Object Oriented Programming

Main concepts

Object Oriented Programming

Piotr Fulmański

Object Oriented Programming

Why?

Put together properties of things and their behavior to avoid design, logic and coding problems.

Better control over the "concurrent" modification of data.

Hello world in Java

Hello world in Java

```
public class HelloWorld {
  public static void main(String[] args){
    System.out.println("Hello world");
  }
}
```

Classes and their objects

Class

A class (may) ***defines***:

- the types of all **internal object *properties*** that hold the **object state**;
- **object behavior** expressed by the code of the ***methods***.

A class determines how an object will behave and what the object will contain. In other words, it is a **blueprint** or a set of instruction to build a specific type of object. It provides initial values for member variables and member functions or methods.

Class

Design

Data members: List down the differences between real life things you want to model. Some of the *differences* you might have listed out are also some *common characteristics* shared by them.

Methods: Next, list out their common behaviors. So these will be the actions of your software objects.

Class

Example

No code, only ideas...

For example: class to represent car.

But what kind of a car? For what you will use it? In what kind of application are you going to us it?

Car parking application?

Service car application?

Insurance application?

All these applications require an existence of a car object, but all of them need different type of information.

If you don't know the type of application, you can't design correct class with correct abstraction of real object.

Object

class

blueprint, template (but not template in OOP sense) spirit

object

representation on one real existing object, body

Object

Dependence: The object is an instance of a class. Because a class is a template for creating objects, all objects of the same class are identical (in some sense).

Entity: A class is a logical entity while object is a physical entity.

Space: A class does not allocate memory space on the other hand object allocates memory space.

How many: You can declare class only once but you can create more than one object using a class.

Usability: Classes can’t be manipulated while objects can be manipulated.

Data: Classes doesn’t have any values, whereas objects have its own values.

Keyword: You can create class using `class` keyword and you can create object using `new` keyword (in Java).

Object

Possible object definitions:

- The object is an *instance of a class*.
- The object is an *entity which has state and behavior*.
- An object is a *real-world entity*.
- An object is a *runtime entity*.

Class and object

Dummy working example

```
// main within the class
class Main {
    public static void main(String[] args) {
        Main m = new Main();
    }
}
```

```
// main outside the class
class Test {

}

class Main {
    public static void main(String[] args) {
        Test t = new Test();
    }
}
```

Object

Initialization

There are three ways to initialize object in Java:

- by reference variable,
- by method,
- by constructor.

Object

Initialization by reference

```
class Point2D {
    int x;
    int y;
}

class Main {
    public static void main(String[] args) {
        Point2D p = new Point2D();
        System.out.println(p.x);
        p.x = 12;
        System.out.println(p.x);
    }
}
```

Output:

```
0
12
```

Object

Initialization by method

```
class Point2D {
    int x;
    int y;

    void setToBeTheSame(int v) {
        x = v;
        y = v;
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Point2D p = new Point2D();
        System.out.println(p.x);
        p.setToBeTheSame(12);
        System.out.println(p.x);
    }
}
```

Output:

```
0
12
```

Object

Again initialization by constructor (this time with explicite constructor)

Rules for the constructor (initializers):

- Constructor name
  - must be the same as its class name (in Java);
  - reserved name, for example `init` (in Swift).
- A Constructor must have no explicit return type.

Object

Initialization by constructor (default constructor)

```
class Point2D {
    int x;
    int y;

    Point2D() {
        x = 7;
        y = 7;
    }
}

class Main {
    public static void main(String[] args) {
        Point2D p = new Point2D();
        System.out.println(p.x);
    }
}
```

Output:

```
7
```

Object

Initialization by constructor (default constructor)

If there is no constructor in a class, compiler automatically creates a default constructor.

```
class Point2D {
    int x;
    int y;
}
```

```
class Main {
    public static void main(String[] args) {
        Point2D p = new Point2D();
        System.out.println(p.x);
    }
}
```

Output:

```
0
```

Object

Again initialization by constructor (this time with parametrized constructor)

```
class Point2D {
    int x;
    int y;

    Point2D(int v1, int v2){
        x = v1;
        y = v2;

    }
}

class Main {
    public static void main(String[] args) {
        Point2D p = new Point2D(12, 17);

        System.out.println(p.x);
    }
}
```

Output:

```
12
```

Object

Initialization by constructor (constructor overloading)

```
class Point2D {
  int x;
  int y;

  Point2D() {
    x = 7;
    y = 7;
  }

  Point2D(int v) {
    x = v;
    y = v;
  }

  Point2D(int v1, int v1) {
    x = v1;
    y = v2;
  }
}

class Main {
  public static void main(String[] args) {
    Point2D p = new Point2D();
    System.out.println(p.x);

    p = new Point2D(17);
    System.out.println(p.x);

    p = new Point2D(3,5);
    System.out.println(p.x);
  }
}
```

Output:

```
7
17
3
```

Object

Difference between constructor and method in Java

Constructor	Method
Used to initialize the state of an object	Used to expose the behavior of an object
Must not have a return type	Must have a return type
Is invoked implicitly	Is invoked explicitly
The Java compiler provides a default constructor if you don't have any	The method is not provided by the compiler in any case
The constructor name must be same as the class name	The method name may or may not be same as the class name

Object

Method signature

The method name and the list of parameter types together are called the ***method signature***. The number of input parameters is called an ***arity***.

Two methods have the same signature if they have:

- the same *name*,
- the same *arity*,
- and the same *sequence of types* in the list of input parameters.

Signature does not include the return type - this is explained in subsequent section: *Polymorphism*, part: *Overloading*.

Object

Method signature

The following two methods have the same signature:

```
double doSomething(String s, int i){
   // Do something
}

int doSomething(String i, int s){
   // Do something
}
```

Object

Method signature

The following two methods have different signature but because of type promotion you will get an error:

```
class OverloadingAmbiguity {
  void test(int a,long b) {
    System.out.println("1");
  }

  void test(long a,int b) {
    System.out.println("2");
  }

  public static void main(String args[]){
    OverloadingAmbiguity obj=new OverloadingAmbiguity();
    obj.test(20,20); // Ambiguity
  }
}
```

Output:

```
Main.java:12: error: reference to test is ambiguous
    obj.test(20,20); // Ambiguity
       ^
  both method test(int,long) in OverloadingAmbiguity and method test(long,int)
in OverloadingAmbiguity match
1 error
```

Type promotion: byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int, long, float or double and so on.

Object

Method signature

It is possible to have method with the name which is the same as the class name.

```
class Test {
    int a = 5;
    Test() {
        a = 7;
    }

    void Test() {
        a = 9;
    }
    void doSomething() {
        System.out.println(a);
    }
}

class Main {
    public static void main(String[] args) {
        Test t = new Test();
        t.doSomething();
        t.Test();
        t.doSomething();
    }
}
```

Output:

```
7
9
```

Inheritance

Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

The idea behind inheritance in Java is that you can **create new classes** that are built **upon existing classes**. When you inherit from an existing class, you can ***reuse*** methods and fields of the parent class. Of course, you can also add new methods and fields in your current class also.

This way, sharing properties and behavior, objects establish a parent-child relationship.

It is possible to make a child behave differently than the inherited behavior would do.

A parent-child relationship is also known as the **IS-A relationship**.

Inheritance

Subclass is a class which inherits the other class. It is also called a *derived class*, *extended class*, or *child class*.

Superclass is the class from where a subclass inherits the features. It is also called a *base class* or a *parent class*.

Inheritance

The syntax of Java Inheritance

```
class SubclassName extends SuperclassName
{
  //methods and fields
}
```

Inheritance

Types of inheritance in Java

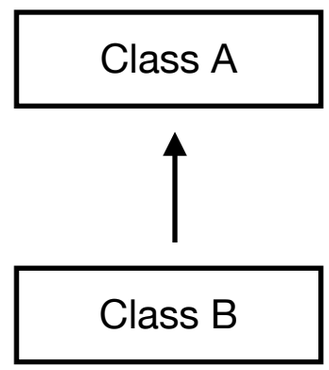
There can be three types of inheritance in java:

- single,
- multilevel
- and hierarchical.

Inheritance

Types of inheritance in Java (single)

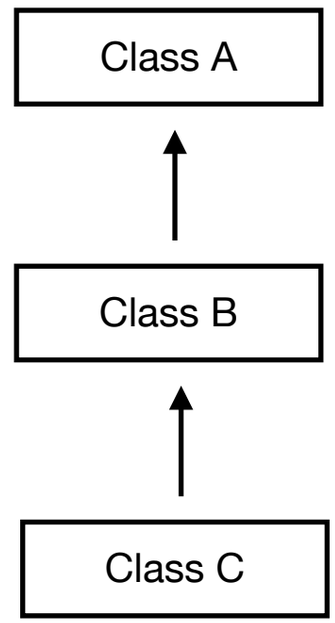
```
class A {
  void doSomething_A() {
    System.out.println("A...");
  }
}
class B extends A {
  void doSomething_B(){
    System.out.println("B...");
  }
}
class Main{
  public static void main(String args[]){
    B b=new B();
    b.doSomething_A();
    b.doSomething_B();
  }
}
```



Inheritance

Types of inheritance in Java (multilevel)

```
class A {
  void doSomething_A() {
    System.out.println("A...");
  }
}

class B extends A {
  void doSomething_B(){
    System.out.println("B...");
  }
}
```

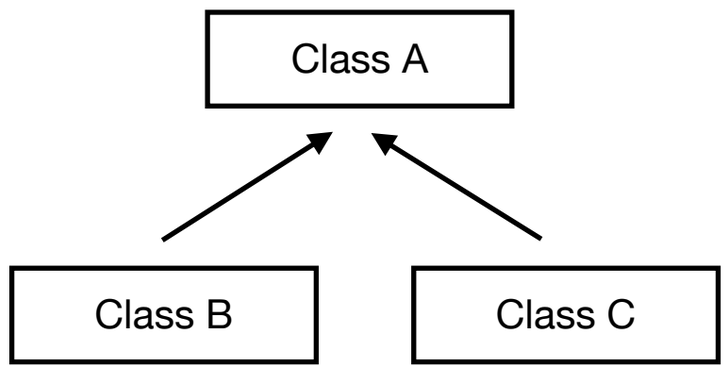



```
class C extends B {
  void doSomething_C(){
    System.out.println("C...");
  }
}

class Main{
  public static void main(String args[]){
    C c=new C();
    c.doSomething_A();
    c.doSomething_B();
    c.doSomething_C();
  }
}
```

Inheritance

Types of inheritance in Java (hierarchical)

```
class A {
  void doSomething_A() {
    System.out.println("A...");
  }
}

class B extends A {
  void doSomething_B(){
    System.out.println("B...");
  }
}
```




```
class C extends A {
  void doSomething_C(){
    System.out.println("C...");
  }
}

class Main{
  public static void main(String args[]){
    C c=new C();
    c.doSomething_A();
    // c.doSomething_B();  // Cause a compilation time error
    c.doSomething_C();
  }
}
```

Inheritance

Why multiple inheritance is a source of problem?

```
class A{
  void msg(){System.out.println("Hello");}
}

class B{
  void msg(){System.out.println("Welcome");}
}
```

```
class C extends A,B{//suppose it is possible
  public static void main(String args[]){
    C obj=new C();
    obj.msg();// Which msg() method should be invoked?
  }
}
```

Inheritance

Aggregation (HAS-A)

Constructor is never inherited. It wouldn't make sense because subclass is different type than superclass – much "larger", with more properties.

Inheritance

Aggregation (HAS-A)

If a class have an **entity reference**, it is known as ***aggregation***. Aggregation represents HAS-A relationship.

Inheritance

Aggregation (HAS-A)

```
class A {
  void doSomething_A() {
    System.out.println("A...");
  }
}

class B {
  A a = new A();

  B() {
    a = new A();
  }

  void doSomething_B(){
    System.out.println("B...");
  }
}

class Main{
  public static void main(String args[]){
    B b=new B();
    // b.doSomething_A();  // Error
    b.a.doSomething_A();
    b.doSomething_B();
  }
}
```

Note: This code will have more sense if you use `this` - see next part for details.

this and static

this

Keyword

`this`

In Java, this is a reference variable that *refers* to the **current object**.

Six usage of java this keyword:

- to refer *current* class instance variable;
- to invoke current class method (implicitly);
- to invoke current class constructor;
- passed as an argument in the method call;
- passed as argument in the constructor call;
- to return the current class instance from the method.

See for `super` keyword in subsequent part to refer to **superclass** variable, methods constructors etc.

Keyword

`this` - refer *current* class instance variable

```
class Point2D {
  int x;
  int y;

  Point2D() {
    x = 7;
    y = 7;
  }

  Point2D(int v) {
    x = v;
    y = v;
  }

  Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }
}

class Main {
  public static void main(String[] args) {
    Point2D p = new Point2D();
    System.out.println(p.x);

    p = new Point2D(17);
    System.out.println(p.x);

    p = new Point2D(3,5);
    System.out.println(p.x);
  }
}
```

Keyword

`this` - invoke current class method

```
class A {
  void callMe() {
    System.out.println("***");
  }

  void doSomething_A() {
    this.callMe();
    System.out.println("A...");
    callMe(); // Also possible; same as this.callMe()
  }
}

class Main{
  public static void main(String args[]){
    A a=new A();
    a.doSomething_A();
  }
}
```

Keyword

`this` - invoke current class constructor

Calling default constructor from parameterized constructor:

```
class A {
  A() {
    System.out.println("!!!");
  }

  A(String msg) {
    this();
    System.out.println(msg);
    // error: call to this must be first
    // statement in constructor
    //this();
  }
}

class Main{
  public static void main(String args[]){
    A a=new A("message");
  }
}
```

Keyword

`this` - invoke current class constructor

Calling parameterized constructor from default constructor:

```
class A {
  A() {
    this("message");
    System.out.println("!!!");
  }
  A(String msg) {
    System.out.println(msg);
  }
}

class Main{
  public static void main(String args[]){
    A a=new A();
  }
}
```

Keyword

`this` - passed as an argument in the method call

```
class A {
  int x;

  void method(A obj) {
    obj.x = 5;
  }

  void doSomething() {
    method(this);
    System.out.println(x);
  }
}

class Main{
  public static void main(String args[]){
    A a=new A();
    a.doSomething();
  }
}
```

Keyword

`this` - passed as argument in the constructor call

Useful if we have to use one object in multiple classes:

Pass Counter to A and maybe many other classes and operate on (one common) Counter.

```
class A {
  int x;

  A(Counter c) {
    c.inc();
  }
}
```

```
class Counter {
  int x;

  void inc() {
    x += 1;
  }
}

class Main{
  public static void main(String args[]){
    Counter c = new Counter();
    A a1=new A(c);
    System.out.println(c.x);
    A a2=new A(c);
    System.out.println(c.x);
  }
}
```

Keyword

`this` - passed as argument in the constructor call: aggregation (HAS-A)

Pass B to A to allow A to operate on B

```
class A {
  int a;
  B b;

  A(B b) {
    this.b = b;
  }

  void doSomething_A() {
    System.out.println("from A");
    b.doSomething_B();
  }
}

class B {
  int b;
  A a = new A(this);

  void doSomething_B() {
    System.out.println("from B");
  }
}

public class Main {
  public static void main(String[] args) {
    B b = new B();
    b.a.doSomething_A();
    }
}
```

Keyword

`this` - return the current class instance from the method

```
Example - have no idea ;)
```

static

Instance and static properties and methods

`static`

Instance methods are methods that can be invoked only on an object (instance) of a class.

If method you implement does not use the object state, it can be made ***static*** and invoked **without creating an object**.

Similarly, a property can be declared static and thus accessible without creating an object.

Instance and static properties and methods

static

```
class A {
   static int x;

   A() {
      x += 1;
   }
}
```

```
class Main{
   public static void main(String args[]){
      A a1=new A();
      System.out.println(a1.x);
      A a2=new A();
      System.out.println(a1.x);
   }
}
```

Instance and static properties and methods

static

```
class Calculate{
  static int cube(int x){
    return x*x*x;
  }

  public static void main(String args[]){
    int result=Calculate.cube(5);
    System.out.println(result);
    }
}
```

Instance and static properties and methods

`static` - limitations

- The static method can not use non static data member or call non-static method directly.

- `this` and `super` cannot be used in static context.

Abstraction

Abstrac class

Abstraction

Idea – communicate only through well defined interfaces

As you know, the name of a method along with the sequence of its parameter types is called a method signature. Together with a return type it informs you how to call this method and what you will get. If the name is reasonable, you may infer the purpose of this method, but in general you know nothing about the code that does calculations.

All the **implementation details are hidden** (***encapsulated***) within the class.

This is how you create abstraction, providing only an interface to communicate with class and keep closed all the internal details which are not relevant.

A process of hiding the implementation details and showing only functionality to the user is called an **abstraction**.

As we have mentioned already, a class can implement many different interfaces. But two different classes (and their objects) can behave differently even when they implement the same interface.

Abstraction

Idea – communicate only through well defined interfaces

There are two ways to achieve abstraction in Java

- abstract class,
- interface.

Abstraction

Abstract class

- An abstract class must be declared with an `abstract` keyword.
- It can have abstract as well non-abstract methods.
- It **cannot be instantiated**.
- It can have constructors and static methods.
- It can have final methods which will force the subclass not to change the body of the method.

A method which is declared as *abstract* and does not have implementation is known as an abstract method.

Abstraction

Abstract class – example 1

The following code will cause errors:

```
class Test {
    int a = 5;

    Test() {
        a = 7;
    }

    abstract void Test() {
        a = 9;
    }

    void doSomething() {
        System.out.println(a);
    }
}

class Main {
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println(t.a);
        t.doSomething();
    }
}
```

Output:

```
Compilation failed due to following error(s). Main.java:4: error: Test is not abstract and does not
override abstract method Test() in Test
class Test {
^
Main.java:11: error: abstract methods cannot have a body
    abstract void Test() {
                  ^
2 errors
```

Abstraction

Abstract class – example 2

The following code will cause errors:

```
abstract class Test {
    int a = 5;

    Test() {
        a = 7;
    }

    abstract void Test() {
        a = 9;
    }

    void doSomething() {
        System.out.println(a);
    }
}

class Main {
    public static void main(String[] args) {
        Test t = new Test(12, 17);
        System.out.println(t.a);
        t.doSomething();
    }
}
```

Output:

```
Main.java:8: error: abstract methods cannot have a body
    abstract void Test() {
                  ^
Main.java:19: error: Test is abstract; cannot be instantiated
        Test t = new Test(12, 17);
```

Abstraction

Abstract class – example 3

The following code will cause errors:

```
abstract class Test {
    int a = 5;

    Test() {
        a = 7;
    }

    abstract void Test();

    void doSomething() {
        System.out.println(a);
    }
}

class Main {
    public static void main(String[] args) {
        Test t = new Test(12, 17);
        System.out.println(t.a);
        t.doSomething();
    }
}
```

Output:

```
Main.java:17: error: Test is abstract; cannot be instantiated
        Test t = new Test(12, 17);
                 ^
1 error
```

Abstraction

Abstract class – example 4

The following code is correct:

```
abstract class Test {
    int a = 5;

    Test() {
        a = 7;
    }

    abstract void Test();

    void doSomething() {
        System.out.println(a);
    }
}

class TestAbstraction extends Test {
    void Test() {
        a = 9;
    }
}

class Main {
    public static void main(String[] args) {
        TestAbstraction t = new TestAbstraction();
        System.out.println(t.a);
        t.doSomething();
        t.Test();
        System.out.println(t.a);
    }
}
```

Output:

```
7
7
9
```

Interface

Abstraction

Interface

However strange that might sound, even though classes are "templates", they can also be created with "templates" themselves. Such a "template" or blueprint of class is an ***interface***.

In the Java interfaces can have:

- abstract methods
- and variables.

It cannot have a method body.

Interface fields are public, static and final by default, and the methods are public and abstract.

Abstraction

Interface – why you need them?

There are mainly three reasons to use interface:

- to achieve abstraction;
- to "get impression" of multiple inheritance;
- to achieve loose coupling or to get *horizontal relationships* (in contrast to *vertical relationships* achieve with classes).

A class ***extends*** another class.

An interface ***extends*** another interface.

A class ***implements*** an interface.

Abstraction

Interface – example 1

```
interface Printable {
    void print();
}

class Test implements Printable{
    int a = 5;

    Test() {
        a = 7;
    }

    public void print() {
        System.out.println(a);
    }
}

class Main {
    public static void main(String[] args) {
        Test t = new Test();
        t.print();
    }
}
```

Explanation: The default scope for method in class is *package-private*. All classes in the same package can access the method/ field/class. Package-private is *stricter* than protected and public scopes, but more *permissive* than private scope. Members of an interface are always *publicly* accessible.

Output:

```
7
```

In `Test` class a `public` is necessary in front of `print()` to avoid this error:

```
Main.java:12: error: print() in Test cannot implement print() in Printable
    void print() {
         ^
  attempting to assign weaker access privileges; was public
1 error
```

Abstraction

Interface – example 2

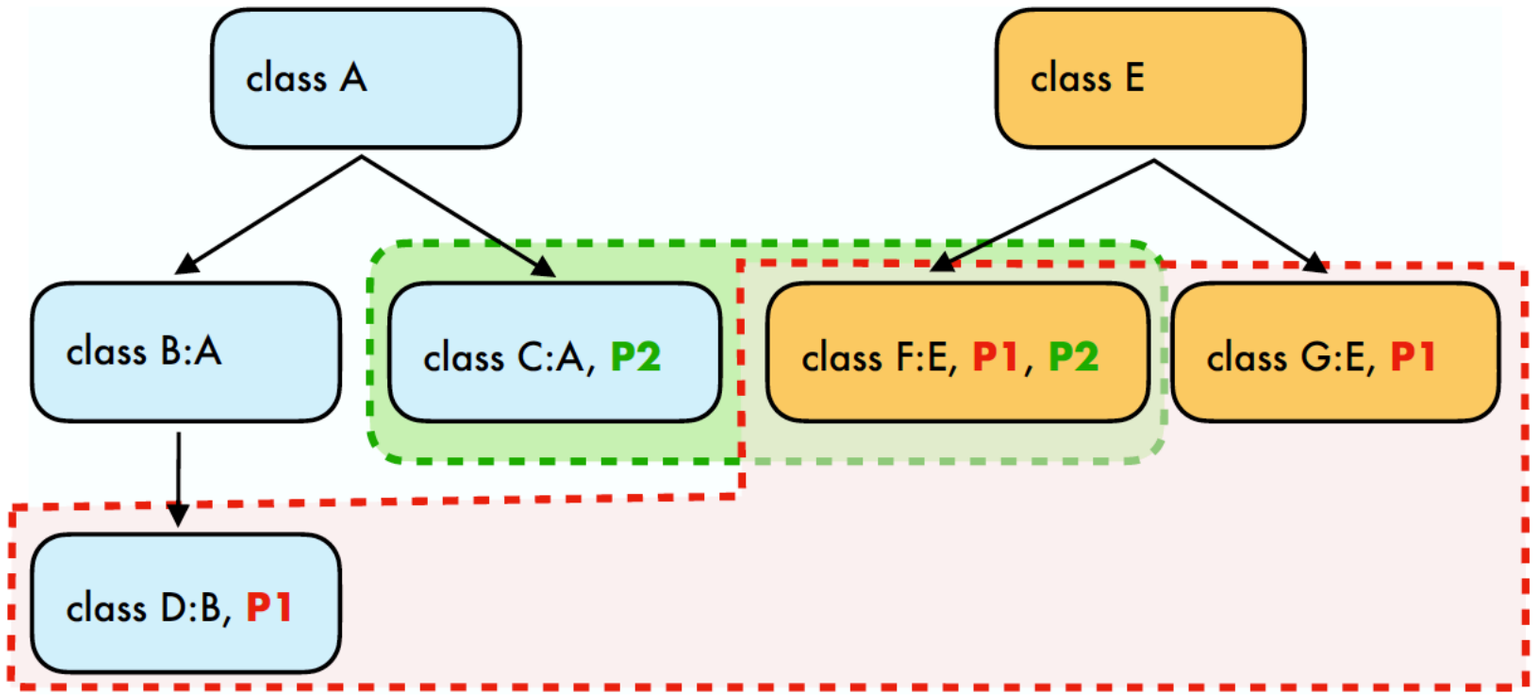
```
interface A {
    void a();
    void b();
}

abstract class B implements A{
    public void b(){System.out.println("B:b");}
}

class C extends B{
    public void a(){System.out.println("C:a");}
}

class Main {
    public static void main(String[] args) {
        C c=new C();
        c.a();
        c.b();
    }
}
```

Output:

```
C:a
B:b
```

Abstraction

Interface – example 3: horizontal relationship



Abstraction

Interface – example 3: horizontal relationship

```
interface A {
    void a();
    void b();
}

abstract class B implements A{
    public void b(){System.out.println("B:b");}
}

class C extends B{
    public void a(){System.out.println("C:a");}
}

class Main {
    public static void main(String[] args) {
        C c=new C();
        c.a();
        c.b();
    }
}
```

Output:

```
C:a
B:b
```

Abstract class vs. interface

Abstract class vs. interface

Abstract class	Interface
Doesn't support multiple inheritance	Supports multiple inheritance
Can have final, non-final, static and non-static variables	Has only static and final variables
Can provide the implementation of interface.	Can't provide the implementation of abstract class.
Can extend another Java class and implement multiple Java interfaces.	Can extend another Java interface only
Can have class members like private, protected, etc	Members of a Java interface are public by default

Encapsulation

Encapsulation

Encapsulation refers to the

- bundling of data with the methods that operate on that data,
- or the restricting of direct access to some of an object's components.

Encapsulation is **used to hide** the values or state of a structured data object inside a class, **preventing direct access** to them by clients in a way that could expose hidden implementation details or violate state invariance maintained by the methods.

Encapsulation addresses the main issue that motivated the creation of object-oriented programming – **better management of concurrent access to shared data**.

Package

Encapsulation via package

- Java package is used to **categorize** the classes and interfaces so that they can be easily maintained.
- Java package **provides access protection**.
- Java package **removes naming collision**.

Access modifiers

Encapsulation via access modifiers

In Java there are four types of access modifiers:

- **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

- **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

- **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

- **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access modifiers

Encapsulation via access modifiers

In Swift there are four types of access modifiers:

- **Open access** (`open` keyword) and more restrictive **public access** (`public`) enable entities to be used within any source file from their defining module, as well as in a source file from another module that imports the defining module.

- **Internal access** (`internal`) enables entities to be used within any source file from their defining module, but not in any source file outside of that module. This is default access specifier in Swift.

- **File-private** access (`fileprivate`) restricts the use of an entity to its own defining source file.

- **Private** access (`private`) restricts the use of an entity to the enclosing declaration.

Polymorphism

Polymorphism

Polymorphism is derived from two Greek words: *poly* and *morphs*. The word *poly* means ***many*** and *morphs* means ***forms***. So polymorphism means many forms.

Polymorphism is the ability of an object to behave as an object of a different class or as an implementation of a different interface.

Polymorphism is a concept by which you can perform a single action in different ways.

There are two types of polymorphism:

- *compile-time* polymorphism (for example, if you overload a static method, it is the example of compile time polymorphism) [related things: static binding/compile-time binding/early binding/ method overloading (in same class)],

- and *runtime* polymorphism
  [related things: dynamic binding/run-time binding/late binding/ method overriding (in different classes)].

Overloading

Overloading

If a class has multiple methods having same name but different in parameters, it is known as **method overloading**.

There are two ways to overload the method in Java:

- by changing number of arguments;
- by changing the data type.

In Java, method overloading is not possible by changing the return type of the method only.

Overloading

Method overloading is not possible by changing the return type of method only.

```
class A {
  String doSomething(int x, int y) {
    String s = "";
    for (int i=0; i<y; i++) {
      s += Integer.toString(x);
    }
    return s;
  }

  int doSomething(int x, int y) {
    return x * y;
  }
}

class Main{
  public static void main(String args[]){
    A a=new A();
    // How to determine which 'doSomething' method should be called?
    System.out.println(a.doSomething(3,5));
  }
}
```

Output:

```
Main.java:10: error: method doSomething(int,int) is already defined in class A

  int doSomething(int x, int y) {

      ^

1 error
```

Overloading

Java always calls `main` method with `String[] args` argument:

```
class Main {
  public static void main(String[] args) {
    System.out.println("main 1");
  }

  public static void main(String args) {
    System.out.println("main 2");
  }

  public static void main() {
    System.out.println("main 3");
  }
}
```

Output:

```
main 1
```

Note when name of the class is different than `Main`:

```
Error: Could not find or load main class Main
Caused by: java.lang.ClassNotFoundException: Main
```

Note when there is no `public static void main(String[] args)` method:

```
Error: Main method not found in class Main, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Overriding

Overriding

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding**.

Overriding

Method overloading	Method overriding
Is used to increase the readability of the program	Is used to provide the specific implementation of the method that is already
Performed within class	Occurs in two classes that have IS-A (inheritance) relationship
Parameter must be different	Parameter must be same
Example of compile time polymorphism	Example of run time polymorphism
Can't be performed by changing return type of the method only (return type can be same or different, but you must have to change the parameters)	Return type must be same

super

`super`

The super keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

It can be used to:

- refer immediate parent class instance variable;
- invoke immediate parent class method;
- invoke immediate parent class constructor.

See for `this` keyword in preceding part to refer to **current object** variable, methods constructors etc.

super

```
class A {
  int x = 5;
  void doSomething() {
    System.out.println("A");
  }
}

class B extends A {
  int x = 7;
  void doSomething() {
    System.out.println("B");
  }

  void execute() {
    System.out.println(x);
    System.out.println(super.x);
    doSomething();
    super.doSomething();
  }
}

class Main {
  public static void main(String[] args) {
    B b = new B();
    b.execute();
  }
}
```

Output:

```
7
5
B
A
```

super

The super keyword can also be used to invoke the parent class constructor. super() is added in each class constructor automatically by compiler if there is no super() or this().

```
class A {
  A() {
    System.out.println("A");
  }
}

class B extends A {
  B() {
    super(); // Explicite invoke of the parent class constructor
    System.out.println("B");
  }
}

class Main {
  public static void main(String[] args) {
    B b = new B();
  }
}
```

Output:

```
A
B
```

final

`final`

The final keyword in java is used to restrict the user. Final can be:

- variable,
- method,
- class.

If you make any variable as final, you cannot change the value.

If you make any method as final, you cannot override it.

If you make any class as final, you cannot extend it.

Final method is inherited but you cannot override it.

A final variable that is not initialized at the time of declaration is known as *blank final variable*. `final` is useful to create a variable that is initialized at the time of creating object and once initialized may not be changed.

Runtime polymorphism

Runtime polymorphism

Runtime polymorphism or *dynamic method dispatch* is a process in which a **call to an overridden method is resolved at runtime** rather than compile-time.

In this process, an **overridden method is called through the reference variable of a *superclass***. The determination of the method to be called is based on the object being referred to by the reference variable.

Dynamic Polymorphism in OOPs is the mechanism by which multiple methods can be defined with same name and signature in the superclass and subclass. The call to an overridden method are resolved at run time.

Note:
A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

Runtime polymorphism

```
class A {
  int x = 5;

  void doSomething() {
    System.out.println("A");
  }
}

class B extends A {
  int x = 7;

  void doSomething() {
    System.out.println("B");
  }
}

class Main {
  public static void main(String[] args) {
    A a = new B();
    System.out.println(a.x);
    System.out.println(((B)a).x);
    a.doSomething();
  }
}
```

Output:

```
5
7
B
```

Runtime polymorphism with multilevel inheritance

```
class A {
  int x = 5;
  void doSomething() {
    System.out.println("A");
  }
}

class B extends A {
  int x = 7;
  void doSomething() {
    System.out.println("B");
    }
}

class C extends B {
  int x = 9;
  void doSomething() {
    System.out.println("C");
    }
}

class Main {
  public static void main(String[] args) {
    A a = new A();
    A b = new B();
    A c = new C();
    System.out.println(a.x);
    System.out.println(b.x);
    System.out.println(c.x);
    a.doSomething();
    b.doSomething();
    c.doSomething();
    System.out.println(((B)b).x);
    System.out.println(((C)c).x);
  }
}
```

Output:

```
5
5
5
A
B
C
7
9
```

Runtime polymorphism

```
class A {
  int x = 5;
  void doSomething() {
    System.out.println("A");
  }
}

class B extends A {
  int x = 7;
  void doSomething() {
    System.out.println("B");
  }
}

class C extends B {
  int x = 9;
  void doSomething() {
    System.out.println("C");
  }
}

class Main {
  public static void main(String[] args) {
    A[] a = {new A(), new B(), new C(), new B(), new A()};
    for (A obj: a) {
      obj.doSomething();
    }
  }
}
```

Output:

```
A
B
C
B
A
```

Static polymorphism

Static polymorphism

Static polymorphism is a type of polymorphism that collects the information for calling a method at compilation time, which is in contrast to dynamic polymorphism which collects the information for calling a method at runtime.

Static polymorphism is realized through method overloading.

Static polymorphism

```
class A {
  void doSomethingWith(int i){
    System.out.println("int: "+i);
  }

  void doSomethingWith(double d){
    System.out.println("double: "+d);
  }

  void doSomethingWith(String s){
    System.out.println("String: "+s);
  }
}

public class Main {
  public static void main(String[] args) {
    A a = new A();
    a.doSomethingWith(3);
    a.doSomethingWith(3.0);
    a.doSomethingWith("test");
  }
}
```

Output:

```
int: 3
double: 3.0
String: test
```

Binding

Binding

Polymorphism is realized by ***binding***.

Connecting a method call to the method body is known as *binding*.

There are two types of binding:

- *static binding* (also known as *early binding*),
- *dynamic binding* (also known as *late binding*).

When type of the object is determined at compiled time (by the compiler), it is known as *static binding*.

When type of the object is determined at run-time, it is known as *dynamic binding*.

Binding

```
class A {
  int x = 5;
  void doSomething() {
    System.out.println("A");
  }
}

class B extends A {
  int x = 7;
  void doSomething() {
    System.out.println("B");
  }
}

class Main {
  public static void main(String[] args) {
    B b = new B(); // static binding
    A a = new B(); // dynamic binding
    b.doSomething(); // prints B
    a.doSomething(); // prints B
  }
}
```

Dynamic binding:
In the above example object type cannot be determined by the compiler, because the instance of `B` is also an instance of `A`. So compiler doesn't know its type, only its base type.

Binding

This would be much more clear in the following example:

```
import java.util.Random;

class A {
  int x = 5;
  void doSomething() {
    System.out.println("A");
  }
}

class B extends A {
  int x = 7;
  void doSomething() {
    System.out.println("B");
  }
}

class Main {
  public static void main(String[] args) {
    Random random = new Random(System.currentTimeMillis());
    A[] a = {new A(), new B(), new A(), new B(), new A(), new B()};
    int min = 0, max = 5;
    int index = random.nextInt(max - min) + min;
    A aa = a[index];
    aa.doSomething();
  }
}
```

Every run may return either `A` or `B`.

Static polymorphism through templtes or generics

Templates/generics

```
class Test<T> {
  T obj;
  Test(T obj) {
    this.obj = obj;
  }

  public T getObject() {
    return this.obj;
  }
}
```

```
class Main {
  public static void main(String[] args) {
    Test<Integer> iObj = new Test<Integer>(15);
    System.out.println(iObj.getObject());

    Test<String> sObj = new Test<String>("Test");
    System.out.println(sObj.getObject());
  }
}
```

Output:

```
15
Test
```