

```
public static void main(String[] args) {  
  
    ArrayList<Integer> a1 = new ArrayList<>();  
    for (int i=1; i<=5; i++)  
        a1.add(i);  
  
    a1.set(0,999);  
  
    System.out.println(a1.get(0));    // 999  
}
```

Output:

999

ArrayList – sort by comparator

Example: Sorting ArrayList<Integer>.

Method used: ***sort(comparator)***

```
public static void main(String[] args) {  
  
    ArrayList<Integer> a1 = new ArrayList<>();  
    for (int i=1; i<=5; i++)  
        a1.add(6-i);    // 5,4,3,2,1  
  
    a1.sort(new SortIntegerList());  
}
```

```
class SortIntegerList implements Comparator<Integer> {  
    @Override  
    public int compare(Integer a, Integer b) {  
        return (a-b);  
    }  
}
```

ArrayList – other methods

Example: ArrayList<Integer>.

Other methods ***size()***, ***contains()***, ***isEmpty()***, ***clear()*** etc

```
public static void main(String[] args) {  
  
    ArrayList<Integer> a1 = new ArrayList<>();  
    for (int i=1; i<=5; i++)  
        a1.add(6-i);    // 5,4,3,2,1  
  
    a1.add(999);  
  
    System.out.println(a1.size());           // 6  
    System.out.println(a1.contains(999));    // true  
    System.out.println(a1.contains(888));    // false  
    System.out.println(a1.isEmpty());        // false  
  
    a1.clear();  
    System.out.println(a1.isEmpty());        // true  
}
```


ArrayList

ArrayList allows random access, i.e. an element can be accessed directly using the index value.

```
public static void main(String[] args) {  
  
    ArrayList<Integer> a1 = new ArrayList<>();  
  
    for (int i=1; i<=5; i++)  
        a1.add(i);  
  
    // supports random access  
    System.out.println(a1.get(2));    // get object at index 2  
  
    a1.remove(0);                      // remove object at index 0  
  
    System.out.println(a1.get(2));    // get object at index 2  
  
}
```

ArrayList

ArrayList has several constructors.

```
ArrayList<String> a1 = new ArrayList(Arrays.asList("Oppa", "Bobo"));
```

Unlike Array, ArrayList is not strongly typed.

To create an ArrayList to store mixed objects, use this declaration.

```
ArrayList a1 = new ArrayList();
```



Generic Class

- ***ArrayList*** is a generic class as it is able to handle different data types.
- For instance, the following statement creates a list of String:

```
ArrayList<String> a1 = new ArrayList<>();
```

- The following statement creates a list of Double:

```
ArrayList<Double> a2 = new ArrayList<>();
```

- Java allows custom generic classes and methods to be defined

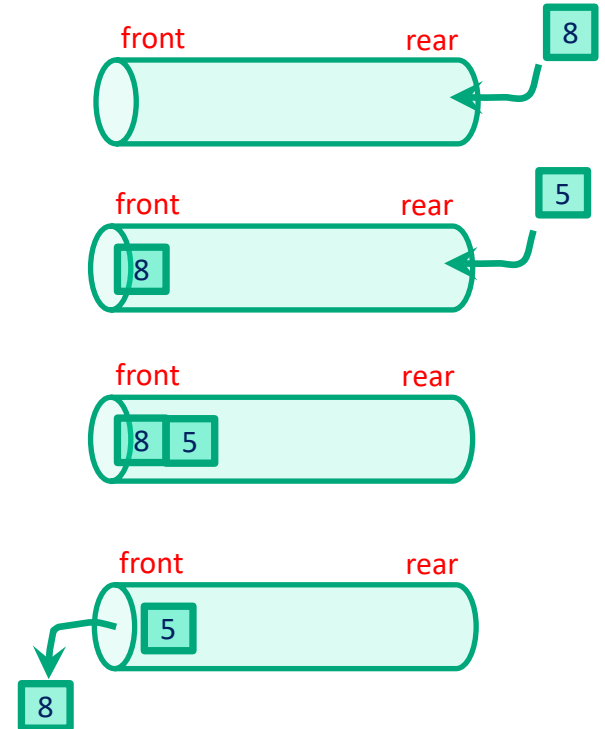
Custom Generic Class

Assuming that we want to define a custom class **MyQueue** to simulate a queue data structure, which is a First-In-First-Out protocol – just like a line in front of a hawker stall.

Assuming that the class is supposed to support these 3 simple operations:

<i>addToRear()</i> : void
<i>popFromFront()</i> : E
<i>size()</i> : int

Assuming that internally, we use an ArrayList to store the elements.



Custom Generic Class

Use <E> to represent the generic type.

Class MyQueue

```
class MyQueue<E>{
    ArrayList<E> a = new ArrayList<>();

    void addToRear(E newObj){
        a.add(newObj);
    }

    E popFromFront() {
        E obj;
        if (a.size()>0){
            obj = a.get(0);
            a.remove(0);
            return (obj);
        }
        return(null);
    }

    int size(){
        return a.size();
    }
}
```

Test program

```
public static void main(String[] args) {
    MyQueue<Integer> q1 = new MyQueue<>();
    MyQueue<String> q2 = new MyQueue<>();
    q1.addToRear(111);
    q1.addToRear(222);
    q1.addToRear(333);
    q2.addToRear("SP");
    q1.popFromFront();

    System.out.println (q1.size());
    System.out.println (q2.size());
}
```

Output:

```
2
1
```

Custom Generic Method

Using the generic class MyQueue.

Create the following method to convert an array into a MyQueue object.

```
public static <E> MyQueue<E> fromArrayToMyQueue(E arr[]){  
    MyQueue<E> a = new MyQueue<>();  
    for (int i=0; i<arr.length; i++){  
        a.addToRear(arr[i]);  
    }  
    return(a);  
}
```

Output:

5

```
public static void main(String[] args) {  
    Integer a[] = {111,222,333,444,555};  
    MyQueue<Integer> q1 = fromArrayToMyQueue(a);  
    System.out.println(q1.size());  
}
```

**Test
program**

Summary of modifier

Modifier	class	variable	method	constructor
public	yes visible to all	yes visible to all	yes visible to all	yes visible to all
protected	N.A.	yes, visible in subclass and same package	yes, visible in subclass and same package	yes, visible in subclass and same package
(default)	yes, visible in same package only	yes, visible in same package only	yes, visible in same package only	yes, visible in same package only
private	N.A.	yes, only visible to current class	yes, only visible to current class, cannot be overriden	yes, but cannot be used to create object
final	yes, but cannot be subclassed	yes, but cannot be changed	yes but cannot be overridden	no
static	N.A.	yes Attached to class and shared by all instance	yes Attached to class, cannot be overridden	no

OOP Terminology

OOP Term	Definition
method	Same as <i>function</i> /behaviour. To call a method: <code>x.f(y)</code> , where <code>x</code> is an object of class that contains <code>f</code> method.
send a message	Call a method, invoke/use a method
instantiate	Create object/instance from template class with new
class	A template/blueprint/structure with both data and functions. A user-defined complicated data type
object	Concrete thing/instance created from class with new . Memory is allocated to it.
member	A data field or method is a member of a class if it's defined in that class
constructor	A special method that creates and initializes new objects. It has the same name as class name, no return data type, not even void. Can be overloaded.



OOP Terminology

OOP Term	Definition
Inheritance	Defining a class (child) in terms of another class (parent). All of the public members of the public class are available in the child class.
Polymorphism	Defining functions with the same name, but different parameters.
overload	A method is overloaded if there is more than one definition(same method name but different parameter in the same class). See polymorphism.
override	In a child class, redefine a method from a parent class.(same method name and parameter, different implementation in child and parent class)
subclass	Same as child, derived, or inherited class
superclass	Same as parent or base class.
attribute	Same as data member/member field/property