Advanced Programming

→ Object Oriented

HELLO!

I am Hamzeh Asefan

Master Degree from King Abdullah II School of Information Technology in Information Systems (2020/2021) / The University of Jordan.

Features of Java Programming Language

Portable

The ability to run a program on different machines.

Simple

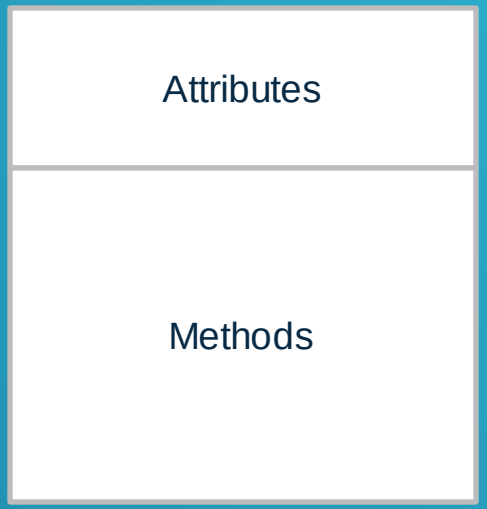
Derives much of its syntax from the C and C++ programming languages.

Object Oriented

Inheritance, Encapsulation, Abstraction, Polymorphism.

Class

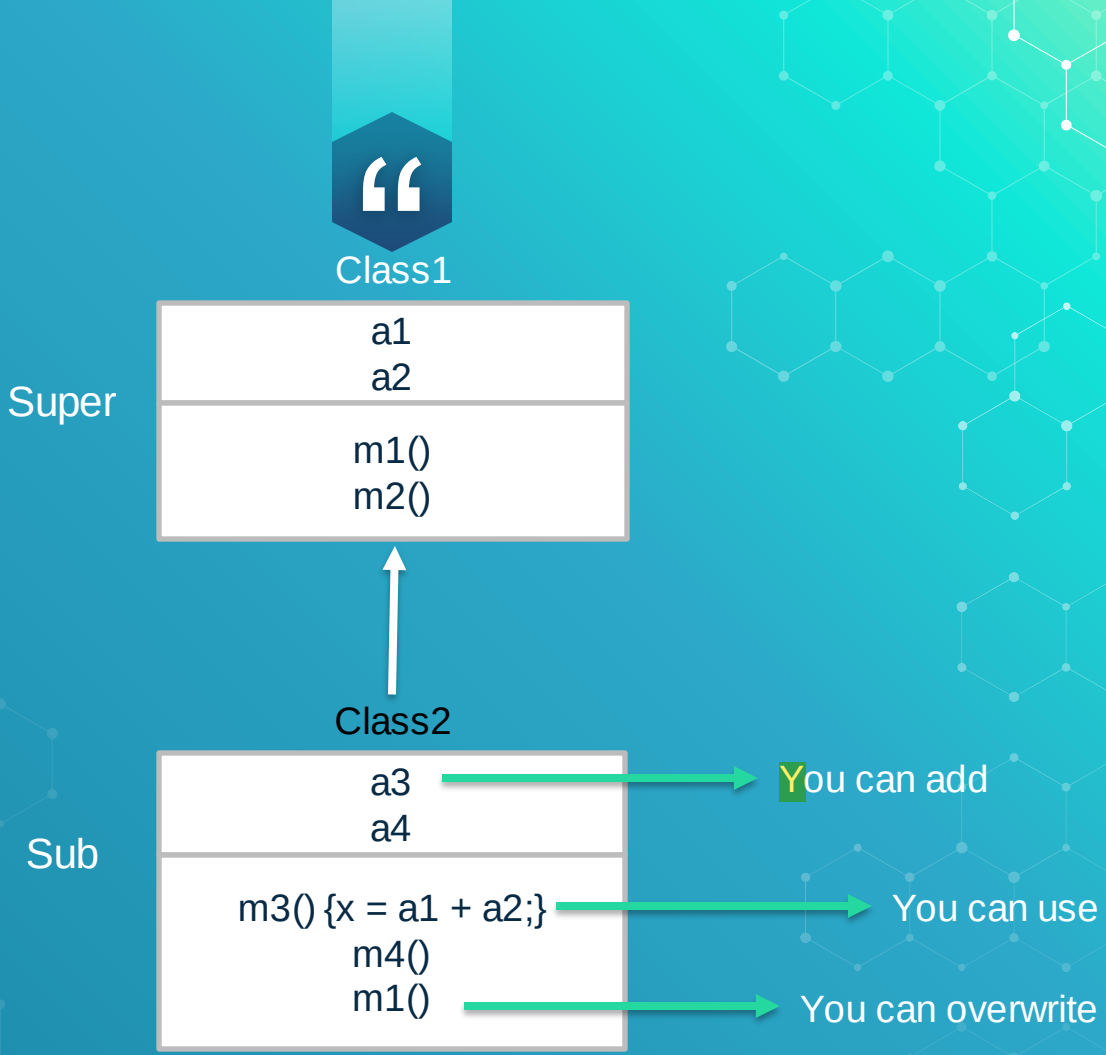
A program contains attributes, methods and classes.



Inheritance

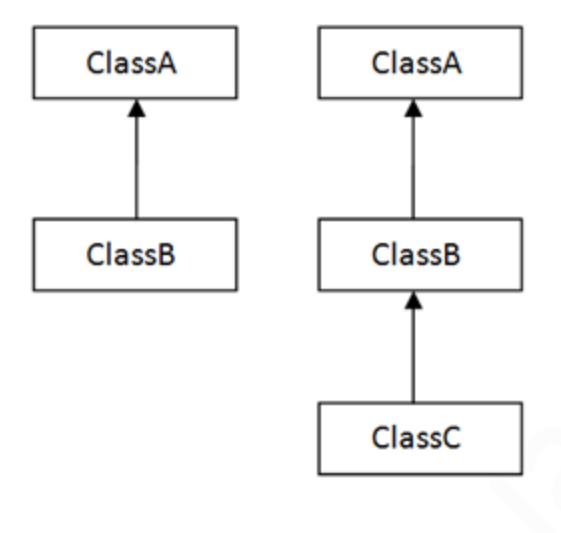
- You can use
- You can add
- You can overwrite

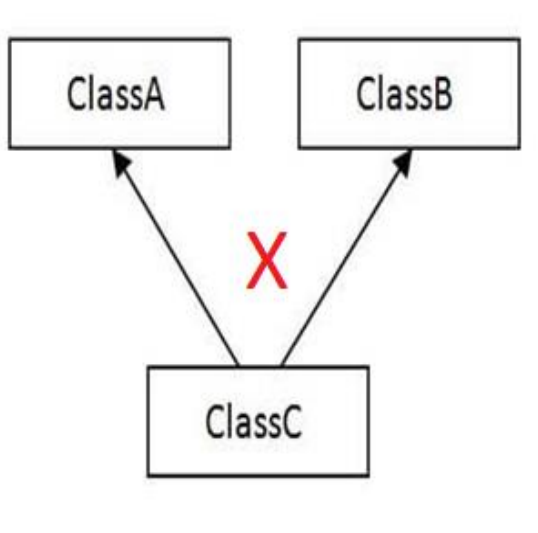
→ The keyword used for inheritance is **extends**.


Class1
Super
a1
a2
m1()
m2()
Class2
Sub
a3
a4
m3() {x = a1 + a2;}
m4()
m1()
You can add
You can use
You can overwrite

Single inheritance

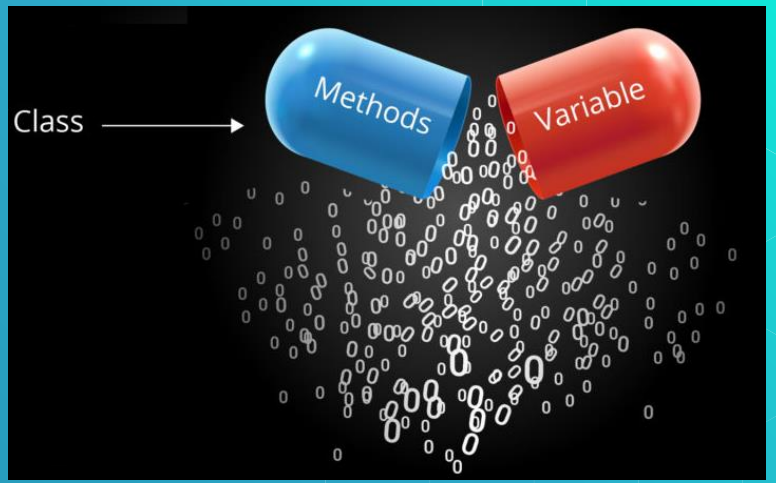
To reduce the complexity and simplify the language, multiple inheritance is not supported in java.



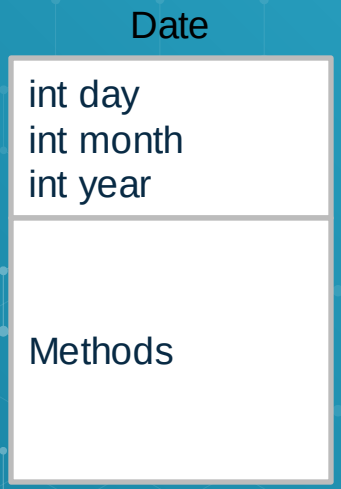


Encapsulation

- Refer to a mechanism of restricting the direct access to some components of an object.



Encapsulation



Date.day = 1

Date.month = 11

Date.year = 2021

Date.day = 35

Date.month = 17

Encapsulation

Date

```
private int day;
private int month;
private int year;
```

```
public setDay(…) {….}
public getDay() {….}
```

Date.day = 35 X → Date.setDay(35);

Encapsulation

- Members of a class that are declared private are not inherited by subclasses.
- Private can be accessed only by code in the same class.
- Methods can be private.

Abstraction

- The process of reducing the object to its essence so that only the necessary characteristics are exposed to the users.
- Different implementations for same method.
- Contract.

Class1

Attributes
String m1(int id); m2(); m3() {…}

Class2

Attributes
m1() {…} m2() {…}

Class3

Attributes
m1() {…} m2() {…}

Polymorphism

- The ability of an object to take many forms:

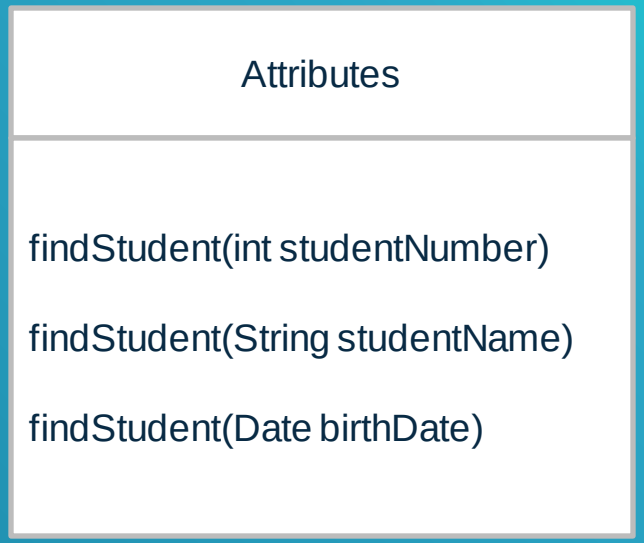
Overloading :

Feature that allows a class to have more than one method having the same name.

Override:

Subclass has the same method as declared in the parent class.

Overloading (Compile Time Polymorphism)



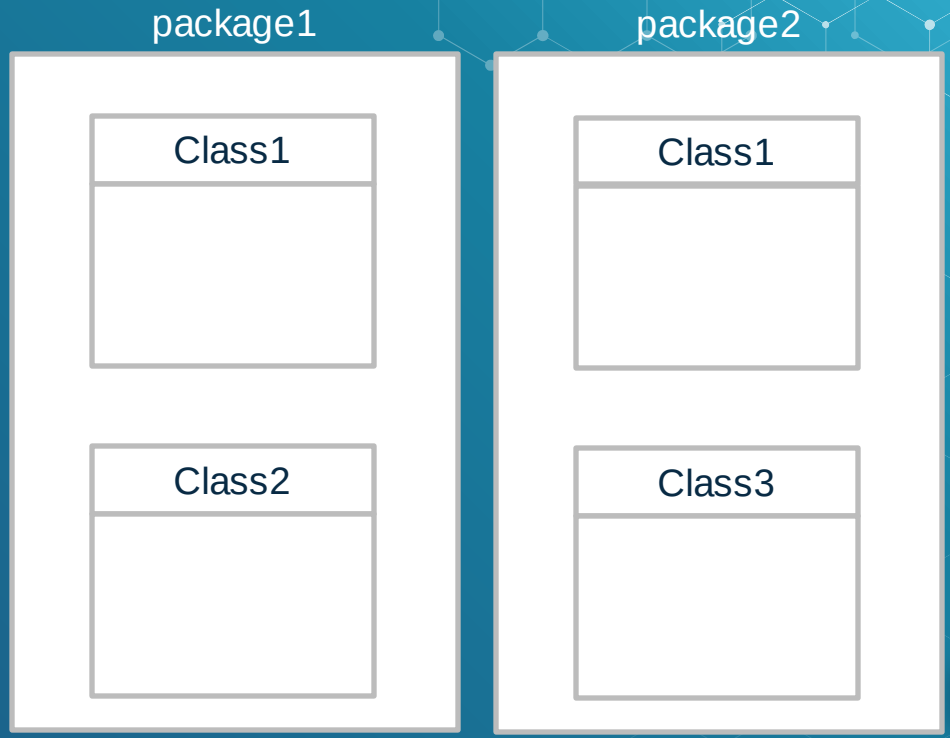
Override (Run Time Polymorphism)

- Method overriding allows a subclass to provide a specific implementation of a method that is already provided by one of its super classes.

- Same name, same parameters or signature, and same return type as the method in the parent class.

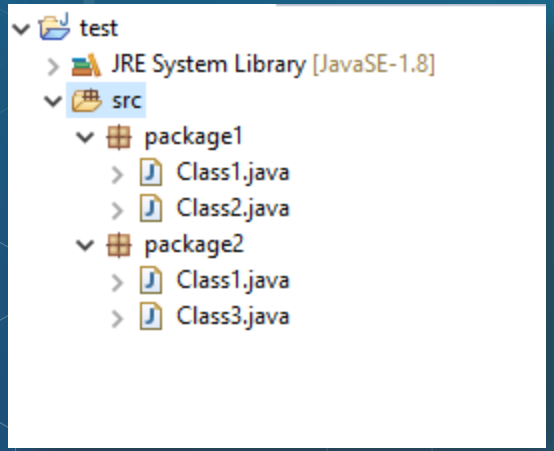
Package

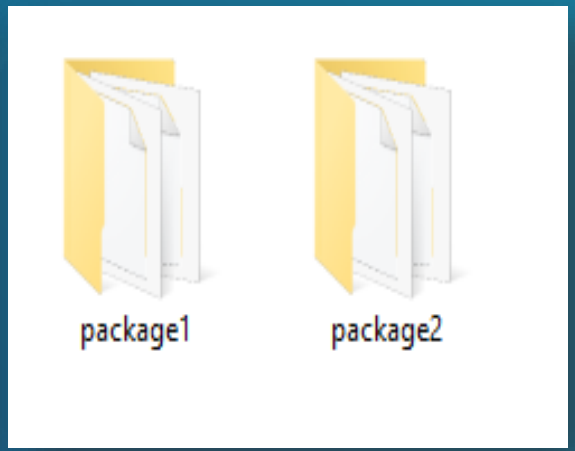
- A group of classes.

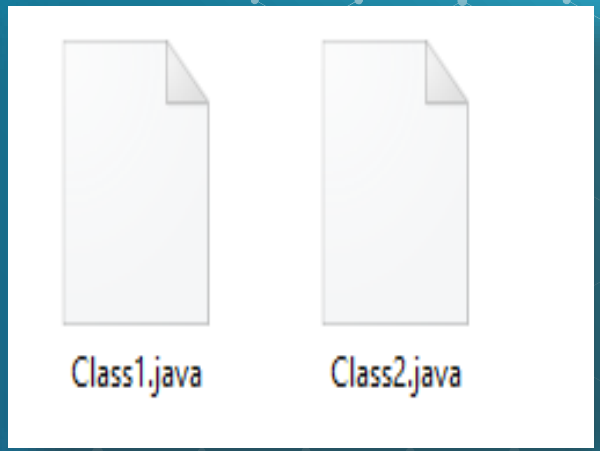


Package

- Folder in a file directory.



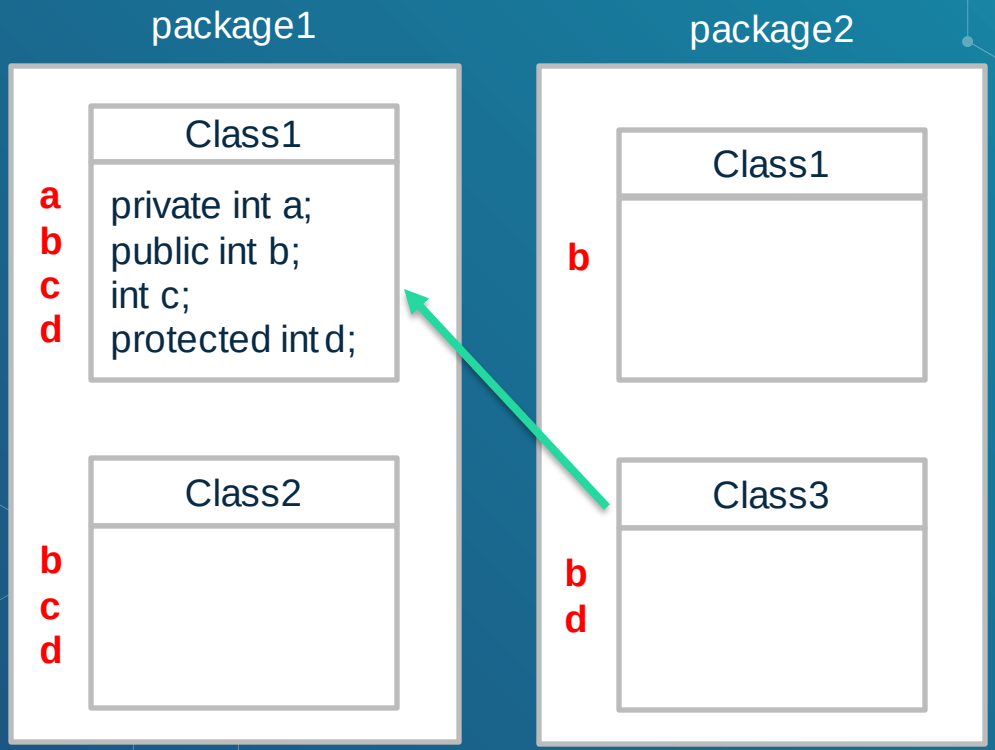




Scope (Access Modifiers)

- **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Public:** The access level of a public modifier is everywhere.
- **Default:** The access level of a default modifier is only within the package.
- **Protected:** The access level of a protected modifier is within the package and outside the package through child class (Default + Subclass).

Scope (Access Modifiers)



Java Naming Convention

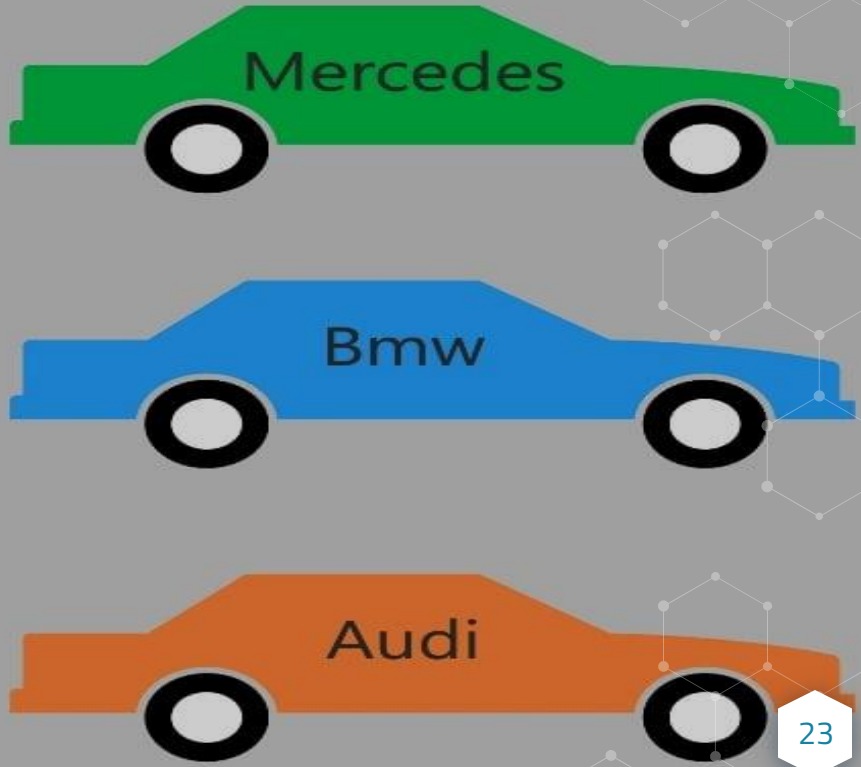
Package	Lowercase letter.	java lang
Class	Noun. Start with the uppercase letter. The initial letter of any subsequent words in the name are capitalized.	Employee FullTimeEmployee
Method	Verb. Start with lowercase letter. The initial letter of any subsequent words in the name are capitalized.	print() addEmployee()
Attribute	Noun. Start with lowercase letter. The initial letter of any subsequent words in the name are capitalized.	name salary studentAverage

Class and Object

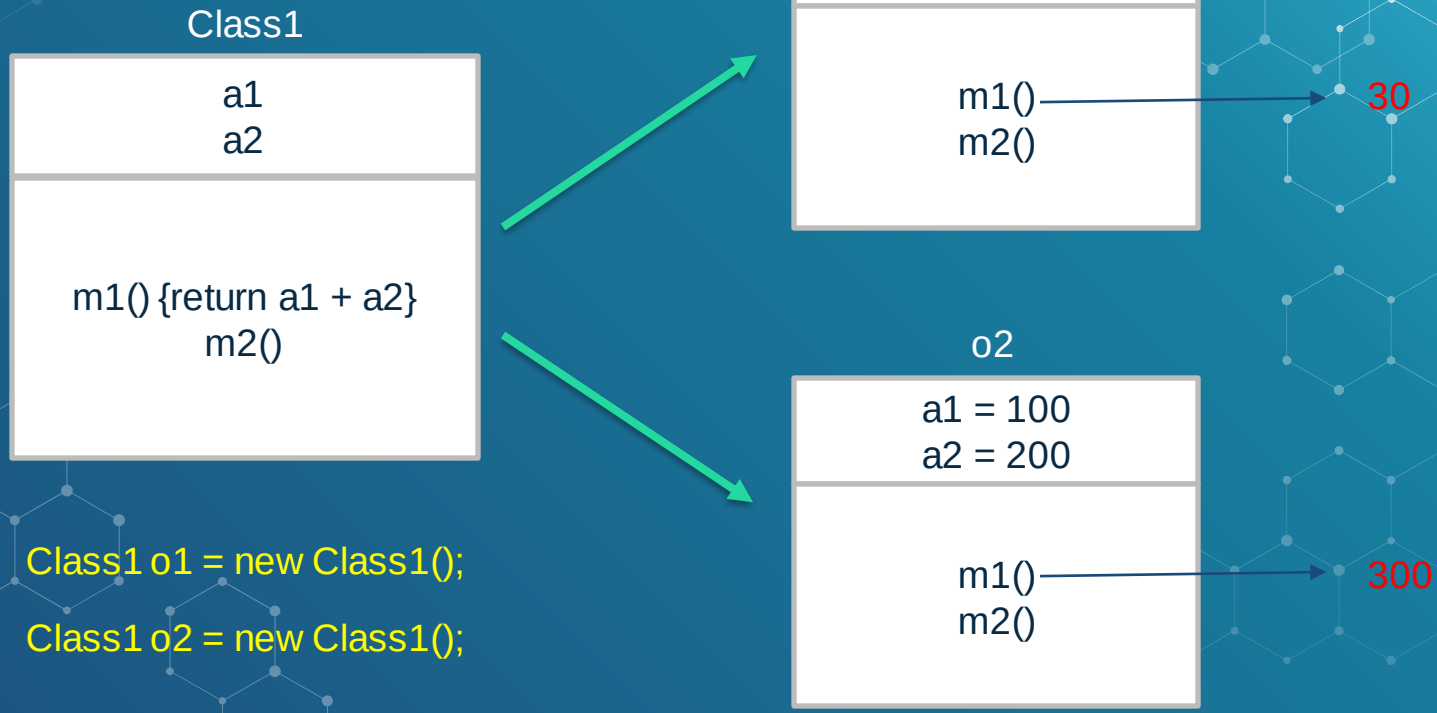
- Class:
  - The basic building block.
  - It represents the set of properties or methods that are common to all objects.
- Object:
  - Runtime instance of a class in memory.
  - Constructed using new keyword.

class

objects



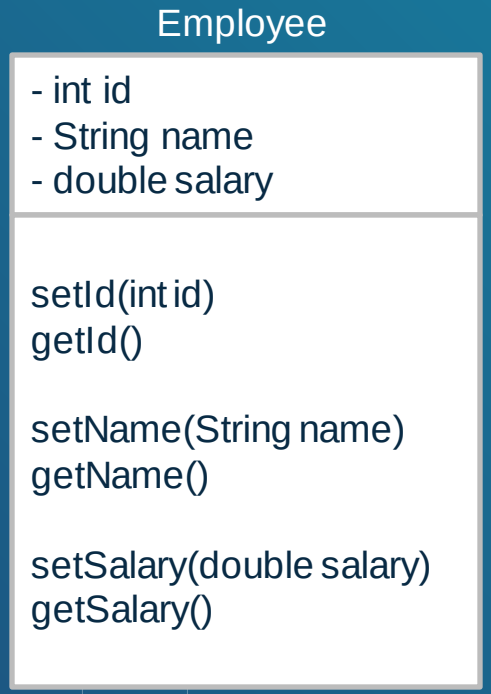
Class and Object



Class1 o1 = new Class1();

Class1 o2 = new Class1();

Class and Object

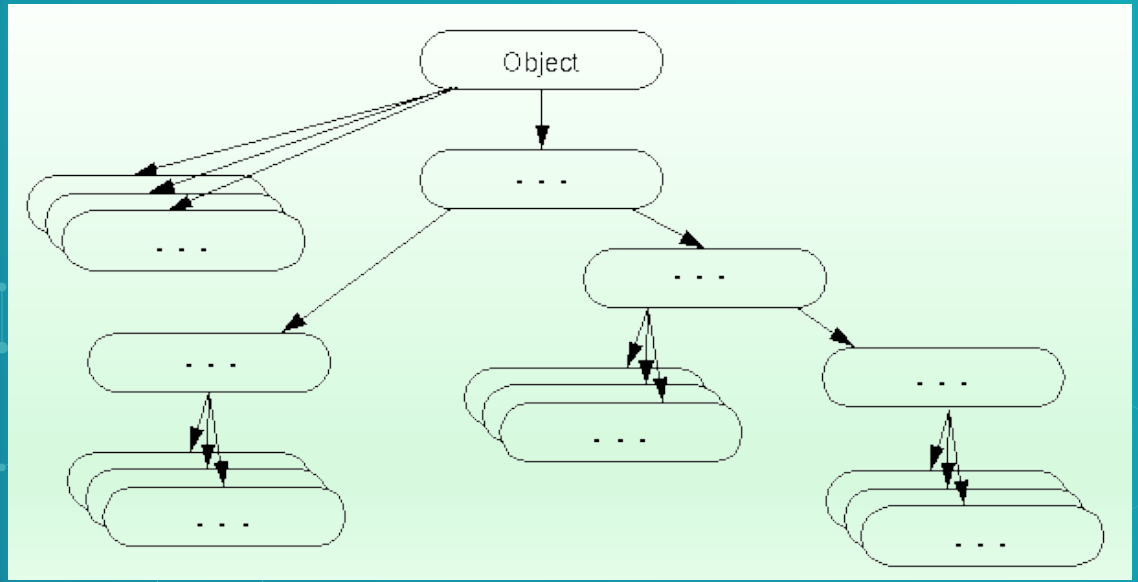


Initialization

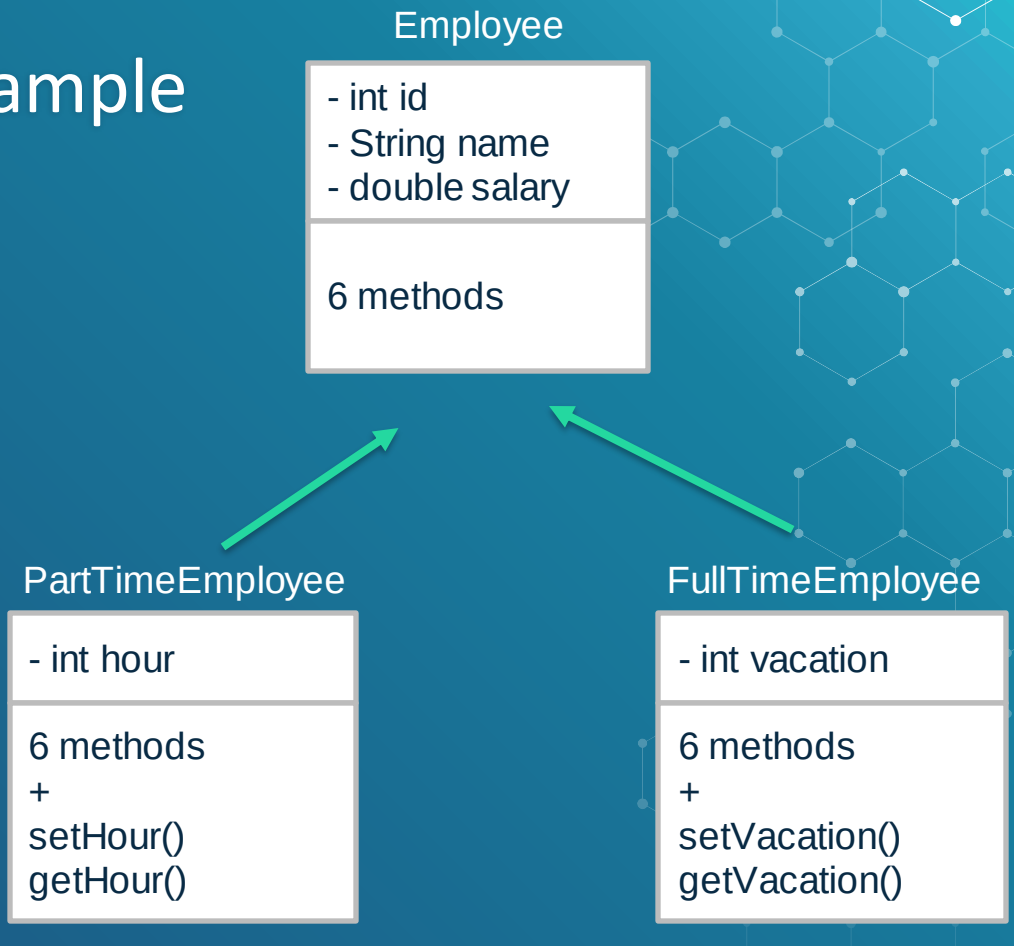
- Automatic
- Simple
- Complex → Constructor:
  - Method
  - Automatic called
  - Class name

Object Class

- The parent class of all the classes in java.
- It is the topmost class of java.

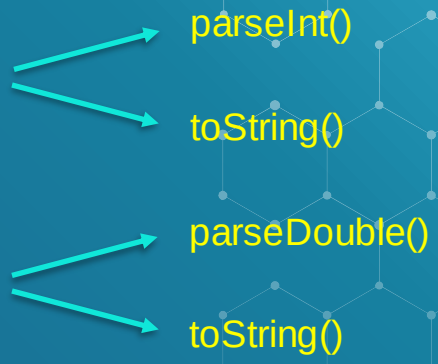


Inheritance Example



Wrappers (Type Conversion)

- Primitive types:
  - byte
  - short
  - int
  - long
  - float
  - double
  - char
  - boolean
- Classes:
  - Byte
  - Short
  - Integer
  - Long
  - Float
  - Double
  - Char
  - Boolean



Example: Calculate student average

Static

- Class attribute / methods.
- Common property of all objects (which is not unique for each object).
- Static method can only access static attributes and static methods.

Static

Class1

int a String b static int c = 15
Methods

Class1.c = 16;

Class1.a = 20; X

o1

a = 10 b = “Ali”
Methods

o1.c = 15;

o2

a = 100 b = “Rahaf”
Methods

print(o2.c); → 15

Containers (Collections) → Array

```
int a[] = new int[5];
System.out.println(a[0]);

String b[] = new String[10];
b[0] = "Ali";
b[9] = "Sami";
System.out.println(b.length);
System.out.println(b[0].length());

Employee e[] = new Employee[100];
e[0].setName("Ali");
for (int i = 0; i < e.length; i++) {
    e[i] = new Employee();
}
e[0].setName("Ali");
```

0

10

3

New for array

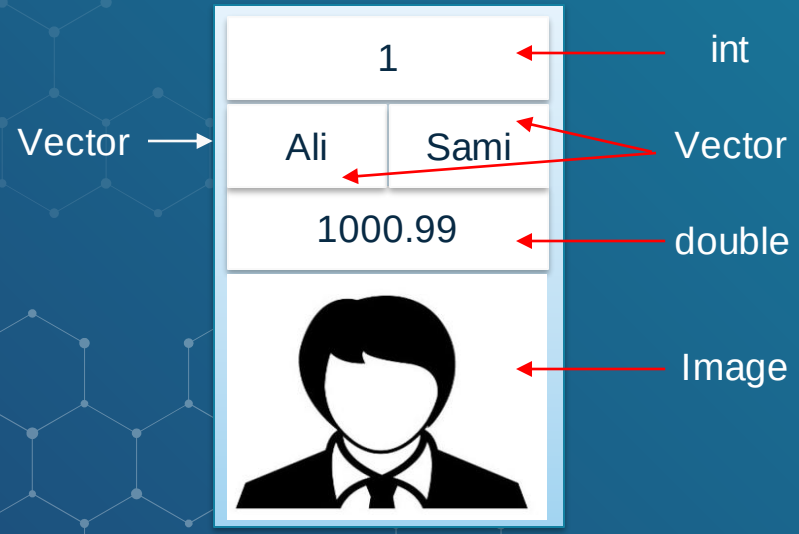
NullPointerException

New for object

Containers (Collections) → Vector

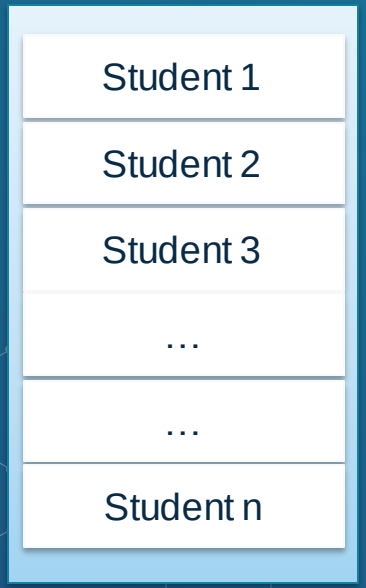
- addElement()
- removeElement()
- insertElementAt()
- removeElementAt()
- Size()

Vector Example



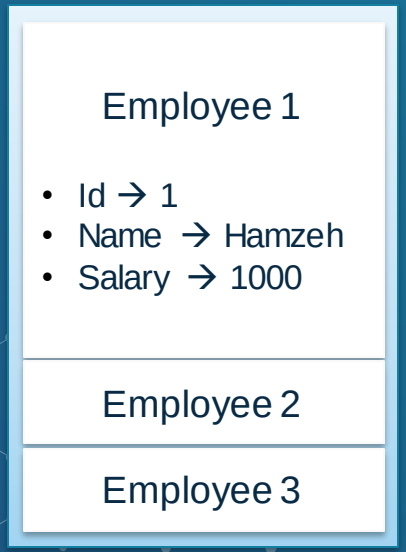

```
Vector vsmall = new Vector();
Vector vlarge = new Vector();

vsmall.add("Ali");
vsmall.add("Sami");

vlarge.add(1);
vlarge.add(vsmall);
vlarge.add(1000.99);


System.out.println(vlarge);
System.out.println("Family Name: " + ((Vector)vlarge.elementAt(1)).elementAt(1));
```

Vector Example




```
Vector<String> vs = new Vector<String>();
vs.add("Student 1");
vs.add("Student 2");
vs.add("Student 3");
```

Vector Example




```
Vector<Employee> ve = new Vector<Employee>();
Employee e1 = new Employee();
Employee e2 = new Employee(4, "Ali", 500);

e1.setId(1);
e1.setName("Hamzeh");
e1.setSalary(1000);

ve.add(e1);
ve.add(e2);
```

Vector VS ArrayList

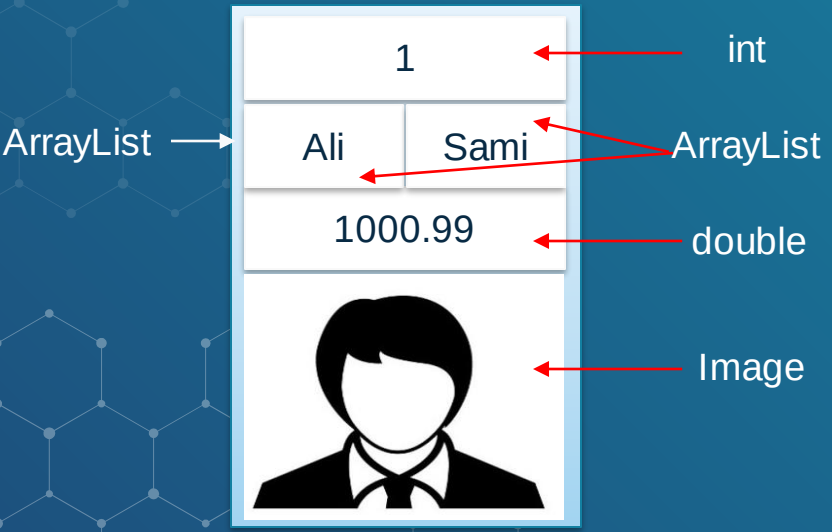
1. **Synchronization:** Vector is synchronized, which means only one thread at a time can access the code, while ArrayList is not synchronized, which means multiple threads can work on ArrayList at the same time.

2. **Performance:** ArrayList is faster, since it is non-synchronized, while vector operations give slower performance since they are synchronized (thread-safe). If one thread works on a vector, it has acquired a lock on it, which forces any other thread wanting to work on it to have to wait until the lock is released.

Vector VS ArrayList

3. **Data Growth:** ArrayList and Vector both grow and shrink dynamically to maintain optimal use of storage – but the way they resize is different. ArrayList increments 50% of the current array size if the number of elements exceeds its capacity, while vector increments 100% – essentially doubling the current array size.

ArrayList Example




```
ArrayList asmall = new ArrayList();
ArrayList alarge = new ArrayList();

asmall.add("Hamzeh");
asmall.add("Asefan");

alarge.add(1);
alarge.add(asmall);
alarge.add(1000);
alarge.add(vlarge);

System.out.println(alarge);
System.out.println(((ArrayList)alarge.get(1)).get(1));
```

HashMap

- It provides the basic implementation of the Map interface of Java.
- It stores the data in (Key, Value) pairs.
- One object is used as a key (index) to another object (value).
- It can store different types: String keys and Integer values, or the same type, like: String keys and String values.

HashMap Example

```
public class TestHashMap {

    public static void main(String[] args) {

        HashMap<String, String> capitalCities = new HashMap<String, String>();

        // Add keys and values (Country, City)
        capitalCities.put("Jordan", "Amman");
        capitalCities.put("Palestine", "Al Quds");
        capitalCities.put("Syria", "Damascus");
        capitalCities.put("Lebanon ", "Beirut");

        System.out.println(capitalCities);

        System.out.println("Access a value in the HashMap --> " + capitalCities.get("Jordan"));

    }

}
```

HashMap Methods

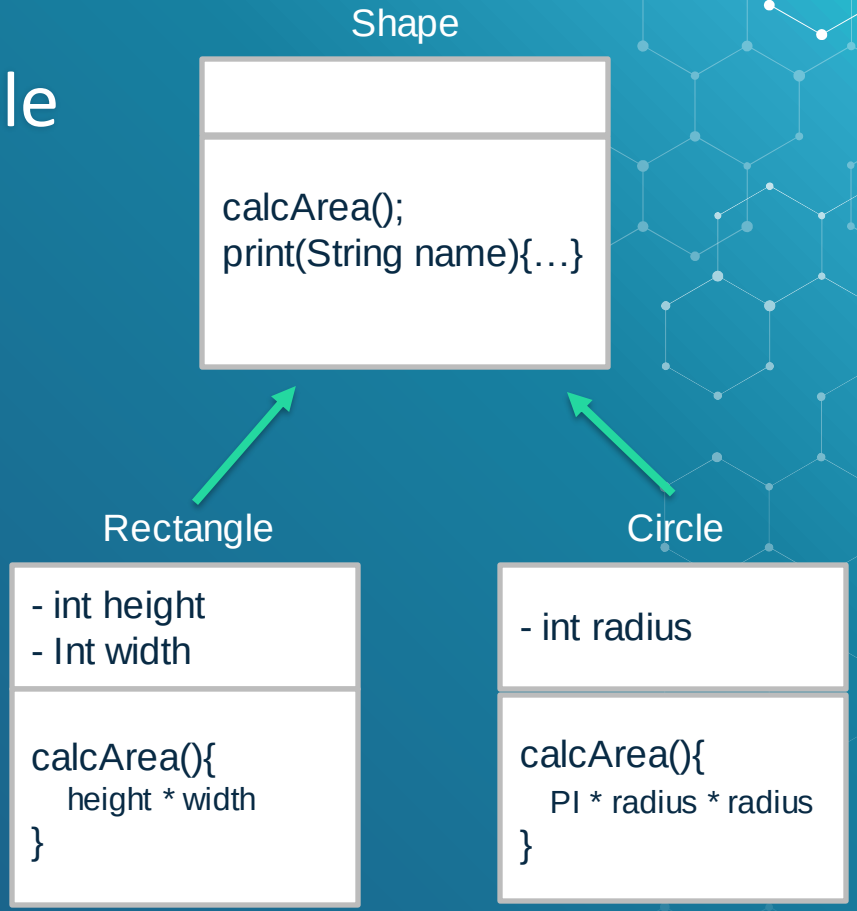
Method	Description
isEmpty()	Returns true if the map is empty; returns false if it contains at least one key.
entrySet()	Used to return a collection view of the mappings contained in this map.
keySet()	Used to return a set view of the keys contained in this map.
values()	Returns a collection view of the values contained in the map.
put(Object key, Object value)	Used to insert an entry in the map.
remove(Object key)	Used to delete an entry for the specified key.

Abstraction

- Abstract keyword is used to create an abstract class in java.
- Abstract class can’t be instantiated.
- Abstract class may or may not have all abstract methods.
- Abstract method is marked with the abstract keyword defined in an abstract class.
- Abstract method must always be redefined in the subclass.

Can you call a method directly from abstract class?

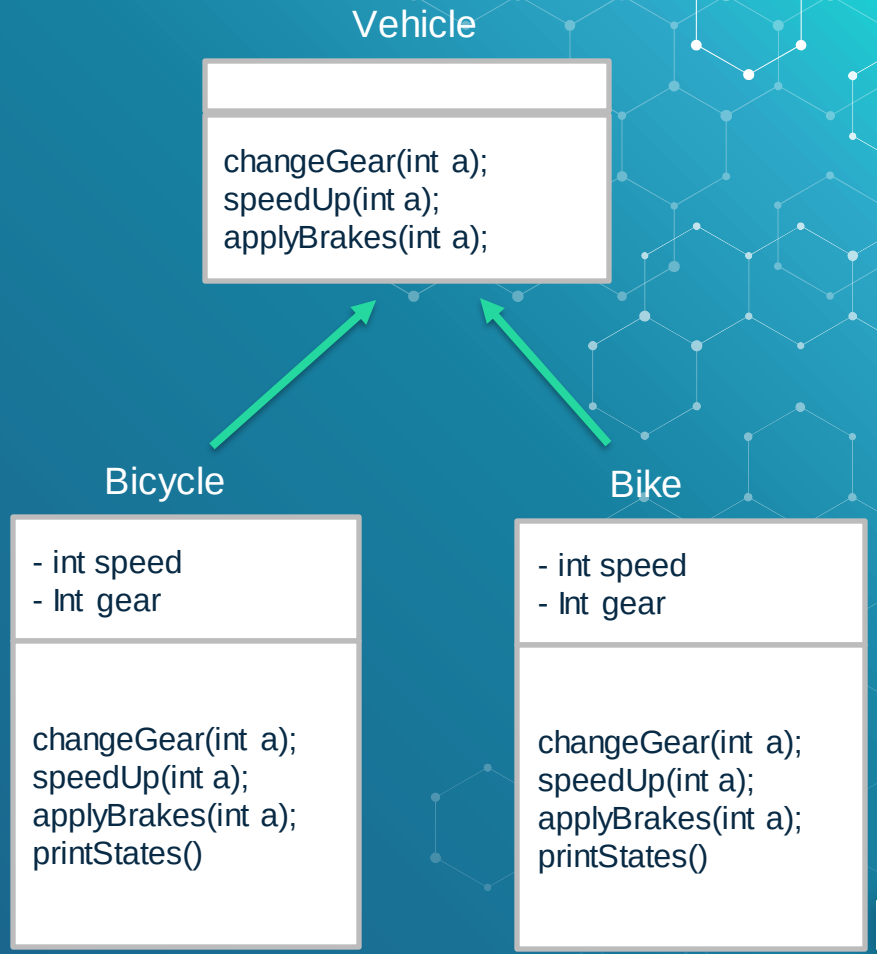
Abstraction Example



Interface

- Full abstract class.
- All the methods in an interface are implicitly abstract.
- The keyword used to deal with interface is **implements**.
- By default all the methods in an interface are public and abstract.
- By default instance variables in an interface are public, static and final.
- Name convention: adjective and start with the uppercase letter:
  - → Printable, Runnable
- Package → A group of classes or interfaces.

Interface Example [1]



Interface Example

```java
public interface Vehicle {

    void changeGear(int a);

    void speedUp(int a);

    void applyBrakes(int a);
}
```

```java
public class Bicycle implements Vehicle {

    int speed;
    int gear;

    @Override
    public void changeGear(int newGear) {
        gear = newGear;
    }

    @Override
    public void speedUp(int increment) {
        speed = speed + increment;
    }

    @Override
    public void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed + " gear: " + gear);
    }

}
```

Interface Example

```java
public class Bike implements Vehicle {

    int speed;
    int gear;

    @Override
    public void changeGear(int newGear) {
        gear = newGear;
    }

    @Override
    public void speedUp(int increment) {
        speed = speed + increment;
    }

    @Override
    public void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed + " gear: " + gear);
    }

}
```

```java
package hmz;

public class Test {

    public static void main(String[] args) {

        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.print("Bicycle present state --> ");
        bicycle.printStates();


        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.print("Bike\s\s\s present state --> ");
        bike.printStates();

    }
}
```

Problems | Console | Progress

```
<terminated> Test (1) [Java Application] D:\jsfCourse\java\jdk1.8.0_202\bin\javaw.exe (
Bicycle present state --> speed: 2 gear: 2
Bike    present state --> speed: 1 gear: 1
```

Final

- Final attribute: It's value can't be modified (constant)

  Name convention: should be all uppercase with words separated by underscores.

  - public final double PI = 3.14
  - static final int MAX_WIDTH = 999;

- Final method: Cannot be overridden
  - public final void m1()
- Final class: Cannot be extended (inherited)
  - public final class Class1

Exception

When executing Java code, different errors can occur:

- Coding errors made by the programmer.
- Errors due to wrong input.
- A file that needs to be opened cannot be found.
- A network connection has been lost.
- The JVM has run out of memory.
- …

Exception Types

1. **Checked Exceptions:**
   - Checked exceptions are checked by the Java compiler so they are called *compile time exceptions*.
   - Exceptions that can occur at compile-time are called checked exceptions since they need to be explicitly checked and handled in code.

2. **Unchecked Exceptions:**
   - Unchecked exceptions are not checked by the compiler. These are called *runtime exceptions*.
   - Unchecked exceptions can be thrown "at any time". Therefore, methods don't have to explicitly catch or throw unchecked exceptions.

Examples

- **Checked Exceptions (Compile):**
  - IOException
  - SqlException
  - …
- **Unchecked Exceptions (Runtime):**
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - ArithmeticException
  - …

Exception Handling

When an Exception occurs the normal flow of the program is disrupted and the terminates abnormally, which is not recommended → therefore, these exceptions are to be handled:

- Throw
- Try and catch:

```
try {
  // Block of code to try
}
catch(Exception e) {
  // Block of code to handle errors
}
```

→ The finally statement lets you execute code, after try...catch, regardless of the result.

Example 1:

```
public static void main(String[] args) {

    int[] a = { 1, 2, 3 };
    System.out.println(a[5]);

}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at section1.Main.main(Main.java:8)
```

Example 1:

```
public static void main(String[] args) {

    try {
        int[] a = { 1, 2, 3 };
        System.out.println(a[5]);
    } catch (Exception e) {
        System.out.println("Something went wrong.");
    }

}
```

Example 2:

- FileWriter:
  - write
  - close
- FileReader:
  - BufferedReader
  - readLine

```java
// Compile Time Exception
try {
    FileWriter fw = new FileWriter("d:\\a.txt");

    for (int i = 1; i <= 5; i++) {
        fw.write("HTU " + i + "\n");
    }
    fw.close();
} catch (IOException e) {
    e.printStackTrace();
}

try {
    FileReader fr = new FileReader("d:\\a.txt");
    BufferedReader br = new BufferedReader(fr);
    String s = br.readLine();
    while (s != null) {
        System.out.println(s);
        s = br.readLine();
    }
    br.close();
    fr.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

Generic

- Allow type (Integer, String, … etc.) to be a parameter to methods, classes, and interfaces.
- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
- We use <> to specify parameter types in generic class creation.

Generic Example

```java
public class GenericClass<myType> {

    private myType value;

    public myType getValue() {
        return value;
    }

    public void setValue(myType value) {
        this.value = value;
    }

}
```

```java
public class TestGenericClass {

    public static void main(String[] args) {

        GenericClass<String> x = new GenericClass<String>();

        // x.setValue(10); Error
        x.setValue("Ali");

    }

}
```

Exercise → Circle

Circle

- int radius

getArea(){
PI * radius * radius
}
 	
getRound(){
2 * PI * radius
}

THANKS!

ANY QUESTIONS?

Textbooks

- Mclaughlin, B.D. et al. (2007). Head First Object-Oriented Analysis and Design. 1st Ed. United States of America: O'Reilly Media.
- Gamma, E. et al. (1995) Design Patterns: Elements of Reusable Object-Oriented Software. 1st Ed. New Jersey: Addison-Wesley.

References

- [1] https://www.geeksforgeeks.org/interfaces-in-java/