
CPE203

JAVA CLASS

Contents

Definitions:	1
Array:	1
ArrayList:	2
HashMap:	2
Example:	3
Example:	3
Overriding:	4
toString():	4
equals:	4
Hash Code:	5
Upcasting and Downcasting:	5
Super and Subclass:	6
Comparable && Comparator:	7
Lambda:	8
Predicate:	8
Stream:	8
Try & Catch:	9

Definitions:

- **OOP** is a programming technique that focuses on the data(=objects) and on the interfaces to that object.
- **Object:** Is the instance of the class
- **Object Class:** is the *ultimate super class*. Every class you ever make in Java will automatically inherit from it.(equals, toString, and getClass)
- **Classes:** A class is the template or blueprint from which objects are made.
 - Static variable:
 - * Belong to the class
 - * Accessed by: ClassName.VARIABLE_NAME
- **Superclass:** class above another in a hierarchy of classes
- **Subclass:** inherits from parent and is a version of the parent class
- **Encapsulation(sometimes called information hiding):** is simply combining data and behavior in one package hiding the implementation details from the users of the objects.
- **Instance variables:** are variables defined in a class, but outside the body of methods. Instance variables are filled when each object is instantiated and belong to the object.
- **Class Variables:** belongs to the class and the value in that variable is shared by every instance of the class by the class itself.
 - **Instantiation:** the object is an instance of a class.
- **Constructor:** set data's values
- **Methods:** Manipulate and access data
- **Public:** is keyword which declares a member's access as public.
- **Private:** is a Java keyword which declares a member's access as private.
- **Static:** object belongs specifically to the class, instead of instances of that class.
 - **functional interface:** An interface with only one abstract method, so it can be implemented with a lambda.
- **Is-a:** inheritance/interfaces
- **has-a:** composition/aggregation

Three key characteristics of objects:

- The object's behavior
- The object's state
- The object's identity

Array:

```
int[] array= new int[20]; // allocating memory for array. it will be fixed sized
array[0]=5; //declaring first element array
```

```
//for loop - if nums is Array
for (int i = 0; i < nums.length i++){
    //nums[i];
}
```

```
//for each loop
for (int i: nums){
```

```
    //do stuff with i
}
```

ArrayList:

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

```
ArrayList<Integer>num=new ArrayList<Integer>();
```

```
LinkedList<String> ll = new LinkedList<>();
```

```
num.add(1);
```

```
num.get(0); // we need put index
```

```
//remove element base on index
```

```
num.remove(1);
```

```
//for loop - if nums is ArrayList
```

```
for (int i = 0; i < nums.size(); i++){
    nums.get(i);
}
```

```
//for each loop
```

```
for (int i: nums){
    //do stuff with i
}
```

HashMap:

```
Map<String,String>myMap=new HashMap<>();
```

```
// add key
```

```
myMap.put("Hadi", "21");
```

```
// get the value of the key
```

```
myMap.get("Hadi");
```

```
// remove the key
```

```
myMap.remove("Hadi");
```

```
// clear whole Map
```

```
myMap.clear();
```

```
// get the size
```

```
myMap.size();
```

```
// Different way of loop
```

```
for (String name: myMap.keySet()){
    System.out.println(name);
}
```

```
for (String age: myMap.values()){
    System.out.println(age);
}
```

```

}

for(Map.Entry<String,String>entry:myMap.entrySet()){
    String key=entry.getKey();
    String value=entry.getValue();
}

```

Example:

```

class Trainer{
    private String id;
    private String name;

    public Trainer(String id, String name)
    {
        this.id = id;
        this.name = name;
    }
    public String getID() { return id; }
    public String getName() { return name; }
}

public static void main(String[]args){

    Map<String,Trainer> train=new HashMap<>();

    train.put("red",new Trainer("40","Hadi"));
    train.put("blue",new Trainer("401","Had"));
    train.put("Yellow",new Trainer("402","Ha"));
    train.put("black",new Trainer("403","H"));

    for(Map.Entry<String,Trainer>data:train.entrySet()){

        System.out.println("Color: "+(String)data.getKey()+" id: "+(String)data.getValue().getID());
    }

}
}

```

Example:

```

import java.util.LinkedList;
import java.util.List;
import java.util.Map;

class ExampleMap
{
    public static List<String> highEnrollmentStudents(
        Map<String, List<Course>> courseListsByStudentName, int unitThreshold)
    {
        List<String> overEnrolledStudents = new LinkedList<>();

        /*
         Build a list of the names of students currently enrolled
         in a number of units strictly greater than the unitThreshold.
        */
    }
}

```

```

    for(String name:courseListsByStudentName.keySet()){
        int sum=0;
        for(Course course:courseListsByStudentName.get(name)){
            sum+=course.getNumUnits();
        }
        // int unit=courseListsByStudentName.get(name).getNumUnits();
        if (sum>unitThreshold){
            overEnrolledStudents.add(name);
        }
    }

    return overEnrolledStudents;
}
}

```

Overriding:

toString():

```

    public String toString(){ return name;}

```

equals:

```

public boolean equals(Object o){
    if (o==null){return false;}
    if (o.getClass()!=this.getClass()){return false;}
    Theater t=(Theater)o;
    return t.seatingCapacity==seatingCapacity && t.numberTicket==numberTicket && t.name.equals(name);
}

```

OR

```

private final String prefix;
private final String number;
private final int enrollment;
private final LocalTime startTime;
private final LocalTime endTime;

public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass()) return false;
    CourseSection that = (CourseSection) o;
    boolean result=true;
    if (prefix==null)
        result=that.prefix==null;
    else
        result=prefix.equals(that.prefix);
    if (number==null)
        result=result && that.number==null;
    else
        result= result && number.equals(that.number);
    if (startTime==null)
        result= result && that.startTime == null;
}

```

```

    else
        result= result && startTime.equals(that.startTime);
    if (endTime==null)
        result= result && that.endTime==null;
    else
        result= result && endTime.equals(that.endTime);

    return result && enrollment == that.enrollment;
}

```

Hash Code:

```

public int hashCode()
{
    int hash = 1;

    hash = hash * 31 + studentLoans; //can leave since int
//cannot add double to int
    hash = hash * 31 + ((Double)gpa).hashCode();
    return hash;
}

```

OR

```

public int hashCode() {
    int hash=1;
    hash=hash *31+((prefix==null)?0:prefix.hashCode());
    hash=hash*31+((number==null)?0:number.hashCode());
    hash=hash*31+enrollment;
    hash=hash*31+((startTime==null)?0:startTime.hashCode());
    hash=hash*31+((endTime==null)?0:endTime.hashCode());
    return hash;
}

```

OR

```

public int hashCode() {
    int hash=1;
    hash=hash *31+Objects.hashCode(surname);
    hash=hash*31+Objects.hashCode(givenName);
    hash=hash*31+age;
    hash=hash*31+Objects.hashCode(currentCourses);
    return hash;
}

```

OR

```

public int hashCode()
{
    return Objects.hash(studentLoans, gpa);
}

```

Upcasting and Downcasting:

The right side should be less than or equal left side.

We can not instantiate the interface.

Will it compile:

- **Check the static types**, do they all match:
 - Are you only calling the methods of the static type?
 - Are you only passing in parameters of (or lower than) the expected static type. (You can upcast here, but not implicitly downcast)
 - Is the left side \geq the right?
- Will it maybe crash:
 - Did you have to explicitly downcast to call a method.
 - Will it maybe work, but you don't know for sure because you didn't check?

Super and Subclass:

```
public class Vehicle{

    private int maxSpeed=120;

    public Vehicle(int maxSpeed){
        this.maxSpeed=maxSpeed;
    }

    public void vroom(){
        System.out.println("Vroom vroom")
    }
    // Override the equals method
    public boolean equals(Object o){
        if (o==null){return false;}
        if (o.getClass()!=this.getClass()){return false;}
        Vehicle t=(Vehicle)o;
        return t.maxSpeed==maxSpeed;
    }

    // Override toString method
    public String toString(){
        return "The Vehicle speed: "+ maxSpeed;
    }
}

public class Car extends Vehicle{
    private int doors;

    public Car(int doors,int maxSpeed){
        super(maxSpeed);
        this.doors=doors;
    }

    public void display(){
        System.out.println(super.maxSpeed)
    }

    public void vroom(){
        super.vroom();
    }
    // Override equals method
```



```

    public boolean equals(Object o){
        return super.equals(o) && ((Car)o).doors==doors;
    }

    // Override toString method
    public String toString(){
        return super.toString()+"The number of doors"+doors;
    }
}

```

Comparable && Comparator:

This will compare the Objects and sort them. We need to make a separate class for Comparator which they call functional interface which can hold only on abstract method.

```

// important note that Comparable is only input one Object
public class Student implements Comparable<Student>{

    private String lastName;
    private String firstName;
    private int age;
    private double gpa;

    public int compareTo(Student other){
        return lastName.compareTo(other.lastName);
    }
}

// Comparator implements two Objects
public class StudentAgeComparator implements Comparator<Student>{
    public int compare(Student s1, Student s2)
    {
        return s1.age() - s2.age(); //compare ascending order
    }
}

public class StudentGpaComparator implements Comparator<Student>
{
    //compare reverse order
    public int compare(Student s1, Student s2)
    {
        if (s1.gpa() > s2.gpa())
            return -1;
        else if (s1.gpa() < s2.gpa())
            return 1;
        else
            return 0;
    }
}

```

for sorting we have to methods:

```
Collections.sort(studentList);
```

```
Arrays.sort(studentArray);
```

```
Collections.sort(studentList, new StudentGpaComparator()); // we can specify which Comparator we want to use
```

Lambda:

Unnamed chunk of code I can pass around. It is a shortcut to implement a functional interface's method. We can make comparator in just one line.

```
Comparator<Student> comp2=(Student s1, Student s2)->{return s1.age()-s2.age();};
Collections.sort(studentList,comp2)
Collections.sort(studentList,(s1, s2)->s1.age()-s2.age());
```

//second way: Key extractor

```
Comparator<Student> comp2=(s1, s2)->s1.age()-s2.age();
Comparator<Student> comp3=Comparator.comparing(s->s.age());
Comparator<Student> comp4=Comparator.comparing(s::age());
```

// Third way

```
Funtion<Student,String>f=Student::getName; // This have a return types
Consumer<String>p=System.out::println; // Consumer don't have return types
Consumer<String>p=s ->System.out.println(s);
```

Predicate:

```
public static void usePredicate(Predicate<Student> pred, Student s)
{
    // .test(s) is method that check if the input value match with Predicate and return boolean
    if(pred.test(s))
        System.out.println("yay");
    else
        System.out.println("no!");
}
```

Stream:

```
List<Student> theStudents = new ArrayList<>();
List<Student> topStudents =
    theStudents.stream()
        .filter(s -> s.getGpa() >= 3.5)
        .collect(Collectors.toList());

topStudents.stream().forEach(PrintStudentName);

System.out.println("Average Student GPA: " +
    theStudents.stream()
        .mapToDouble(Student::getGpa)
        .sum()
        /theStudents.size());

List<Student> raisedStudents =
    theStudents.stream()
        .filter(s -> s.getGpa() < 2.75)
        .map(s -> new Student(s.getName(),s.getAge(),s.getGpa() + .15))
        .collect(Collectors.toList());
```

```
double avg = profs.stream()
    .filter(Professor::hasTenure)
    .mapToDouble(Professor::getMortgage).average().getAsDouble();
```

The difference between Abstract, Interface, Concrete:

Interface and abstract class could not be instantiated, but in the concrete class we can instantiate. Interface only methods are declared; however, in the abstract class we can have some abstract methods and non-abstract methods.

Try & Catch:

What is the difference between checked and unchecked?

For checked, the compiler makes you catch the error and / or report that an error might be thrown. For unchecked, the compiler does not enforce those rules, so if not caught the program will crash.

Runtime exception:

- bad casting
- out of bounds
- null pointer

These are the exceptions that we can handle ourselves.

IOException:

This is not our fault. The compiler could not open the file.

important: the **Error** and **Runtime exception** are both unchecked and **IOException** is checked. The checked one the compiler forces us to check that exception. We can throw it or we can use try & catch to handle it.

Example:

“java

```
public class Circle { public static final double PI = 3.14159; private double radius;

public Circle()
{
    Random rand = new Random();
    radius = rand.nextDouble()*10;
}

public Circle(double radius) throws ZeroRadiusException, NegativeRadiusException
{
    if (radius==0)
        throw new ZeroRadiusException();
    if(radius<0)
        throw new NegativeRadiusException(radius);
    this.radius=radius;
}

public double radius()
{
    return radius;
}
...
}
```