# Java语言概述

## Java环境准备

环境准备只需三步：
第一步：jdk 1.8
第二步：idea
第三步：检查环境是否配置正确

### JDK 1.8

下载路径：JDK 1.8

历史版本下载 JDK历史版本下载

安装完成后，配置环境变量

- 打开系统环境变量，添加几个变量
  - JAVA_HOME=C:\Program Files\Java\jdk1.8
  - CLASSPATH=.;%JAVA_HOME%\lib
- 修改PATH变量的值
  - PATH=%JAVA_HOME%\bin

  检查：cmd启动控制台后，运行java -version，java version "1.8.."有上述字样，说明java安装和配置成功

Linux中修改 `~/.bashrc` 中的环境变量即可

```
> java -version
java version "1.8.0_301"
Java(TM) SE Runtime Environment (build 1.8.0_301-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.301-b09, mixed mode)
```

### 安装Maven

安装文档：https://maven.apache.org/install.html

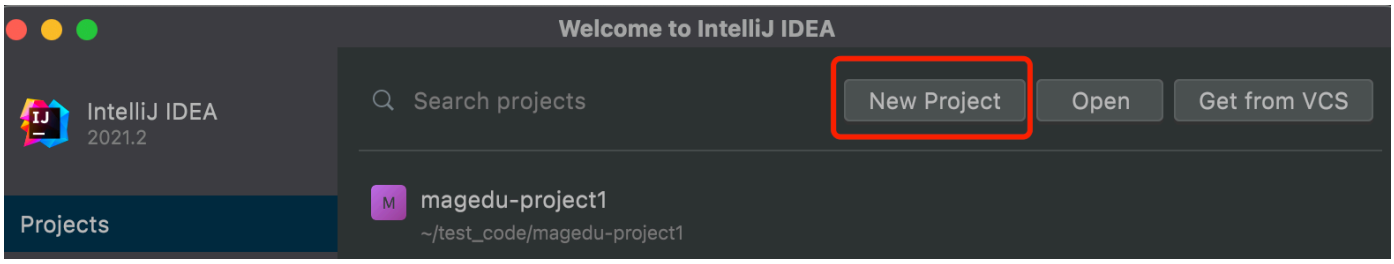
下载链接：https://maven.apache.org/download.cgi

### 安装IDEA

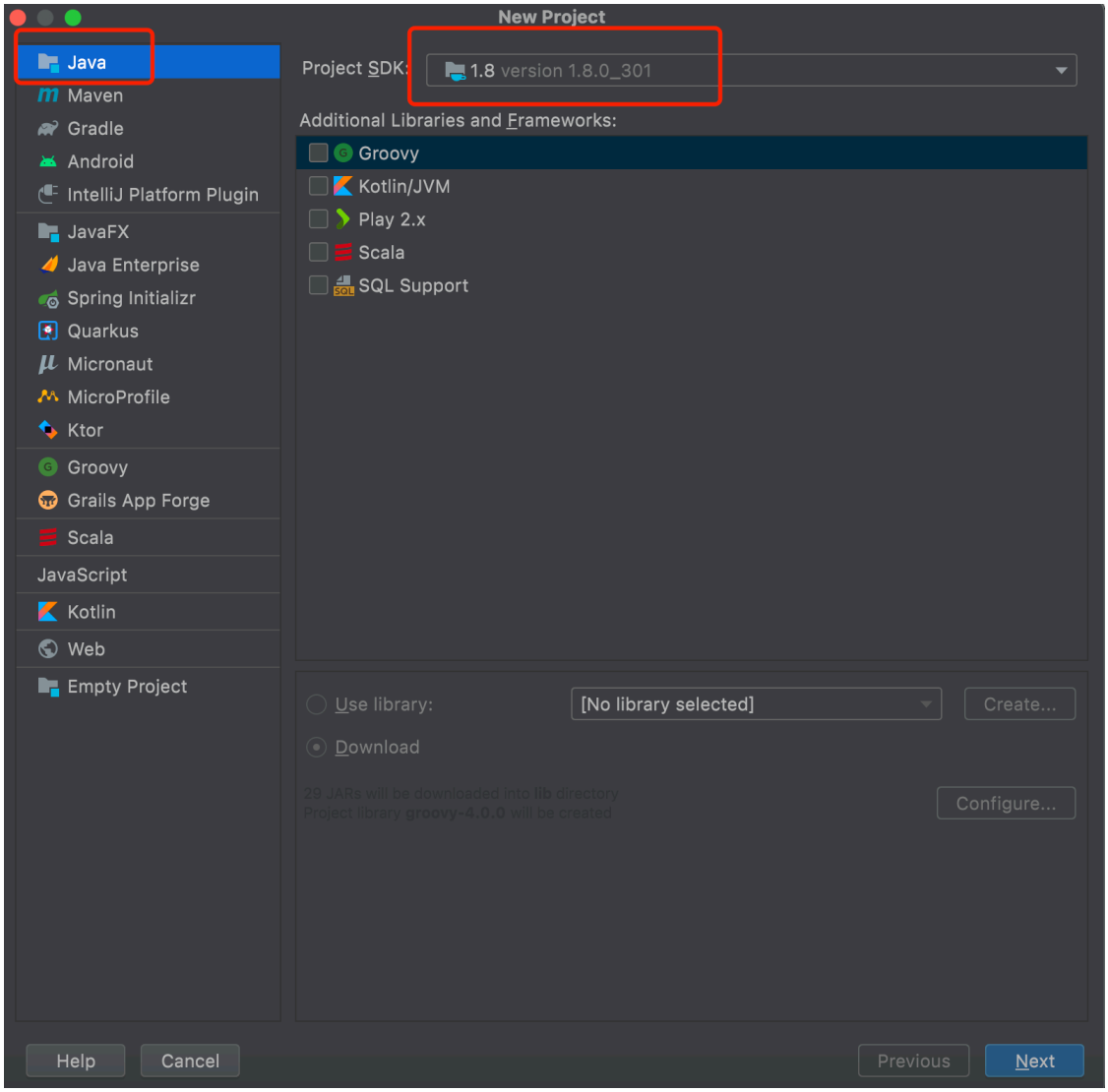
IDEA下载链接


# Hello World-- java开始之旅

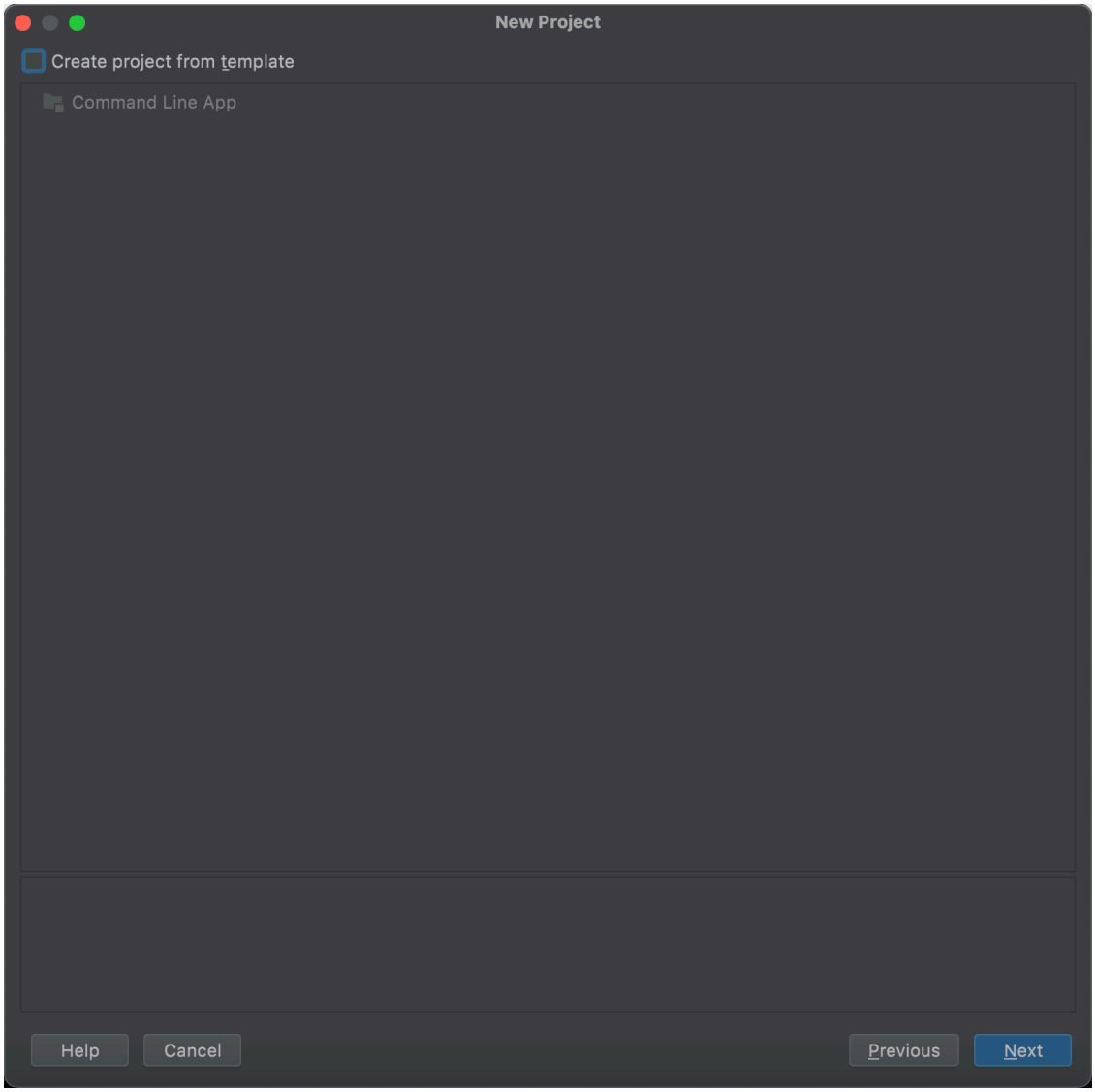
1.新建一个java项目，直接点击 New Project



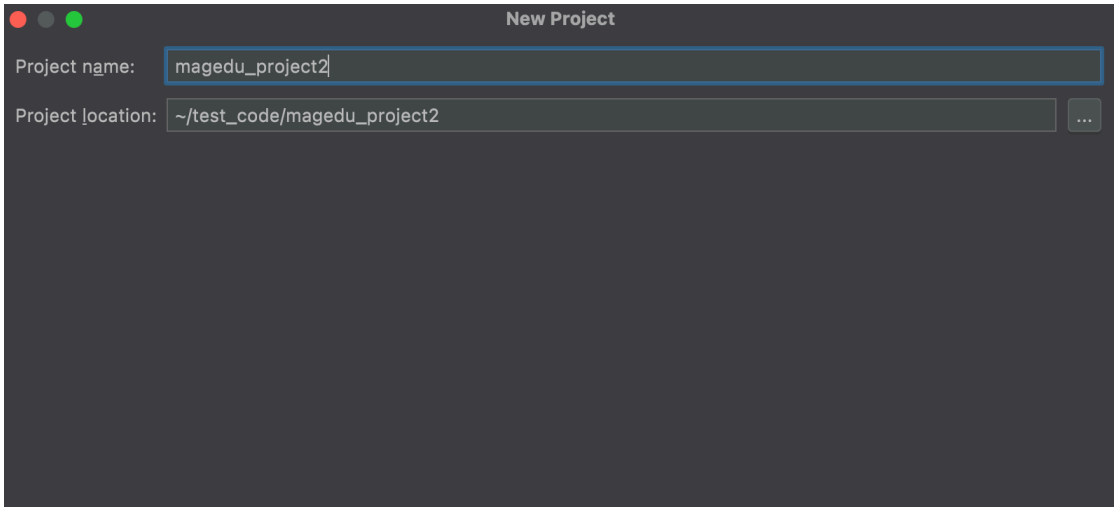

2. 选择Java，sdk选择之前我们安装好的 1.8, 之后点击 Next


New Project
Java
Maven
Gradle
Android
IntelliJ Platform Plugin
JavaFX
Java Enterprise
Spring Initializr
Quarkus
Micronaut
MicroProfile
Ktor
Groovy
Grails App Forge
Scala
JavaScript
Kotlin
Web
Empty Project
Project SDK:
1.8 version 1.8.0_301
Additional Libraries and Frameworks:
Groovy
Kotlin/JVM
Play 2.x
Scala
SQL Support
Use library:
[No library selected]
Create...
Download
29 JARs will be downloaded into lib directory
Project library groovy-4.0.0 will be created
Configure...
Help
Cancel
Previous
Next

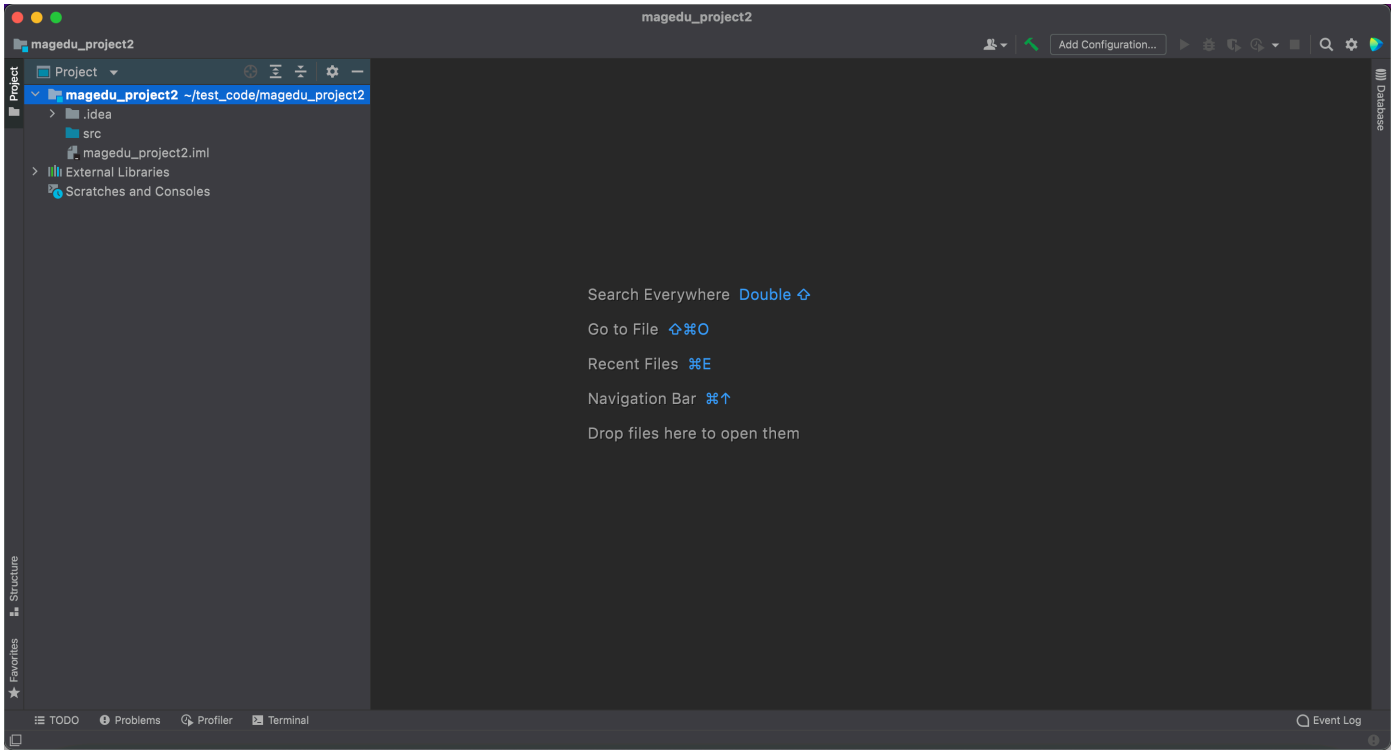
3. 直接点击Next



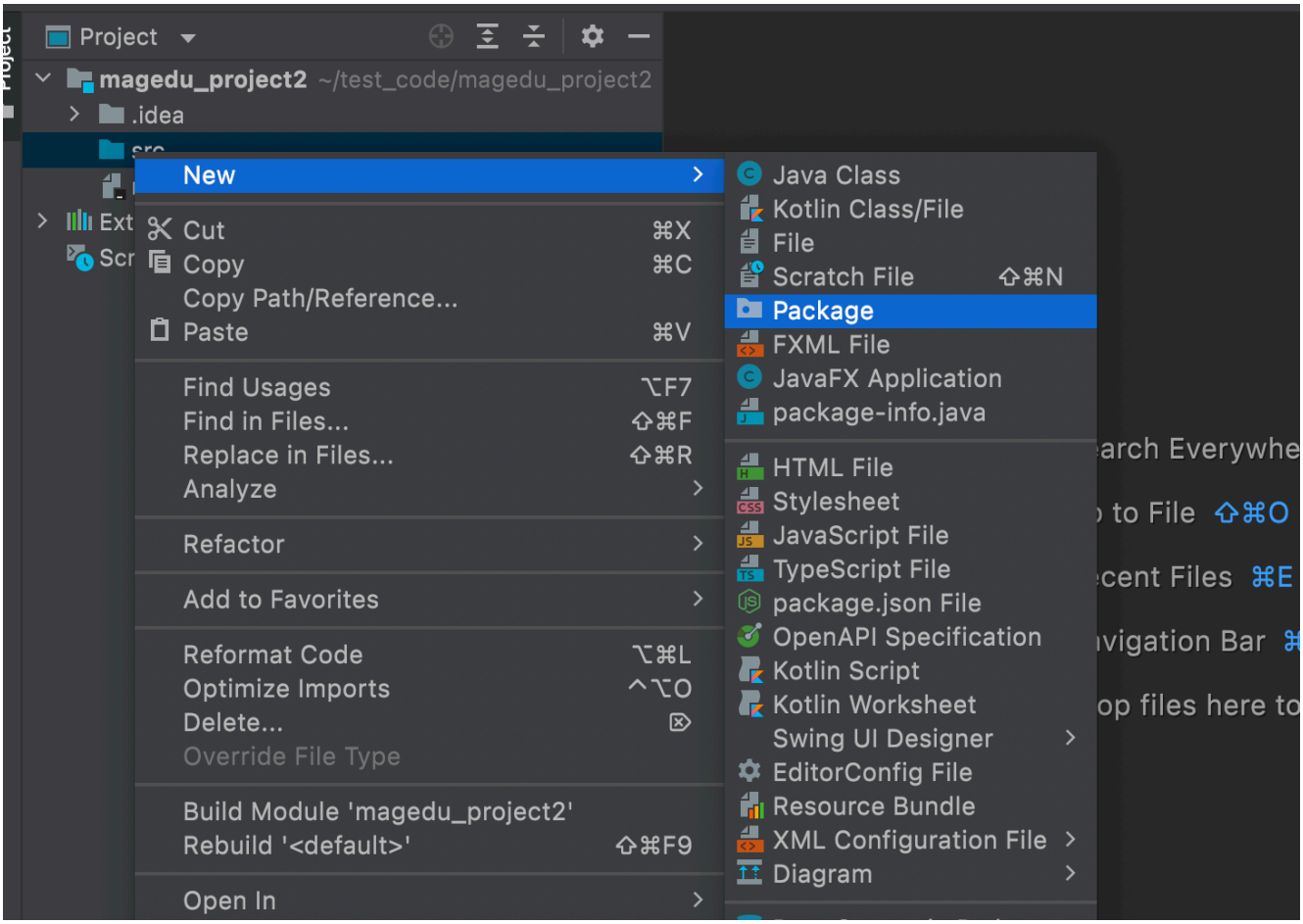

4. 给项目进行命名，并点击 Finish



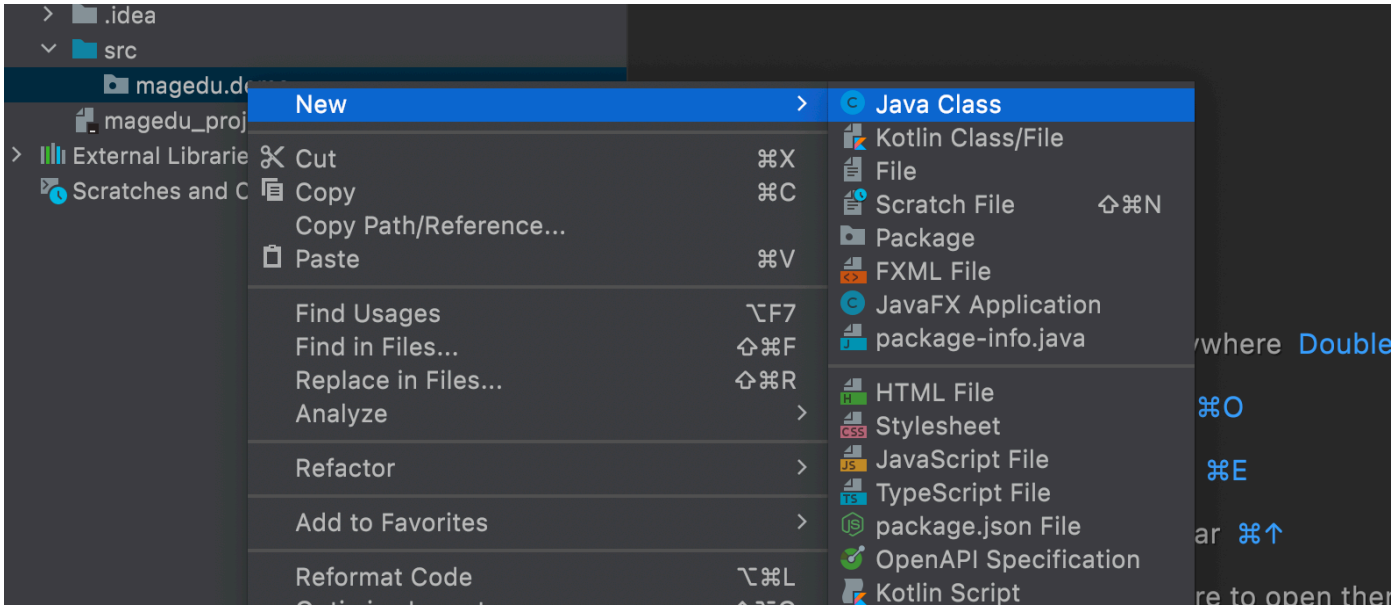

5.创建好之后我们会直接进入到开发界面



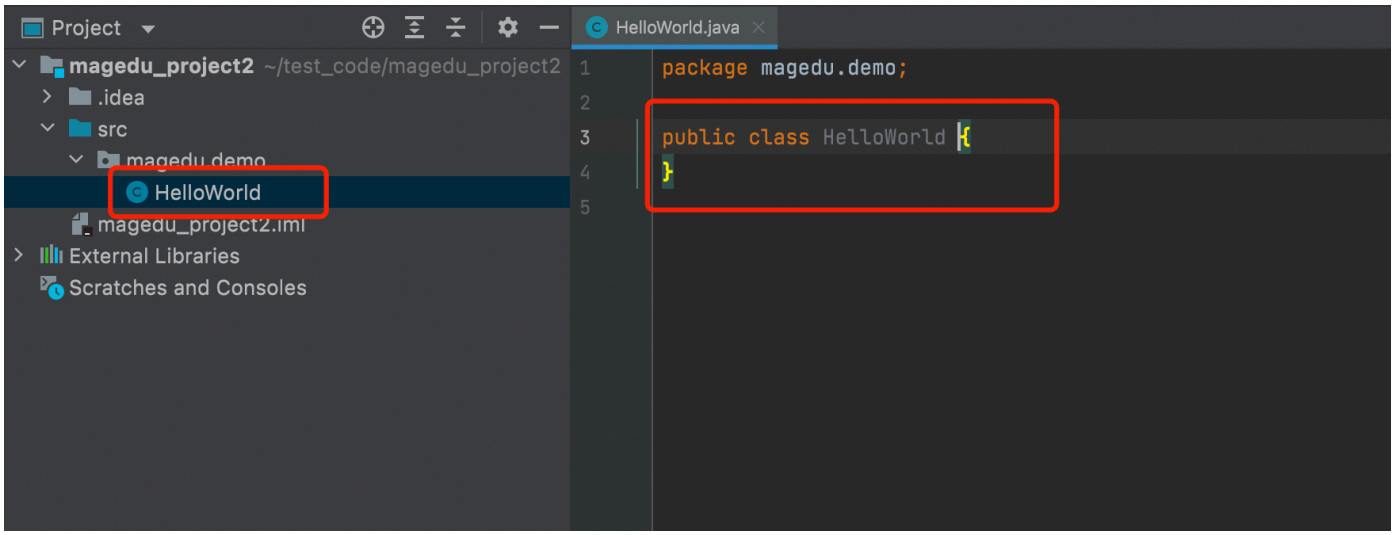

6. 右键点击 src 选择 New -> Package，并进行命名 `magedu.demo`



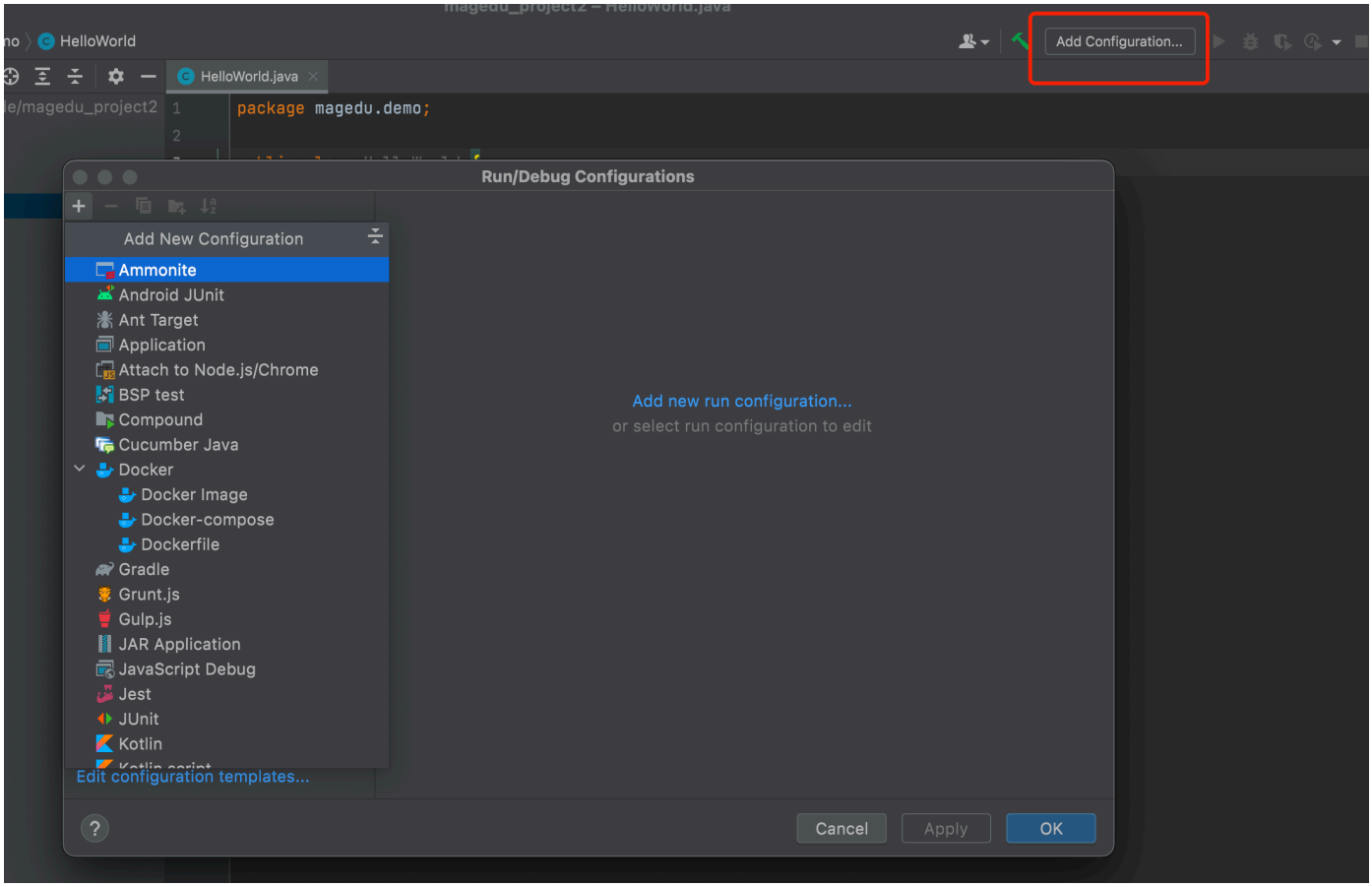

7. 在创建好的package下新建 Java Class 文件




8. 我们这里新建为 HelloWorld， 注意：文件名需要和Java代码的类名保持一致

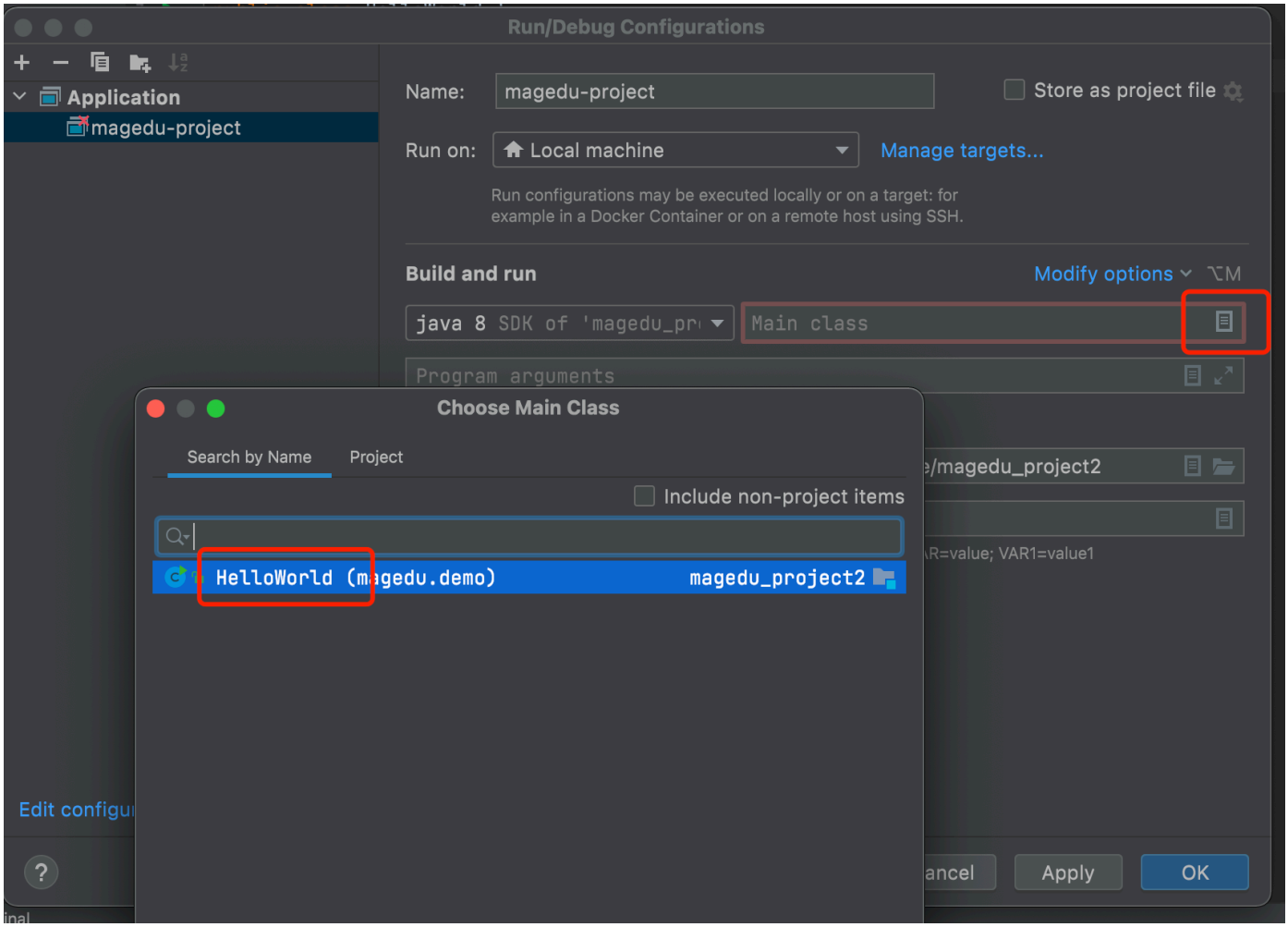



9. 选择右上角 Add Configuration， 在弹出的窗口中选择 Application

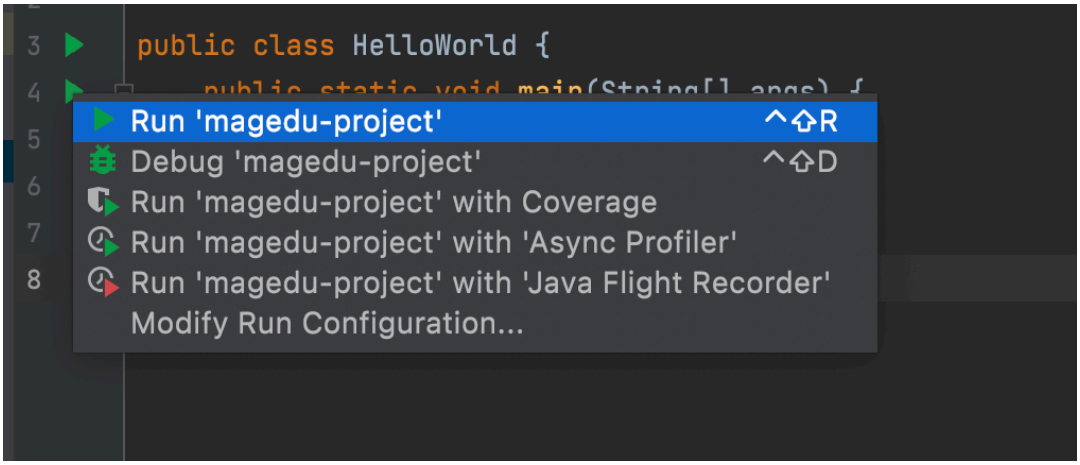



10.需要先在代码中写好 main 函数，之后再进行选择


11. 设置项目名称，选择main函数




12. 运行代码




```
package magedu.demo;

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

# Java 基础语法

一个 Java 程序可以认为是一系列对象的集合，而这些对象通过调用彼此的方法来协同工作。下面简要介绍下类、对象、方法和实例变量的概念。

- **对象**：对象是类的一个实例，有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- **类**：类是一个模板，它描述一类对象的行为和状态。
- **方法**：方法就是行为，一个类可以有很多方法。逻辑运算、数据修改以及所有动作都是在方法中完成的。
- **实例变量**：每个对象都有独特的实例变量，对象的状态由这些实例变量的值决定。

编写 Java 程序时，应注意以下几点：

- **大小写敏感**：Java 是大小写敏感的，这就意味着标识符 Hello 与 hello 是不同的。
- **类名**：对于所有的类来说，类名的首字母应该大写。如果类名由若干单词组成，那么每个单词的首字母应该大写，例如 **MyFirstJavaClass** 。
- **方法名**：所有的方法名都应该以小写字母开头。如果方法名含有若干单词，则后面的每个单词首字母大写。
- **源文件名**：源文件名必须和类名相同。当保存文件的时候，你应该使用类名作为文件名保存（切记 Java 是大小写敏感的），文件名的后缀为 **.java**。（如果文件名和类名不相同则会导致编译错误）。
- **主方法入口**：所有的 Java 程序由 **public static void main(String[] args)** 方法开始执行。

## 例1：BasicDataTypes -- 基本数据类型举例(int, boolean, char)

```
package magedu.demo;

public class BasicDataTypes {
    public static void main(String[] args) {
        byte bt = 127;
        System.out.println(bt);

        int a = 1;
        int b = 2;
        int c = a + b;
        System.out.println("c = " + c);

        boolean flag = true;
        System.out.println("flag = " + flag);
        flag = false;
        System.out.println("flag now = " + flag);

        char c1 = '安';
        System.out.println(c1);
    }
}
```

## 例2：Parameter -- 定义变量、初始化、赋值(初始化)

```
package magedu.demo;

public class Parameter {
    public static void main(String[] args) {
        int a = 3;
//        int b;
        System.out.println("a = " + a);
//        System.out.println("b = " + b);
    }
}
```

## 例3：MathOperators-- 运算符

```
package magedu.demo;


public class MathOperators {
    public static void main(String[] args) {
        int a = 1;
        System.out.println("a = " + a);
        int b = a++;
        //a++ ==> a = a + 1
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("int b = a++ 可分解为 int b = a; \n\t\t\t\t\t a = a+1;");

        System.out.println("***********************************************");
        int c = 10;
        System.out.println("c = " + c);
        int d = ++c;
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("int d = ++c 可分解为 c = c + 1; \n\t\t\t\t   int d = c;");

        //d-- ==> d = d - 1
        d--;
        System.out.println("d = " + d);

    }
}
```

## 例4：RelationOperators -- 运算符

```
package magedu.demo;


public class RelationOperators {
    public static void main(String[] args) {
        int a = 90;
        int b = 90;
        if (a == b) {
            System.out.println("a equals b");
        }
        b--;
        if (a > b) {
            System.out.println("a > b");
        }

        if (a < b) {
            System.out.println("a < b");
        }
    }
}
```

## 例5：LogicalOperators-- 运算符

```
package com.test.basic.chapter2;


public class LogicalOperators {
    public static void main(String[] args) {
        int a = 100;
        int b = 100;

        System.out.println(a > b && a > 99);
        //a大于b并且a大于99
        if (a > b && a > 99) {
            System.out.println("a > 99 and a > b");
        }

        //a大于b或者是a大于99
        System.out.println(a > b || a > 99);
        if (a > b || a > 99) {
            System.out.println("a > 99 but not a > b");
        }

        //a小于等于b
```

```
        System.out.println((a > b));
        System.out.println(!(a > b)); // !(a > b) ==> (a <= b)
        if (!(a > b)) {
            System.out.println("not a > b");
        }

    }
}
```

# 数组和字符串

## [array/ArrayDemo] -- 数组 初始化、遍历、工具类

```
package magedu.arrays;

import java.util.Arrays;

/**
 * 功能：介绍数组
 * 注意：数组是在内存中是一片连续的空间，下标从0开始
 */
public class ArrayDemo {
    public static void main(String[] args) {
        int[] arr = new int[]{1,2,9,5,3};
        System.out.println(arr.toString());
        int sum = 0;
        for (int i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
        System.out.println("数组的和为：" + sum);

        Arrays.sort(arr);
        System.out.println("用工具类Arrays进行排序后的结果：");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + ",");
        }
    }
}
```

## [GetSecondMaxNum]-- 获取数组中的次大数

```
package magedu.arrays;

/**
 * 功能：对于给定的一个数组，找到其中的次大数，并输出
 * 思路：
 *       想类似的情况：找到最大数，这个比较好做，一趟比较下来，能获得最大数
 *       那次大数如何获得，应该是在最大数的情况下，多一次比较，比最大数小的那个就是次大数了
 *       1. 通过for循环取出当前数，把当前数跟最大数和次大数进行比较
 *       2. 另声明一个变量，保存比最大数小的那个数
 *       3. 比最大数大，则次大数为之前的最大数，最大数为当前数；比最大数小，比次大数大，则次大数为当前数
 */
public class GetSecondMaxNum {
    public static void main(String[] args) {
        int[] array = {4, 3, 8, 1, 10, 6};
        for (int i = 0; i < array.length; i++) {
            System.out.print(array[i] + " ");
        }
        int max = array[0];
        int secondMax = 0;
        for (int i = 1; i < array.length; i++) {
            //当前元素是最大数，需要更新最大数
            if (array[i] > max) {
                secondMax = max;
                max = array[i];
            } else if (array[i] <= max && array[i] >= secondMax) {
                //当前元素比次大数大，更新次大数
                secondMax = array[i];
            }
        }
        System.out.println("max=" + max);
        System.out.println("second max=" + secondMax);
    }
}
```

## [BubbleSort]-- 冒泡排序(分析，思路，步骤，代码)

```
package magedu.arrays;

/**
 * 功能：冒泡排序 从小到大排
 * 思路：相邻两个数比较，左边比右边大则交换，整体比较完毕是一次排序
 *       这样的排序要进行n-1趟
 */
public class BubbleSort {
```

```
    public static void main(String[] args) {
        int[] array = new int[]{63, 4, 24, 1, 3, 13};
        System.out.println("冒泡排序法从小到大排序的过程是：");
        //i是趟数
        for (int i = 1; i < array.length; i++) {
            //j是一趟中比较次数
            for (int j = 0; j < array.length - i; j++) {
                if (array[j] > array[j + 1]) {
                    //swap
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
                System.out.print(array[j] + "  ");
            }
            System.out.print("【");
            for (int j = array.length - i; j < array.length; j++) {
                System.out.print(array[j] + "  ");
            }
            System.out.println("】");
        }
    }
}
```

## [string/StringDemo]-- 字符串

```
package magedu.arrays;

/**
 * 功能：介绍字符串
 */
public class StringDemo {
    public static void main(String[] args) {
        String str = new String("abc"); //初始化
        String str1 = "abc"; //初始化
        System.out.println("str == str1? 是否同一个对象：" + (str == str1));
        System.out.println("纯字符串比较：" + ("abc" == "abc"));
        System.out.println("对象str1和字符串比较：" + (str1 == "abc"));
        System.out.println("对象str和字符串比较：" + (str == "abc"));

        System.out.println("字符串长度：" + str.length());
        System.out.println("字符c在字符串str中的位置(下标)：" + str.indexOf("c"));
        System.out.println("字符串str第2个位置是什么字符：" + str.charAt(1));

        str = "    |" + str + "|    ";
```

```
        System.out.println("加上空格后的str: " + str);
        System.out.println("去除str两头的空格后: " + str.trim());
        System.out.println("把|都去除: " + str.replace("|", ""));
        System.out.println("str的内容与str1的内容是否相同: " +
str1.equals(str.trim().replace("|", "")));

        String s = "abcd,efgh,dddj";
        String[] sArray = s.split(",");
        System.out.println(s + " 分割后结果: ");
        for (String item : sArray) {
            System.out.println(item);
        }
    }
}
```

# 流程控制

## 例1：IfDemo-- if分支(if/else)

```
package magedu.flow.control;


public class IfDemo {
    public static void main(String[] args) {
        int age = Integer.parseInt(args[0]);

        System.out.println("您输入的年龄是: " + age);
        System.out.print("您是: ");
        if (age < 8) {
            System.out.println("学龄前儿童");
        } else if (age >= 8 && age < 14) {
            System.out.println("小学生");
        } else if (age >= 14 && age < 20) {
            System.out.println("中学生");
        } else if (age >= 20 && age < 25) {
            System.out.println("大学生");
        } else {
            System.out.println("职场人");
        }
    }
}
```

## 例2：SwitchDemo-- switch分支(switch/case)

```
package magedu.flow.control;


public class SwitchDemo {
    public static void main(String[] args) {
        int level = Integer.parseInt(args[0]);

        System.out.println("您输入的年龄阶段是：" + level);
        System.out.print("您是：");
        switch (level){
            case 1:
                System.out.println("学龄前儿童");
                break;
            case 2:
                System.out.println("小学生");
                break;
            case 3:
                System.out.println("中学生");
                break;
            case 4:
                System.out.println("大学生");
                break;
            default:
            System.out.println("职场人");
        }
    }
}
```

## 例3：ForDemo -- for循环(for)

```
package magedu.flow.control;


public class ForDemo {
    public static void main(String[] args) {

        for(int x = 10; x < 20; x = x+1) {
            System.out.print("value of x : " + x );
            System.out.print("\n");
        }
    }
}
```

## 例4：WhileDemo -- while循环(while/do.while)

```
package magedu.flow.control;


public class WhileDemo {
    public static void main(String[] args) {
        int sum = 0;
        int i = 1;
        while (i <= 100) {
            sum += i++;
        }
        System.out.println("1+2+3+...+100=" + sum);
        System.out.println(i);

//        do while
        int doSum = 0;
        int j = 1;
        do {
            doSum += j++;
        } while (j <= 100);
        System.out.println("1+2+3+...+100=" + doSum);
    }
}
```

## 例5：SkipDemo-- break continue return 跳转语句

```
package magedu.flow.control;


public class SkipDemo {
    public static void main(String[] args) {
        int a = 1;
        while (true) {
            a++;
            //break
            //当a大于3的时候，跳出循环
            if (a > 3) {
                break;
            }
        }

        System.out.println(a);

        a = 1;
        while (a < 10) {
```

```
            a++;
            //当a的值能被2整除，表示该数不是奇数
            if (a % 2 == 0) {
                continue;                   //跳过下面的语句，进入下一次循环
            }
            System.out.print(a + " ");
        }

        return;
//        System.out.println("sss");
    }
}
```

# 类和对象

## 例1：ClassDemo -- 类和对象的概念

```
package magedu.classd;

public class Person {
    String name;    //姓名
    int age;        //年龄
    String gender; //性别 “男”, “女”
}
```

```
package magedu.classd;


public class ClassDemo {

    public static void main(String[] args) {
        Person xiaoming = new Person();
        System.out.println(xiaoming);

        Person xiaowang = new Person();
        System.out.println(xiaowang);
    }
}
```

## 例2：PropertyDemo -- 成员变量

```
package magedu.classd;


public class PropertyDemo {

    public static void main(String[] args) {
        Person xiaoming = new Person();
        xiaoming.name = "小明";
        xiaoming.age = 21;
        xiaoming.gender = "男";
        System.out.println("姓名：" + xiaoming.name + " 年龄：" + xiaoming.age + " 性别："
+ xiaoming.gender);

        Person b;
        b = xiaoming;
        System.out.println(b.name);
        b.age = 200;
        System.out.println(xiaoming.age);

        System.out.println(b);
        System.out.println(xiaoming);
    }
}
```

## 例3：MethodDemo -- 成员方法

```
package magedu.classd;


public class MethodDemo {
    public static void main(String[] args) {
        Person xiaoming = new Person();
        xiaoming.name = "小明";
        xiaoming.age = 21;
        xiaoming.gender = "男";
        System.out.print(xiaoming.name + "说：");
        xiaoming.speak();
    }
}
```

## 例4：Constructor -- 构造函数

```
package magedu.classd;

/**
 *
 * 功能：构造函数
 * 总结：
 *       1.构造函数名和类名相同
 *       2.构造函数没有返回值
 *       3.对新对象的初始化
 *       4.在创建新对象时，系统自动的调用该类的构造函数
 *       5.一个类可以有多个构造函数
 *       6.每个类都有一个默认的构造函数
 */
public class PersonC {
    String name;
    int age;

    PersonC(String pname, int page) {
        name = pname;
        age = page;
    }

    String speak() {
        return "我是" + name + ",今年" + age + "岁";
    }
}
```

```
package magedu.classd;


public class ConstructorDemo {
    public static void main(String[] args) {
        PersonC p = new PersonC("小明", 25);
        System.out.println(p.speak());
    }
}
```

## 例5：ThisDemo -- this在类中的使用

```
package magedu.classd;

/**
 * 功能：this代表当前对象
 * 注意：this不能在类定义的外部使用，只能在类定义的方法中使用
 */
class PersonT {
    String name;
    int age;

    PersonT(String name, int age) {
        this.name = name;
        this.age = age;
    }

    PersonT(String name) {
        this(name, 1);
    }

    String speak() {
        return "我是" + this.name + ",今年" + this.age + "岁";
    }

}
```

```
package magedu.classd;


public class ThisDemo {
    public static void main(String[] args) {
        PersonT p1 = new PersonT("小王", 30);
        System.out.println(p1.speak());
    }

}
```

## 例6：StaticDemo -- static静态变量、静态方法

**定义： 在类中使用static修饰的静态方法会随着类的定义而被分配和装载入内存中；而非静态方法属于对象的具体实例，只有在类的对象创建时在对象的内存中才有这个方法的代码段。**

```
package magedu.classd;

class Student {
    static int totalNo = 0;
    String name;
    int age;
    Student(String name, int age) {
        totalNo++;
    }

    public static void speak() {
        System.out.println("我是红领巾。");
    }

    public int getTotalNo() {
        return totalNo;
    }
}
```

```
package magedu.classd;


public class StaticDemo {
    public static void main(String[] args) {
        Student s1 = new Student("小明", 10);
        Student s2 = new Student("小王", 13);
        System.out.println("总人数: " + s2.getTotalNo());

        Student.speak();
    }
}
```

静态方法，系统会为静态方法分配一个固定的内存空间。而普通方法，会随着对象的调用而加载，当使用完毕，会自动释放掉空间。普通方法的好处是，动态规划了内存空间的使用，节省内存资源。静态方法，方便，运行快，而如果全部方法都用静态方法，那么每个方法都要有一个固定的空间，这样的话太占内存。

因而也就解释了，为什么静态方法可以直接被类名调用，而不需要用对象调用， 因为他有固定空间，随类的加载而加载。

静态方法不需要对象，它在你定义对象就有了，因此就可以方便地直接类名调用。不需要实例化对象。

## 例8：extendsDemo/PersonsMain -- 小学生继承自人，构造函数的继承及引用方式

```
package magedu.classd;

public class Human {
    private String name;
    private char gender;
    private int age;

//    public Human() {
//        System.out.println("Human none con");
//    }

    public Human(String name, char gender, int age) {
        System.out.println("Human...");
        this.name = name;
        this.gender = gender;
        this.age = age;
    }

    protected String getName() {
        return name;
    }

    public void think() {
        System.out.println(this.name + "在思考。。。");
    }
}
```

```
package magedu.classd;

public class Pupil extends Human{
    private String studentNo;

//    public Pupil() {
//        System.out.println("Pupil none con");
//    }

    public Pupil(String studentNo, String name, char gender, int age) {
        super(name, gender, age);  //super用来调用父类构造方法，必须是第一句
        this.studentNo = studentNo;
        System.out.println("Pupil....");
```

```
    }

    public void learn() {
        System.out.println(this.getName() + "在学习。。。");
    }
}
```

```
package magedu.classd;


public class PersonsMain {
    public static void main(String[] args) {
        Human person = new Human("林冲", '男', 30);
        person.think();

        Pupil liming = new Pupil("001", "李明", '男', 8);
        liming.think();
        liming.learn();
    }
}
```

## 例9：overloadVSoverride/OverloadDemo -- 重载

Java 允许同一个类中定义多个同名方法，只要它们的形参列表不同即可。如果同一个类中包含了两个或两个以上方法名相同的方法，但形参列表不同，这种情况被称为方法重载（overload）。

```
package magedu.classd;


public class OverloadDemo {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        getMax(a, b);

        float c = 5f;
        float d = 10f;
        getMax(c, d);
    }

    private static void getMax(int firstNum, int lastNum) {
        if (firstNum > lastNum) {
            System.out.println("较大的数是" + firstNum);
```

```
        } else if (firstNum < lastNum) {
            System.out.println("较大的数是" + lastNum);
        } else {
            System.out.println("两个数相等");
        }
    }

    //只有方法的返回类型不同不能算是重载
//    private static int getMax(int firstNum, int lastNum) {return 1;}

    private static void getMax(float firstNum, float lastNum) {
        if (firstNum > lastNum) {
            System.out.println("较大的数是" + firstNum);
        } else if (firstNum < lastNum) {
            System.out.println("较大的数是" + lastNum);
        } else {
            System.out.println("两个数相等");
        }
    }

    //只有方法的修饰符不同不能算是重载
//    public static void getMax(float firstNum, float lastNum) {}
}
```

## 例10：overloadVSoverride/OverrideDemo -- 重写

在子类中如果创建了一个与父类中相同名称、相同返回值类型、相同参数列表的方法，只是方法体中的实现不同，以实现不同于父类的功能，这种方式被称为方法重写（override），又称为方法覆盖。当父类中的方法无法满足子类需求或子类具有特有功能的时候，需要方法重写。

```
package magedu.classd;

public class Animal {
    private String name;
    private int age;
    //动物都会叫唤
    public void cry() {
        System.out.println("动物都会叫唤，但是具体的某一个种类的动物叫唤方式不同，需要重写我的cry方法");
    }

}
```

```java
package magedu.classd;

public class Cat extends Animal{
    @Override
    public void cry() {
        System.out.println("猫猫叫！ ");
    }
}
```

```java
package magedu.classd;


public class OverrideDemo {
    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.cry();
    }
}
```

## 例11：multiStatus/MultiStatusDemo -- 多态，处理问题的方式

多态性是面向对象编程的又一个重要特征，它是指在父类中定义的属性和方法被子类继承之后，可以具有不同的数据类型或表现出不同的行为，这使得同一个属性或方法在父类及其各个子类中具有不同的含义。

```java
package magedu.classd;

public class Dog extends Animal{
    @Override
    public void eat() {
        System.out.println("狗吃： ");
    }
}
```

```java
package magedu.classd;

public class Bone extends Food {
    @Override
    public void showName() {
        System.out.println("食物是骨头");
    }
}
```

```java
package magedu.classd;

public class Fish extends Food {
    @Override
    public void showName() {
        System.out.println("食物是鱼");
    }
}
```

```java
package magedu.classd;

public class Master {
    //给某种动物喂相应的食物
    public void feed(Animal animal, Food food) {
        System.out.println("主人喂");
        animal.eat();
        food.showName();
    }
}
```

```java
package magedu.classd;

public class CatM extends Animal {
    @Override
    public void eat() {
        System.out.println("猫吃：");
    }
}
```

```
package magedu.classd;


public class MultiStatusDemo {
    public static void main(String[] args) {
        Master master = new Master();
        Dog dog = new Dog();
        Bone bone = new Bone();
        master.feed(dog, bone);
        System.out.println("**********************");
        master.feed(new Cat(), new Fish());
    }
}
```

- 多态就是同一个接口，使用不同的实例而执行不同操作

**多态存在的三个必要条件**
1.继承
2.重写
3.父类引用指向子类对象

## 例12：abstractDemo/AbstractDemo -- 抽象类，关键字 abstract

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，那么这样的类称为抽象类。

在 Java 中抽象类的语法格式如下：

```
<abstract>class<class_name> {
    <abstract><type><method_name>(parameter-iist);
}
```

其中，abstract 表示该类或该方法是抽象的；class_name 表示抽象类的名称；method_name 表示抽象方法名称，parameter-list 表示方法参数列表。

```
package magedu.classd;

public class CatA extends AnimalA {
    @Override
    public void cry() {
        System.out.println("猫猫叫");
    }
}
```

```java
package magedu.classd;

public abstract class AnimalA {
    String name;
    int age;
    //动物会叫
    public abstract void cry();
}
```

```java
package magedu.classd;


public class AbstractDemo {
    public static void main(String[] args) {
        //抽象类不能被实例化
//        AnimalA a = new AnimalA();
        Animal cat = new Cat();
        cat.cry();
    }
}
```

### 抽象类有什么作用? (结合多态思考)

抽象类是用来捕捉子类的通用特性的，是被用来创建继承层级里子类的模板。现实中有些父类中的方法确实没有必要写，因为各个子类中的这个方法肯定会有不同；而写成抽象类，这样看代码时，就知道这是抽象方法，而知道这个方法是在子类中实现的，所以有提示作用。

## 例13：interfaceDemo/InterfaceDemo -- 接口

抽象类是从多个类中抽象出来的模板，如果将这种抽象进行的更彻底，则可以提炼出一种更加特殊的“抽象类”——接口（Interface）。接口是 Java 中最重要的概念之一，它可以被理解为一种特殊的类，不同的是接口的成员没有执行体，是由全局常量和公共的抽象方法所组成。

### 定义接口

Java 接口的定义方式与类基本相同，不过接口定义使用的关键字是 interface，接口定义的语法格式如下：

```
[public] interface interface_name [extends interface1_name[, interface2_name,…]] {
// 接口体，其中可以包含定义常量和声明方法
[public] [static] [final] type constant_name = value;     // 定义常量
[public] [abstract] returnType method_name(parameter_list);     // 声明方法
}
```

对以上语法的说明如下：

- public 表示接口的修饰符，当没有修饰符时，则使用默认的修饰符，此时该接口的访问权限仅局限于所属的包；
- interface_name 表示接口的名称。接口名应与类名采用相同的命名规则，即如果仅从语法角度来看，接口名只要是合法的标识符即可。如果要遵守 Java 可读性规范，则接口名应由多个有意义的单词连缀而成，每个单词首字母大写，单词与单词之间无需任何分隔符。
- extends 表示接口的继承关系；
- interface1_name 表示要继承的接口名称；
- constant_name 表示变量名称，一般是 static 和 final 型的；
- returnType 表示方法的返回值类型；
- parameter_list 表示参数列表，在接口中的方法是没有方法体的。

注意：一个接口可以有多个直接父接口，但接口只能继承接口，不能继承类。

**接口的主要用途就是被实现类实现，一个类可以实现一个或多个接口，继承使用 extends 关键字，实现则使用 implements 关键字。因为一个类可以实现多个接口，这也是 Java 为单继承灵活性不足所作的补充。**

```
package magedu.classd.interfaceDemo;

public abstract class AbstractUSB {

    abstract void start();

}
```

```
package magedu.classd.interfaceDemo;

public class Camera implements IUSB {
    public void start() {
        System.out.println("我是照相机，开始工作了。");
    }

    public void stop() {
        System.out.println("我是照相机，停止工作了。");
    }
}
```

```
package magedu.classd.interfaceDemo;

public class Computer {
    public Computer() {
        System.out.println("计算机启动了");
    }

    public void useUSB(IUSB usb) {
```

```
        usb.start();
        usb.stop();
    }

    public void useAb(AbstractUSB ausb) {
        ausb.start();
    }
}
```

```
package magedu.classd.interfaceDemo;

public class Fen extends AbstractUSB {
    @Override
    void start() {
        System.out.println("feng shan");
    }
}
```

```
package magedu.classd.interfaceDemo;

public interface IUSB {
    //开始工作
    public void start();
    //停止工作
    public void stop();
}
```

```
package magedu.classd.interfaceDemo;

public class Phone implements IUSB{

    public void start() {
        System.out.println("我是手机，开始工作了。");
    }

    public void stop() {
        System.out.println("我是手机，停止工作了。");
    }
}
```

```
package magedu.classd.interfaceDemo;
```

```
public class InterfaceDemo {
    public static void main(String[] args) {
        Computer computer = new Computer();
        IUSB phone = new Phone();
        computer.useUSB(phone);

        //计算机连接照相机
        IUSB camera = new Camera();
        computer.useUSB(camera);

        //计算机连接风扇
        AbstractUSB fen = new Fen();
        computer.useAb(fen);
    }
}
```

### Java 中的接口有什么作用?

例如我定义了一个接口，但是我在继承这个接口的类中还要写接口的实现方法，那我不如直接就在这个类中写实现方法岂不是更便捷，还省去了定义接口?

接口就是个招牌。

接口只是一个规范，所以里面的方法都是空的。

比如说你今年放假出去旅游，你有点饿了，突然看到前面有个店子，上面挂着必胜客，然后你就知道今天中饭有着落了。

必胜客就是接口，我们看到了这个接口，就知道这个店会卖炸鸡、汉堡（实现接口）。

那么为什么我们要去定义一个接口呢，这个店可以直接卖炸鸡、汉堡（直接写实现方法）。

是的，这个店可以直接卖炸鸡，但没有挂必胜客的招牌，我们就不能直接简单粗暴的冲进去叫服务员给两个炸鸡。

要么，我们就要进去问，你这里卖不卖炸鸡腿啊，卖不卖汉堡啊，卖不卖圣代啊（这就是反射）。很显然，这样一家家的问实在是非常麻烦（反射性能很差）。

要么，我们就要记住，灵境胡同108号卖炸鸡，牛街45号卖炸鸡（硬编码），很显然这样我们要记住的很多很多东西（代码量剧增），而且，如果有新的店卖炸鸡腿，我们也不可能知道（不利于扩展）。

#### 什么时候使用接口

如果你想**实现多重继承**，那么你必须使用**接口**。由于Java不支持多继承，子类不能够继承多个类，但可以实现多个接口。因此你就可以使用接口来解决它。

# super关键字的使用

1. 调用子类中重写的父类的方法。
2. 如果超类(superclass)和子类(subclass)都有同名的属性，则访问超类的属性(字段)。
3. 从子类构造函数显式地调用超类无参数化构造函数或参数化构造函数。

下面让我们了解所有这些用途。

# 1.访问超类的重写方法

如果在超类和子类中都定义了相同名称的方法，则子类中的方法将覆盖超类中的方法。这称为方法重写。

## 示例1：方法重写

```
class Animal {

  //方法
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  //重写方法
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){
    display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

**输出结果**

```
I am a dog
```

在本示例中，通过创建Dog类的对象dog1，我们可以调用它的方法printMessage()，然后该方法执行display()语句。

由于display()在两个类中都定义，所以子类Dog的方法覆盖了超类Animal的方法。因此，调用了子类的display()。

如果需要调用超类Animal的重载方法display()，则使用super.display()。

## 示例2：super调用超类方法

```
class Animal {

  //方法
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  //重写方法
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){

    //这调用重写方法
    display();

    // 这调用父类的方法
    super.display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

**输出结果**

```
I am a dog
I am an animal
```

在这里，上述程序是如何工作的。

# 2.访问超（父）类的属性

超类和子类可以具有相同名称的属性。我们使用super关键字来访问超类的属性。

## 示例3：访问超类属性

```
class Animal {
  protected String type="动物";
}

class Dog extends Animal {
  public String type="哺乳动物";

  public void printType() {
    System.out.println("我是 " + type);
    System.out.println("我是一只 " + super.type);
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog = new Dog();
    dog.printType();
  }
}
```

**输出**：

```
我是哺乳动物
我是一只动物
```

在这个实例中，我们在超类Animal和子类Dog中定义了相同的实例字段类型。
然后我们创建了Dog类的对象dog。然后，使用此对象调用printType()方法。

在printType()函数内部，

- type - 指的是子类Dog的属性。
- super.type - 指超类Animal的属性。

因此，System.out.println("我是 " + type);输出 “**我是哺乳动物**”，并且，System.out.println("我是一只 " + super.type);打印输出“**我是一只动物**”。

# 3.使用super()访问超类构造函数

众所周知，创建类的对象时，将自动调用其默认构造函数。

要从子类构造函数中显式调用超类构造函数，我们使用super()。这是super关键字的一种特殊形式。

**注意：super() 只能在子类构造函数中使用，并且必须是第一条语句。**

## 示例4：使用super()

```
class Animal {

  //Animal类的默认或无参数构造函数
  Animal() {
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // Dog类的默认或无参数构造函数
  Dog() {

    //调用超类的默认构造函数
    super();

    System.out.println("I am a dog");
  }
}

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

**输出结果**

```
I am an animal
I am a dog
```

在这里，当Dog类的对象dog被创建时，它会自动调用该类的默认或无参数构造函数。
在子类构造函数中，super()语句调用超类的构造函数并执行其中的语句。因此，我们得到的结果“I am an animal”。

## 示例5：使用super()调用参数化构造函数

```
class Animal {

  //默认或无参数的构造函数
  Animal() {
    System.out.println("I am an animal");
  }
```

```
  //参数化构造函数
  Animal(String type) {
    System.out.println("Type: "+type);
  }
}

class Dog extends Animal {

  //默认构造函数
  Dog() {

    //调用超类的参数化构造函数
    super("Animal");

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

**输出结果**

```
Type: Animal
I am a dog
```

编译器可以自动调用无参数构造函数。但是，它不能调用带有参数的构造函数。

如果必须调用参数化的构造函数，则需要在子类构造函数中显式定义它，如上面代码中的语句：

```
super("Animal");
```

请注意，在上面的示例中，我们使用了super("Animal")，显式地调用参数化构造函数。在这种情况下，编译器不会调用超类的默认构造函数。

# Java 反射(Reflection)

Java中，反射允许我们在运行时检查和操作类、接口、构造函数、方法和字段。

## Java 类名为Class

在学习Java反射之前，我们需要了解一个名为Class的Java类。

Java中有一个名为Class的类，该类在运行时保留有关对象和类的所有信息。

Class对象描述了特定类的属性。该对象用于执行反射。

# 创建名为Class的类的对象

我们可以创建Class的对象，通过：

**使用getClass()方法**

getClass()方法使用特定类的对象来创建新的对象Class。例如，

```
public class test {
    public static void main(String[] args) {
        Dog d = new Dog();
        Class c = d.getClass();
        System.out.println(c.getName());
    }
}
```

- **使用.class**

我们还可以使用**.class**扩展名创建Class对象。例如，

```
Class c = Dog.class;
System.out.println(c.getName());
```

创建Class对象后，我们可以使用这些对象执行反射。

# 获取接口

我们可以使用Class的getInterfaces()方法来收集类实现的接口的信息。此方法返回一个接口数组。

## 示例：获取接口

```
import java.lang.Class;
import java.lang.reflect.*;

interface Animal {
   public void display();
}

interface Animal2 {
   public void makeSound();
}

class Dog implements Animal, Animal2 {
   public void display() {
      System.out.println("I am a dog.");
   }
```

```
    public void makeSound() {
       System.out.println("Bark bark");
    }
}

class ReflectionDemo {
  public static void main(String[] args) {
      try {
          //创建一个Dog类的对象
          Dog d1 = new Dog();

          //使用getClass()创建Class对象
          Class obj = d1.getClass();

          //查找由Dog实现的接口
          Class[] objInterface = obj.getInterfaces();
          for(Class c : objInterface) {

              //打印接口名称
              System.out.println("Interface Name: " + c.getName());
          }
      }

      catch(Exception e) {
          e.printStackTrace();
      }
   }
}
```

**输出结果**

```
Interface Name: Animal
Interface Name: Mammal
```

# 获取超类和访问修饰符

类Class的方法getSuperclass()可用于获取有关特定类的超类的信息。

而且，Class提供了一种getModifier()方法，该方法以整数形式返回class的修饰符。

## 示例：获取超类和访问修饰符

```
package magedu.reflect;

public class Animal {
    public void display() {};
}
```

```java
package magedu.reflect;


public class Dog extends Animal{
    @Override
    public void display() {
        System.out.println("I am a dog.");
    }
}
```

```java
package magedu.reflect;

import java.lang.reflect.Modifier;

public class reflectDemo {
    public static void main(String[] args) {
        try {
            //创建一个Dog类的对象
            Dog d1 = new Dog();

            //使用getClass()创建Class对象
            Class obj = d1.getClass();

            //以整数形式获取Dog的访问修饰符
            int modifier = obj.getModifiers();
            System.out.println("修饰符:  " + Modifier.toString(modifier));

            //找到Dog的超类
            Class superClass = obj.getSuperclass();
            System.out.println("Superclass: " + superClass.getName());
        }

        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

**输出结果**

```
修饰符:  public
Superclass: Animal
```

# 反射字段，方法和构造函数

该软件包java.lang.reflect提供了可用于操作类成员的类。例如

- **方法类** - 提供有关类中方法的信息
- **字段类** - 提供有关类中字段的信息
- **构造函数类** - 提供有关类中构造函数的信息

# Java 反射与字段

我们可以使用Field类提供的各种方法检查和修改类的不同字段。

- **getFields() -** 返回该类及其超类的所有公共字段
- **getDeclaredFields()** - 返回类的所有字段
- **getModifier()** - 以整数形式返回字段的修饰符
- **set(classObject,value)** - 使用指定的值设置字段的值
- **get(classObject)** - 获取字段的值
- **setAccessible(boolean)** - 使私有字段可访问

**注意：**如果我们知道字段名称，则可以使用

- **getField("fieldName")** - 从类返回名称为**fieldName**的公共字段。
- **getDeclaredField*(*"fieldName"*)* *** - 从类返回名称为**fieldName**的字段。

## 示例：访问访问公共字段

```
package magedu.reflect;

class Dog {
    public String type;
}
```

```
package magedu.reflect;

import java.lang.reflect.Field;
import java.lang.reflect.Modifier;

public class reflectDemo {
    public static void main(String[] args) {
        try {
            Dog dog = new Dog();

            Class obj = dog.getClass();
            Field field = obj.getField("type");
            field.set(dog, "labrador");
```

```
            String typeValue = (String)field.get(dog);
            System.out.println("type: " + typeValue);

        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

**输出结果**

```
type: labrador
修饰符:  public
```

## 示例：访问私有字段

```
package magedu.reflect;

class Dog {
    private String color;
}
```

```
package magedu.reflect;

import java.lang.reflect.Field;
import java.lang.reflect.Modifier;

class reflectDemo {
    public static void main(String[] args) {
        try {
            Dog dog = new Dog();
            // 创建类Class对象
            Class obj = dog.getClass();
            // 访问私有字段
            Field field2 = obj.getDeclaredField("color");
            // 使私有字段可访问
            field2.setAccessible(true);
            // 设置color的值
            field2.set(dog, "brown");
            // 获取color的值
            String colorValue = (String)field2.get(dog);
            System.out.println("color: " + colorValue);
```

```
            // 获取color的访问修饰符
            int mod2 = field2.getModifiers();
            String modifier2 = Modifier.toString(mod2);
            System.out.println("modifier: " + modifier2);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

**输出结果**

```
color: brown
modifier: private
```

# Java 反射与方法

像字段一样，我们可以使用Method类提供的各种方法来检查类的不同方法。

- **getMethods()** - 返回该类及其超类的所有公共方法
- **getDeclaredMethod()** - 返回该类的所有方法
- **getName()** - 返回方法的名称
- **getModifiers()** - 以整数形式返回方法的访问修饰符
- **getReturnType()** - 返回方法的返回类型

## 示例：方法反射

```
package magedu.reflect;


class Dog {
    public void display() {
        System.out.println("I am a dog.");
    }

    protected void eat() {
        System.out.println("I eat dog food.");
    }

    private void makeSound() {
        System.out.println("Bark Bark");
    }
}
```

```
package magedu.reflect;
```

```
import java.lang.Class;
import java.lang.reflect.*;


class reflectDemo {
    public static void main(String[] args) {
        try {
            Dog dog = new Dog();

            //创建一个Class对象
            Class obj = dog.getClass();

            //使用getDeclaredMethod()获取所有方法
            Method[] methods = obj.getDeclaredMethods();

            //获取方法的名称
            for(Method m : methods) {

                System.out.println("方法名称:  " + m.getName());

                //获取方法的访问修饰符
                int modifier = m.getModifiers();
                System.out.println("修饰符:  " + Modifier.toString(modifier));

                //获取方法的返回类型
                System.out.println("Return Types: " + m.getReturnType());
                System.out.println(" ");
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

**输出结果**

```
方法名称： display
修饰符： public
Return type: void

方法名称： eat
修饰符： protected
返回类型： void

方法名称： makeSound
修饰符： private
返回类型： void
```

# Java 反射与构造函数

我们还可以使用Constructor类提供的各种方法检查类的不同构造函数。

- **getConstructors()** - 返回该类的所有公共构造函数以及该类的超类
- **getDeclaredConstructor()** -返回所有构造函数
- **getName()** - 返回构造函数的名称
- **getModifiers()** - 以整数形式返回构造函数的访问修饰符
- **getParameterCount()** - 返回构造函数的参数数量

## 示例：构造函数反射

```
package magedu.reflect;

class Dog {
    public Dog() {

    }
    public Dog(int age) {

    }
    private Dog(String sound, String type) {

    }
}
```

```
package magedu.reflect;

import java.lang.Class;
import java.lang.reflect.*;


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

```
        try {
            Dog d1 = new Dog();
            Class obj = d1.getClass();

            //使用getDeclaredConstructor()获取一个类中的所有构造函数
            Constructor[] constructors = obj.getDeclaredConstructors();

            for(Constructor c : constructors) {
                //获取构造函数的名称
                System.out.println("构造函数名称：  " + c.getName());

                //获取构造函数的访问修饰符
                int modifier = c.getModifiers();
                System.out.println("修饰符：  " + Modifier.toString(modifier));

                //获取构造函数中的参数数量
                System.out.println("参数个数：  " + c.getParameterCount());
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

**输出结果**

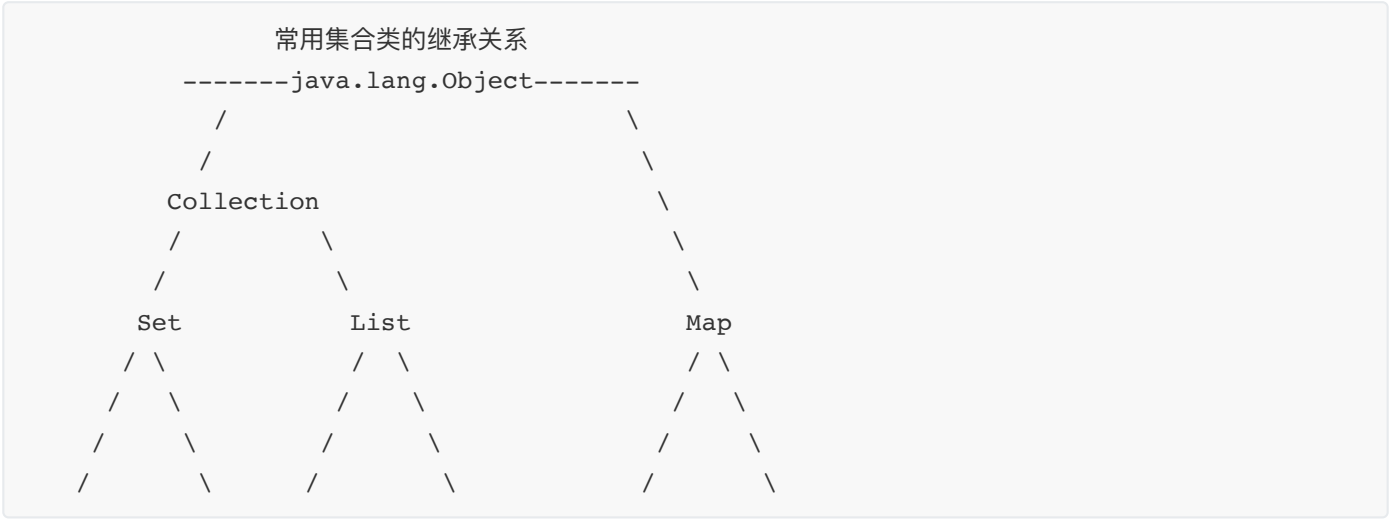
```
构造函数名称：  Dog
修饰符：  public
参数个数：  0

构造函数名称：  Dog
修饰符：  public
参数个数：  1

构造函数名称：  Dog
修饰符：  private
参数个数：  2
```

# 集合

集合 vs 数组：数组的长度是固定的，集合的长度是可变的
常用的集合有List集合、Set集合、Map集合，其中List与Set实现了Collection接口
继承关系




## TreeMapSetDemo

--介绍HashSet的使用方式

```
package magedu.conllection;

public class Person {
    Integer age;

    Person(int Iage){
        age = Iage;
    }
}
```

```
package magedu.conllection;

import java.util.HashSet;
import java.util.Set;


public class SetDemo {
    public static void main(String[] args) {

        Set<String> set = new HashSet<String>();
        set.add("test");
        set.add("test");
        set.add("set1");
        set.add("aaa");
        set.add("bbb");
        //不保证迭代顺序，比方字母排序，数字排序
        for (String item : set) {
            System.out.println(item);
```

```
        }
//        Set<Person> pSet =  new HashSet<Person>();
//        Person p1 = new Person(1111111);
//        Person p2 = new Person(1111111);
//        pSet.add(p1);
//        pSet.add(p2);
//
//        for (Person item : pSet) {
//            System.out.println(item);
//        }
    }
}
```

## ListDemo

--介绍ArrayList的使用方式

```
package magedu.conllection;

import java.util.ArrayList;
import java.util.List;


public class ListDemo {
    public static void main(String[] args) {
        String a = "a";
        String b = "b";
        String c = "c";
        String d = "d";
        String apple = "apple";
        List<String> list = new ArrayList<String>();
        list.add(a);
        list.add(apple);
        list.add(b);
        list.add(apple);
        list.add(c);
        list.add(apple);
        list.add(d);
        System.out.println(list);//输出列表的全部元素
        System.out.println("apple 第一次出现的索引位置是: " + list.indexOf(apple));
        System.out.println("apple 最后一次出现的索引位置是: " + list.lastIndexOf(apple));
    }
}
```

## MapDemo

-- 介绍HashMap的使用方式

```
package magedu.conllection;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;


public class MapDemo2 {
    public static void main(String[] args) {
        Map<String, String> dictionary = new HashMap<String, String>();
        dictionary.put("java", "java编程思想");
        dictionary.put("c", "c语言");
        dictionary.put("shell", "Shell编程");

        System.out.println(dictionary.get("java"));

        for (String value : dictionary.values()) {
            System.out.println("书籍为:" + value);
        }

        Set<String> keys = dictionary.keySet();
        for (String key : keys) {
            System.out.println("书籍代号为: " + key + ", 书籍名称为: " +
dictionary.get(key));
        }
    }
}
```

# 异常

在程序设计和运行的过程中，发生错误是不可避免的。尽管 Java 语言的设计从根本上提供了便于写出整洁、安全代码的方法，并且程序员也尽量地减少错误的产生，但是使程序被迫停止的错误的存在仍然不可避免。

为此，Java 提供了异常处理机制来帮助程序员检查可能出现的错误，以保证程序的可读性和可维护性。

## 例1：WhatsException-- 什么是异常

```
package magedu.unusual;


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

```
        nullPointerException();
        arrayIndexOutOfBoundsException();
    }

    private static void arrayIndexOutOfBoundsException() {
        int[] a = new int[]{1,2,3};
        System.out.println(a[4]);
    }

    private static void nullPointerException() {
        String a = null;
        a.length();
    }

}
```

## 例2：HandleException -- 简单处理异常(try/catch)

```
package magedu.unusual;


public class HandleException {
    public static void main(String[] args) {
        int x = 100;
        int y = 0;
        int z = 0;
        try {
            z = x / y;
            System.out.println(x + "除以" + y + "的商是：" + z);
        } catch (Exception e) {
            e.printStackTrace();//输出异常到标准错误流
            //使用getMessage()方法输出异常信息
            System.out.println("getMessage方法：" + e.getMessage());
        }

    }
}
```

# 泛型

# 概述

泛型，就是“参数化类型”。就是将类型由原来的具体的类型参数化(也就是在写类型的地方当成一个参数来写)，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），

然后在使用/调用时传入具体的类型（类型实参）。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，

操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

---

# 泛型方法

你可以写一个泛型方法，该方法在调用时可以接收不同类型的参数。根据传递给泛型方法的参数类型，编译器适当地处理每一个方法调用。

下面是定义泛型方法的规则：

- 所有泛型方法声明都有一个类型参数声明部分（由尖括号分隔），该类型参数声明部分在方法返回类型之前（在下面例子中的）。
- 每一个类型参数声明部分包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。
- 类型参数能被用来声明返回值类型，并且能作为泛型方法得到的实际参数类型的占位符。
- 泛型方法方法体的声明和其他方法一样。注意类型参数只能代表引用型类型，不能是原始类型（像 int,double,char 的等）。

## 实例

下面的例子演示了如何使用泛型方法打印不同字符串的元素：

```
package magedu;

public class GenericMethodMageTest {
    // 泛型方法 printArray
    public static < A > void printArray(A[] inputArray) {
        // 输出数组元素
        for ( A element : inputArray ){
            System.out.printf( "%s ", element);
        }
        System.out.println();
    }

    public static void main(String args[]) {
        // 创建不同类型数组： Integer, Double 和 Character
        Integer[] intArray = {1, 2, 3, 4, 5};
        Double[] doubleArray = {1.12, 2.23, 3.34, 4.46};
        Character[] charArray = {'M', 'A', 'G', 'E', 'D', 'U'};

        System.out.println("integerArray :");
        printArray(intArray); // 传递一个整型数组
```

```
        System.out.println("doubleArray :");
        printArray(doubleArray); // 传递一个双精度型数组

        System.out.println("characterArray :");
        printArray(charArray); // 传递一个字符型型数组
    }
}
```

运行结果如下所示：

```
integerArray :
1 2 3 4 5
doubleArray :
1.12 2.23 3.34 4.46
characterArray :
M A G E D U
```

---

# 泛型类

泛型类的声明和非泛型类的声明类似，除了在类名后面添加了类型参数声明部分。

和泛型方法一样，泛型类的类型参数声明部分也包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。因为他们接受一个或多个参数，这些类被称为参数化的类或参数化的类型。

## 实例

如下实例演示了我们如何定义一个泛型类:

```
package magedu;

public class MageTest<B> {

    private B t;

    public void add(B t) {
        this.t = t;
    }

    public B get() {
        return t;
    }

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

```
        MageTest<Integer> integerMageTest = new MageTest<Integer>();
        MageTest<String> stringMageTest = new MageTest<String>();

        integerMageTest.add(new Integer(10));
        stringMageTest.add(new String("Hello World"));

        System.out.printf("Integer Value :%d\n\n", integerMageTest.get());
        System.out.printf("String Value :%s\n", stringMageTest.get());
    }
}
```

编译以上代码，运行结果如下所示：

```
Integer Value :10

String Value :Hello World
```

# 读写文件

**FileInputStream** 和 **FileOutputStream**。

---

## FileInputStream

该流用于从文件读取数据，它的对象可以用关键字 new 来创建。

有多种构造方法可用来创建对象。

可以使用字符串类型的文件名来创建一个输入流对象来读取文件：

`InputStream f = new FileInputStream("C:/java/hello");`

也可以使用一个文件对象来创建一个输入流对象来读取文件。我们首先得使用 File() 方法来创建一个文件对象：

`File f = new File("C:/java/hello"); InputStream in = new FileInputStream(f);`

创建了InputStream对象，就可以使用下面的方法来读取流或者进行其他的流操作。

| 序号 | 方法及描述 |
| --- | --- |
| 1 | **public void close() throws IOException{}**<br>关闭此文件输入流并释放与此流有关的所有系统资源。抛出IOException异常。 |
| 2 | **protected void finalize()throws IOException {}**<br>这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出IOException异常。 |
| 3 | **public int read(int r)throws IOException{}**<br>这个方法从 InputStream 对象读取指定字节的数据。返回为整数值。返回下一字节数据，如果已经到结尾则返回-1。 |
| 4 | **public int read(byte[] r) throws IOException{}**<br>这个方法从输入流读取r.length长度的字节。返回读取的字节数。如果是文件结尾则返回-1。 |
| 5 | **public int available() throws IOException{}**<br>返回下一次对此输入流调用的方法可以不受阻塞地从此输入流读取的字节数。返回一个整数值。 |

除了 InputStream 外，还有一些其他的输入流，更多的细节参考下面链接：

- ByteArrayInputStream
- DataInputStream

---

# FileOutputStream

该类用来创建一个文件并向文件中写数据。

如果该流在打开文件进行输出前，目标文件不存在，那么该流会创建该文件。

有两个构造方法可以用来创建 FileOutputStream 对象。

使用字符串类型的文件名来创建一个输出流对象：

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

也可以使用一个文件对象来创建一个输出流来写文件。我们首先得使用File()方法来创建一个文件对象：

```
File f = new File("C:/java/hello"); OutputStream fOut = new FileOutputStream(f);
```

创建OutputStream 对象完成后，就可以使用下面的方法来写入流或者进行其他的流操作。

| 序号 | 方法及描述 |
| --- | --- |
| 1 | **public void close() throws IOException{}**<br>关闭此文件输入流并释放与此流有关的所有系统资源。抛出IOException异常。 |
| 2 | **protected void finalize()throws IOException {}**<br>这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出IOException异常。 |
| 3 | **public void write(int w)throws IOException{}**<br>这个方法把指定的字节写到输出流中。 |
| 4 | **public void write(byte[] w)**<br>把指定数组中w.length长度的字节写到OutputStream中。 |

## 实例

下面是一个演示 InputStream 和 OutputStream 用法的例子：

# mageStreamTest.java 文件代码：

```
package magedu.file;

import java.io.*;

public class mageStreamTest {
    public static void main(String[] args) throws IOException {

        File f = new File("magedu.txt");
        // 构建FileOutputStream对象
        FileOutputStream fos = new FileOutputStream(f);
        // 构建OutputStreamWriter对象,参数可以指定编码,这里指定为UTF-8
        OutputStreamWriter writer = new OutputStreamWriter(fos, "UTF-8");

        writer.append("马哥输入");
        writer.append("\r\n");
        writer.append("Magedu");
        // 关闭写入流
        writer.close();
        // 关闭输出流
        fos.close();
        // 构建FileInputStream对象
        FileInputStream fip = new FileInputStream(f);
        // 构建InputStreamReader对象,编码与写入相同
        InputStreamReader reader = new InputStreamReader(fip, "UTF-8");


        StringBuffer sb = new StringBuffer();
```

```
        while (reader.ready()) {
            sb.append((char) reader.read());
        }
        System.out.println(sb.toString());
        // 关闭读取流
        reader.close();
        // 关闭输入流,释放系统资源
        fip.close();
    }
}
```

# 序列化和反序列化

**序列化**是将某些对象转换为以后可以恢复的数据格式的过程。人们经常序列化对象以便将它们保存到存储中，或者作为通信的一部分发送。

**反序列化**是该过程的逆过程，从某种格式获取结构化数据，并将其重建为对象。今天，用于序列化数据的最流行的数据格式是 JSON。

`ObjectOutputStream`类的`writeObject()`方法可以实现序列化。 `ObjectInputStream`类的`readObject()`方法用于反序列化。

## 示例代码

```
package codeAnalysis;


import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class readObjectTest {
    public static void main(String args[])throws Exception{
        //定义obj对象
        String obj="hello magedu!";
        //创建一个包含对象进行反序列化信息的"object"数据文件
        FileOutputStream fos=new FileOutputStream("magedu_object");
        ObjectOutputStream os=new ObjectOutputStream(fos);
        //writeObject()方法将obj对象写入object文件
        os.writeObject(obj);
        os.close();

        //从文件中反序列化obj对象
        FileInputStream fis=new FileInputStream("magedu_object");
        ObjectInputStream ois=new ObjectInputStream(fis);
```

```
        //恢复对象
        String obj2=(String)ois.readObject();
        System.out.print(obj2);
        ois.close();
    }
}
```

查看 magedu_object 文件内容

```
> xxd magedu_object
00000000: aced 0005 7400 0d68 656c 6c6f 206d 6167  ....t..hello mag
00000010: 6564 7521                                edu!
> cat magedu_object
hello magedu!%
```

这里需要注意的是，`ac ed 00 05`是java序列化内容的特征，如果经过base64编码，那么相对应的是`rO0AB`：

`echo aced0005 | xxd -r -ps | openssl base64`

```
> echo aced0005 | xxd -r -ps | openssl base64
rO0ABQ==
```

## 反序列化漏洞

对 类 进行序列化和反序列化

```
package codeAnalysis;

import java.io.*;

public class readObjectTest2 {
    public static void main(String args[])throws Exception{
        //定义myObj对象
        MyObject myObj = new MyObject();
        myObj.name = "magedu";
        //创建一个包含对象进行反序列化信息的"object"数据文件
        FileOutputStream fos = new FileOutputStream("magedu_object2");
        ObjectOutputStream os = new ObjectOutputStream(fos);
        //writeObject()方法将myObj对象写入object文件
        os.writeObject(myObj);
        os.close();
        //从文件中反序列化obj对象
        FileInputStream fis = new FileInputStream("magedu_object2");
```

```
        ObjectInputStream ois = new ObjectInputStream(fis);
        //恢复对象
        MyObject objectFromDisk = (MyObject)ois.readObject();
        System.out.println(objectFromDisk.name);
        ois.close();
    }
}

class MyObject implements Serializable {
    public String name;
    //重写readObject()方法
    private void readObject(java.io.ObjectInputStream in) throws IOException,
ClassNotFoundException{
        //执行默认的readObject()方法
        in.defaultReadObject();
        //执行打开计算器程序命令
        Runtime.getRuntime().exec("open /Applications/Safari.app");
    }
}
```

可以看到`MyObject`类实现了`Serializable`结果，需要知道，只有实现了`Serializable`接口的类才可以被序列化。