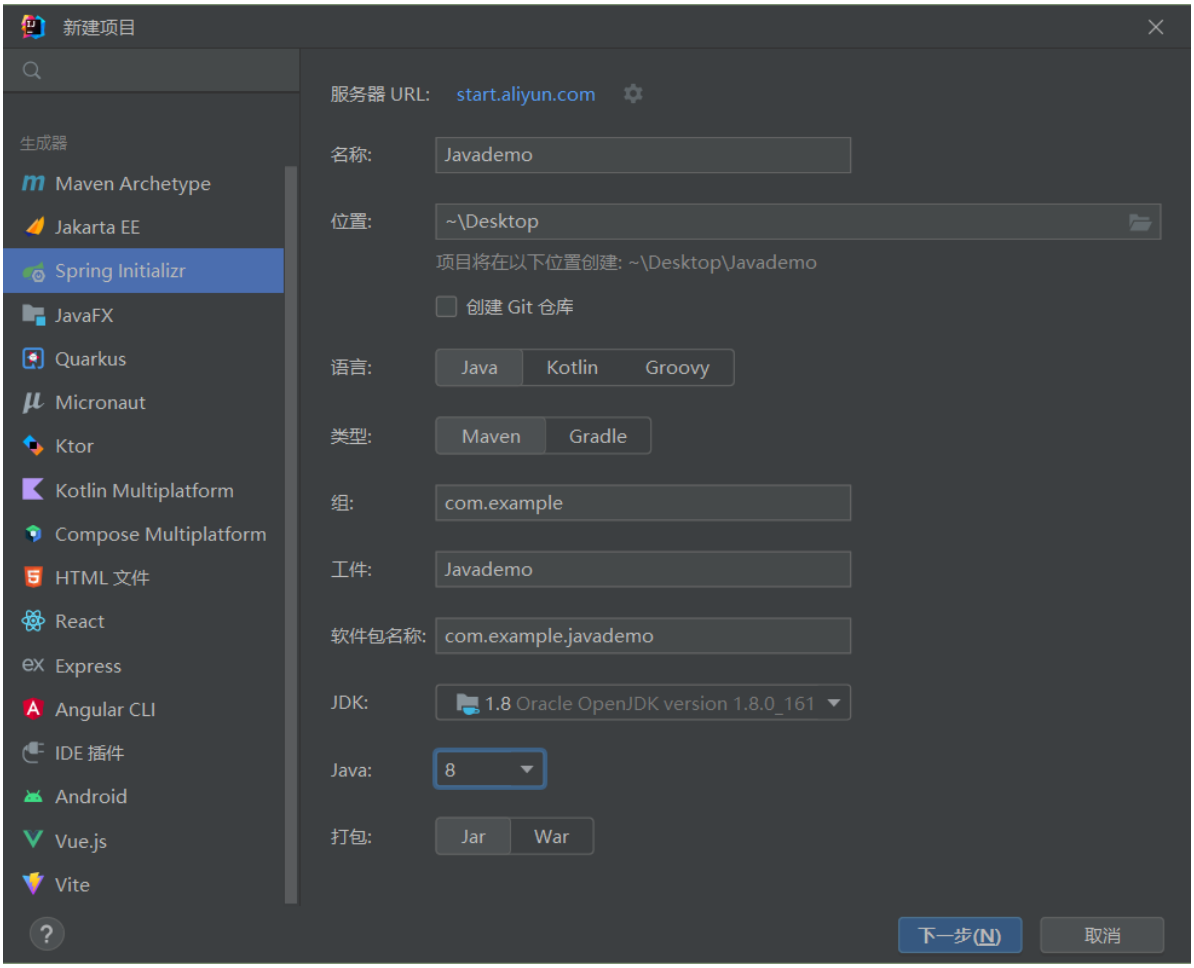
一 基础知识

1 项目搭建

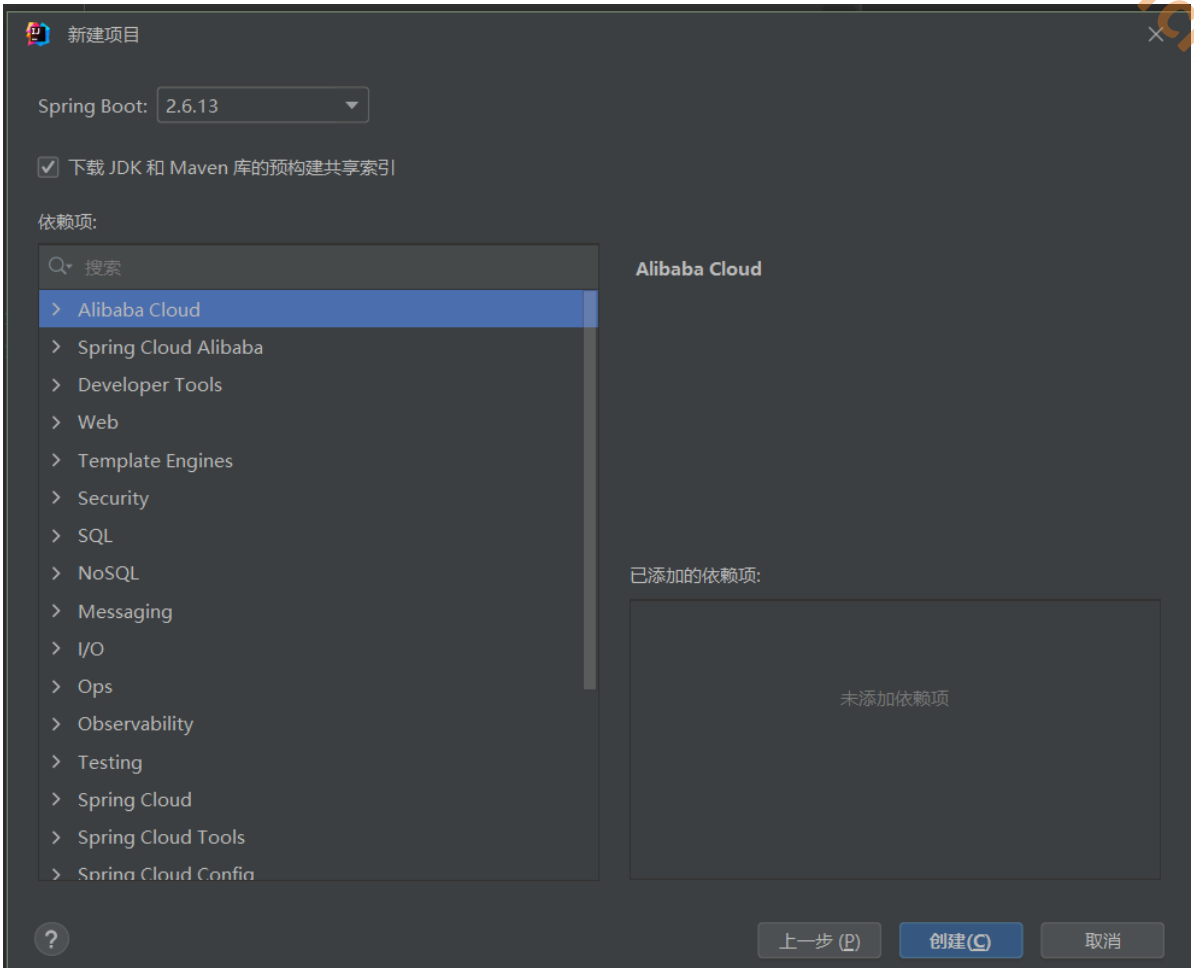
对于下面代码练习，我们需要先创建一个项目。

我个人习惯于直接基础的 SpringBoot 项目，在进行一些代码的调试。

创建方法很简单，在你的第一个 IDEA 项目中也讲到过了。

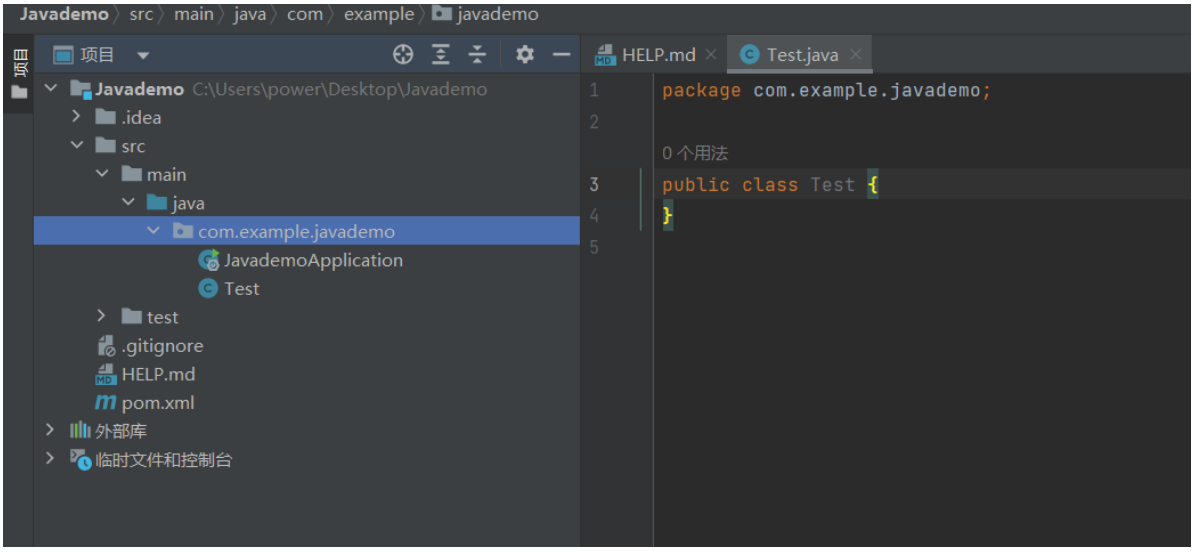


点击下一步，选择依赖项页面，默认内容即可。



最后点击创建。

我们相关练习代码在`src\main\java\com\example\javademo`下右键选中该目录后，新建相关 Java Class 即可，如下图所示：



2 Java 关键字

Java 关键字是对 Java编译器有特殊含义的字符串，是编译器和程序员的一个约定，程序员利用关键字来告诉编译器其声明的变量类型、类、方法特性等信息。

关键字	关键字	关键字	关键字
abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

3 修饰符

public（公开的）：

- **含义：** 使用 `public` 修饰符的类是完全公开的，可以在任何地方访问。
- **使用场景：** 当你希望类的实例能够在任何地方都能被创建和访问时，或者当你编写的是一个库（library）的一部分，希望让其他开发者能够使用这个类。

```
public class MyClass {
    // 类的定义
}
```

没有修饰符（default）：

- **含义：** 如果没有使用任何修饰符，默认情况下，类的可见性为包级别，只能在同一个包中访问。
- **使用场景：** 当你希望类在同一个包中的其他类能够访问，但不希望被包外的类访问时使用。

```
class MyClass {
    // 类的定义
}
```

protected（保护的）：

- **含义：** 使用 `protected` 修饰符的类可以被同一包中的其他类访问，以及该类的子类（无论子类在哪个包中）。
- **使用场景：** 通常较少用于修饰类，更多地用于修饰成员变量和方法。当你希望子类能够访问类的成员，但其他类不能访问时使用。

```
protected class MyClass {
    // 类的定义
}
```

private（私有的）：

- **含义：** 使用 `private` 修饰符的类只能被同一类中的其他类访问，通常用于嵌套类或内部实现细节。
- **使用场景：** 当你希望将类的细节隐藏在同一类的内部，不让其他类直接访问时使用。

```
private class MyClass {
    // 类的定义
}
```

举个例子：

```
public class OuterClass {
    private int outerData;

    // 构造函数
    public OuterClass(int data) {
        this.outerData = data;
    }

    // 内部类
    private class InnerClass {
        private int innerData;

        // 构造函数
        public InnerClass(int data) {
            this.innerData = data;
        }

        public void display() {
            System.out.println("Outer data: " + outerData);
            System.out.println("Inner data: " + innerData);
        }
    }

    // 外部类的方法
    public void outerMethod() {
        InnerClass innerObj = new InnerClass(20);
        innerObj.display();
    }

    public static void main(String[] args) {
        OuterClass Obj = new OuterClass(10);
        outerObj.outerMethod();
    }
```

```
}
```


4 数据类型

4.1 基础概念

在 Java 中，数据类型是用来定义变量的类型，以决定变量可以存储的数据种类。 Java 的数据类型主要分为两大类：基本数据类型和引用数据类型。基本数据类型包括整型（byte short int long） 浮点型（float double） 字符型（char）和布尔型（boolean）。

4.2 代码案例

```
// 整型示例
int integerVariable = 42;

// 浮点型示例
double doubleVariable = 3.14;

// 字符型示例
char charVariable = 'A';

// 布尔型示例
boolean booleanVariable = true;
```

5 变量

5.1 基础概念

变量是用于存储数据的内存空间的标识符。在 Java 中，变量必须先声明后使用，并指定其数据类型。变量的命名要符合 Java 的命名规范，遵循驼峰命名法。

5.2 代码案例

```
// 声明和初始化整型变量
int x = 10;

// 声明和初始化字符串变量
String name = "John";
```

6 基本运算符

6.1 基础概念

基本运算符包括算术运算符（+、-、*、/、%）、关系运算符（==、!=、<、>、<=、>=）、逻辑运算符（&&、||、!）等。它们用于执行常见的数学和逻辑运算。

6.2 代码案例

```
// 算术运算
int a = 5, b = 3;
int sum = a + b;
int difference = a - b;
int product = a * b;
double quotient = (double) a / b; // 注意类型转换
int remainder = a % b;

// 关系运算
boolean isEqual = (a == b);
boolean isGreaterThan = (a > b);

// 逻辑运算
boolean logicalAnd = (true && false);
boolean logicalOr = (true || false);
boolean logicalNot = !true;
```

7 顺序结构

7.1 基础概念

顺序结构是程序中最简单的结构，代码按照书写的顺序一行一行地执行。每一行代码都在前一行代码执行完毕后执行。

7.2 代码案例

```
// 顺序结构示例
int a = 5;
int b = 3;
int sum = a + b;
System.out.println("Sum: " + sum);
```

8 选择结构

8.1 基础概念

选择结构允许根据条件选择执行不同的代码块。在 Java 中，常见的选择结构有if语句、if-else语句和switch语句。

8.2 代码案例

```
// if语句示例
int number = 6;
if (number / 2 == 3) {
    System.out.println("1");
} else {
    System.out.println("2");
}
```

```java
public class NestedIfExample {

    public static void main(String[] args) {
        int x = 5;
        int y = 10;

        if (x > 0) {
            System.out.println("X 大于 0.");

            if (y > 0) {
                System.out.println("Y 也大于 0.");
            } else {
                System.out.println("y 小于 0.");
            }
        } else {
            System.out.println("X 小于0.");
        }
    }
}
```

9 循环结构

9.1 基础概念

循环结构允许多次执行相同的代码块，直到满足退出条件。在 Java 中，常见的循环结构有 for 循环、while 循环和 do-while 循环。

9.2 代码案例

```java
for (初始化; 循环条件; 循环迭代) {
    // 循环体
}
```

```java
// for循环示例
for (int i = 0; i < 5; i++) {
    System.out.println("当前值为" + i);
}
```

10 数组

10.1 基础概念

数组是一种存储相同类型数据的集合。在 Java 中，数组是固定大小的，可以通过索引访问其中的元素。

10.2 代码案例


```
// 数组示例
int[] numbers = {1, 2, 3, 4, 5};
System.out.println("First element: " + numbers[0]);
```

```
int[] numbers = {1, 2, 3, 4, 5};
for (int i = 0; i < numbers.length; i++) {
    System.out.println("数组元素为: " + numbers[i]);
}
```

11 函数方法

在 Java 中，函数方法是一组执行特定任务的代码块。方法提供了程序的模块化和重用性。方法由方法名、参数列表、返回类型和方法体组成。 Java 中的方法可以分为普通方法和静态方法，普通方法需要实例化对象调用，而静态方法属于类，可以通过类名直接调用。

```
public class MyClass {
    // 普通方法
    public void printMessage(String message) {
        System.out.println(message);
    }

    public static void main(String[] args) {
        MyClass myObject = new MyClass();
        myObject.printMessage("Hello, World!");
    }
}
```

```
/** 返回两个整型变量数据的较大值 */
public static int max(int num1, int num2) {
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

如果我们想要使用，只需`max(1,2)`即可。

```java
public class VoidMethodExample {
    public static void main(String[] args) {
        // 调用无返回值的方法
        displayMessage();
    }

    // 一个无返回值的方法，用于显示一条简单的消息
    public static void displayMessage() {
        System.out.println("这是一个无返回值的方法代码案例。");
    }
}
```

12 Java 异常处理

异常是指在程序运行时发生的意外情况，可能导致程序中断或产生不可预知的结果。Java 中的异常处理机制通过 try、catch、finally 块来捕获和处理异常。异常分为检查异常（Checked Exception）和非检查异常（Unchecked Exception）。

```java
public class ExceptionHandling {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // 会抛出ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero.");
        } finally {
            System.out.println("Finally block always executes.");
        }
    }
}
```

`try` **块：** 在 `try` 块内，包含可能引发异常的代码。在这里，`int result = 10 / 0;` 尝试进行除法运算，但由于除数是0，这会导致算术异常。

`catch` **块：** 如果在 `try` 块中发生了异常，控制流会跳转到对应的 `catch` 块。在这里，`catch (ArithmeticException e)` 捕获了 `ArithmeticException` 异常，表示发生了除以零的情况。`catch` 块内的代码会被执行，输出提示信息："Cannot divide by zero."。

`finally` **块：** 无论是否发生异常，`finally` 块中的代码总是会被执行。在这里，输出信息："Finally block always executes."。`finally` 块通常用于确保资源的释放或清理工作，无论是否发生异常。

二 面向对象

面向对象（Object-Oriented，简称OOP）是一种编程思想和程序设计范式，它将程序中的数据和操作数据的方法组织成对象。在面向对象编程中，对象是程序的基本单元，每个对象可以包含数据（称为属性）和方法（称为行为）。

1 类

在编程中，类（Class）是一种抽象数据类型，用于描述具有相似属性和行为的对象集合。它是面向对象编程（Object-Oriented Programming，OOP）的核心概念之一，通过类可以创建对象，而对象则是类的实例。

我们初中生物讲过生物类，鸟类，鱼类等。这两个不同的类别，它们有着不同的特征和行为。同样，在编程中，类是一种将数据和方法组合在一起的结构，用于描述某种抽象概念或实体。

简单来说，类是一种用于描述对象的蓝图或模板。它定义了对象的属性（成员变量）和行为（方法）。


```
// 定义一个Bird类
public class Bird {
    // 成员变量，描述鸟类的属性
    private String feathers;
    private String beakShape;

    // 构造方法，用于初始化对象的属性
    public Bird(String feathers, String beakShape) {
        this.feathers = feathers;
        this.beakShape = beakShape;
    }

    // 成员方法，描述鸟类的行为
    public void fly() {
        System.out.println("The bird is flying.");
    }

    // Getter方法，用于获取羽毛属性
    public String getFeathers() {
        return feathers;
    }

    // Setter方法，用于设置羽毛属性
    public void setFeathers(String feathers) {
        this.feathers = feathers;
    }
}

// 在另一个类中使用Bird类
public class BirdExample {
    public static void main(String[] args) {
        // 创建一个鹰对象
        Bird eagle = new Bird("brown", "hooked");

        // 调用鹰对象的飞行方法
        eagle.fly();

        // 获取并输出鹰的羽毛属性
        String feathers = eagle.getFeathers();
        System.out.println("Feathers: " + feathers);

        // 设置新的羽毛属性并输出
        eagle.setFeathers("golden");
        System.out.println("New Feathers: " + eagle.getFeathers());
    }
}
```

2 对象

前面说到通过类可以创建对象，而对象则是类的实例。

实例化是为了创建对象，也就是我们使用类这个模板，以及可以进行自己所需的改动。

举个简单例子，我们从网上使用寻找 PPT 模板，最后下载使用，改成自己所需的内容。

这个过程和创建对象有些类似。

在 Java 中，实例化一个对象的过程通常包括使用 `new` 关键字来调用类的构造方法，并为对象分配内存空间。以下是一个简单的例子，演示如何在 Java 中实例化对象：

```
public class Dog {
    // 类的属性
    String name;
    int age;

    // 类的方法
    public void bark() {
        System.out.println("Woof! Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        // 实例化一个 Dog 类的对象
        Dog myDog = new Dog();

        // 设置对象的属性值
        myDog.name = "Buddy";
        myDog.age = 3;

        // 调用对象的方法
        myDog.bark();

        // 输出对象的属性值
        System.out.println("Name: " + myDog.name);
        System.out.println("Age: " + myDog.age);
    }
}
```

```
// 定义一个简单的类
class PPTTemplate {
    // 成员变量
    String templateName;

    // 构造方法
    public PPTTemplate(String name) {
        this.templateName = name;
    }
```

```
    // 成员方法
    public void customize(String content) {
        System.out.println("Customizing PPT template '" + templateName + "' with
content: " + content);
    }
}

// 在另一个类中使用 PPTTemplate 类
public class PPTCreator {
    public static void main(String[] args) {
        // 实例化 PPTTemplate 类，创建一个对象
        PPTTemplate myTemplate = new PPTTemplate("SimpleTemplate");

        // 调用对象的成员方法
        myTemplate.customize("This is my custom content.");

        // 创建另一个对象
        PPTTemplate anotherTemplate = new PPTTemplate("FancyTemplate");
        anotherTemplate.customize("Adding some fancy graphics and animations.");

        // ...
    }
}
```

在这个例子中，我们定义了一个简单的 `PPTTemplate` 类，其中包含了一个构造方法用于初始化对象的成员变量 `templateName`，以及一个成员方法 `customize` 用于自定义模板内容。然后，在 `PPTCreator` 类中，我们实例化了两个不同的 `PPTTemplate` 对象，每个对象代表一个不同的PPT模板，并通过调用对象的方法来进行自定义。

关键步骤：

① `PPTTemplate` 类的定义包括构造方法和成员方法。

② 在 `PPTCreator` 类的 `main` 方法中，通过 `new PPTTemplate("SimpleTemplate")` 实例化一个 `PPTTemplate` 对象，并指定模板名称为 "SimpleTemplate"。

③ 通过调用对象的成员方法 `myTemplate.customize("This is my custom content.");` 来自定义模板内容。

④ 同样地，创建另一个对象 `anotherTemplate` 并进行自定义。

实例化对象的过程涉及到为对象分配内存 调用构造方法进行初始化等步骤，这样就可以创建多个相互独立的对象，每个对象都有自己的状态（成员变量值）和行为（成员方法）。

3 继承

基础概念：

继承是面向对象编程中的概念，允许一个类（子类）继承另一个类（父类）的属性和方法。子类可以继承父类的行为，并且可以通过添加新的属性和方法来扩展其功能。

解决的问题：

继承解决了代码重用和扩展的问题。通过继承，子类可以复用父类的代码，而不必重复实现相同的功能。同时，子类可以在保留父类功能的基础上，添加新的功能或修改部分功能，实现功能的扩展和定制。

代码案例：

```
// 父类
class Animal {
    void eat() {
        System.out.println("动物正在吃");
    }
}

// 子类继承父类
class Dog extends Animal {
    void bark() {
        System.out.println("狗在叫");
    }
}

public class Main {
    public static void main(String[] args) {
        // 创建子类对象
        Dog myDog = new Dog();

        // 调用继承自父类的方法
        myDog.eat();

        // 调用子类自己的方法
        myDog.bark();
    }
}
```

4 封装

基础概念：

封装是将对象的内部状态和实现细节隐藏起来，只对外提供访问接口。在 Java 中，通过访问修饰符（如`private` `public`）来实现封装。

解决的问题：

封装解决了对象的安全性和灵活性问题。通过将内部细节隐藏，防止外部直接访问对象的属性，从而保护对象的状态。同时，通过提供公共的方法，使得对象能够以受控制的方式被外部访问和修改。

代码案例：

```
public class Person {
    // 私有属性
    private String name;
    private int age;

    // Getter方法，用于获取name属性值
    public String getName() {
        return this.name;
    }

    // Setter方法，用于设置name属性值，并进行非空验证
    public void setName(String newName) {
```

```
        if (newName != null && !newName.isEmpty()) {
            this.name = newName;
        } else {
            System.out.println("Name cannot be null or empty.");
        }
    }

    // Getter方法，用于获取age属性值
    public int getAge() {
        return this.age;
    }

    // Setter方法，用于设置age属性值，并进行年龄范围验证
    public void setAge(int newAge) {
        if (newAge >= 0 && newAge <= 150) {
            this.age = newAge;
        } else {
            System.out.println("Age must be between 0 and 150.");
        }
    }

    public static void main(String[] args) {
        // 创建一个Person对象
        Person person = new Person();

        // 使用setter设置属性值
        person.setName("John");
        person.setAge(25);

        // 使用getter获取属性值并输出
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());

        // 尝试设置无效值
        person.setName(""); // 输出错误信息
        person.setAge(200); // 输出错误信息
    }
}
```

5 构造函数

基础概念：

构造函数是一个特殊的方法，与类同名，没有返回类型。它在对象被创建时调用，用于执行初始化操作。构造函数的主要目的是确保对象在被使用之前处于一个合理的状态。

构造函数可以显示的定义，也就是我们根据所需设置构造函数。

如果没有显示的提供构造函数，类仍然是可以被实例化的。

因为，如果你没有为类定义任何构造函数， Java 编译器会为你生成一个默认的无参数构造函数。这个构造函数会执行以下操作：

- 将类的实例变量初始化为默认值（数值型为0，布尔型为false，对象型为null等）。
- 如果类继承自其他类，会调用父类的无参数构造函数。

在构造函数中，又分为有参构造函数和无参构造函数。

都很好理解，就是是否需要传入参数。

如果你显式地提供了自定义的构造函数（无论是有参数的还是无参数的），并且没有提供任何无参数构造函数，那么默认的无参数构造函数就不再自动生成。

也就是说除非目标类显示的自定义了无参构造函数，否则如果目标类只定义了有参构造函数的话，那就不会默认生成无参构造函数了。

解决的问题：

构造函数解决了对象初始化的问题。通过构造函数，可以为对象的属性赋予初始值，执行必要的设置，使对象能够在创建时就具备正确的状态。

代码案例（无参构造函数）：

```
public class Dog {
    private String name;
    private int age;

    // 构造函数，初始值
    public Dog() {
        this.name = "juzi";
        this.age = 3;
    }

    // 获取狗的名字
    public String getName() {
        return name;
    }

    // 获取狗的年龄
    public int getAge() {
        return age;
    }

    public static void main(String[] args) {
        // 实例化一个Dog对象，使用初始值
        Dog myDog = new Dog();

        // 获取狗的信息并打印
        System.out.println("Dog's name: " + myDog.getName() + ", Age: " +
myDog.getAge());
    }
}
```

代码案例（有参构造函数）：

```
public class Dog {
    private String name;
    private int age;

    // 无参构造函数，使用默认值
    public Dog() {
        this.name = "juzi";
        this.age = 3;
    }
```

```
    // 有参构造函数，接受名字和年龄参数
    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // 获取狗的名字
    public String getName() {
        return name;
    }

    // 获取狗的年龄
    public int getAge() {
        return age;
    }

    public static void main(String[] args) {
        // 使用无参构造函数实例化一个Dog对象，使用默认值
        Dog defaultDog = new Dog();
        System.out.println("Default Dog's name: " + defaultDog.getName() + ",
Age: " + defaultDog.getAge());

        // 使用有参构造函数实例化一个Dog对象，提供自定义值
        Dog customDog = new Dog("Buddy", 5);
        System.out.println("Custom Dog's name: " + customDog.getName() + ", Age:
" + customDog.getAge());
    }
}
```

5.1 继承中的构造函数

默认情况：

- 如果父类（基类）有一个无参数的构造函数，子类（派生类）会自动继承这个无参数构造函数。
- 如果子类没有显式定义构造函数，编译器会默认生成一个无参数构造函数，并在其中调用父类的无参数构造函数。

```
class Animal {
    // 父类有一个无参数构造函数
}

class Dog extends Animal {
    // 子类没有显式定义构造函数，编译器默认生成一个无参数构造函数
}
```

父类有有参数构造函数：

- 如果父类只提供了有参数的构造函数，子类需要显式定义构造函数，并通过`super()`调用适当的父类构造函数。此时`super()`方法是必须的。

```
class Animal {
    public Animal(String name) {
        // 父类有一个有参数构造函数
    }
}

class Dog extends Animal {
    public Dog(String name, String breed) {
        super(name); // 调用父类的有参数构造函数
        // 初始化子类特有的属性
    }
}
```

子类提供无参数构造函数：

- 如果父类没有提供无参数构造函数，但子类需要使用无参数构造函数，子类需要显式提供无参数构造函数，并通过`super()`调用适当的父类构造函数。此时`super()`方法是必须的。

```
class Animal {
    public Animal(String name) {
        // 父类有一个有参数构造函数
    }
}

class Dog extends Animal {
    public Dog() {
        super("DefaultName"); // 调用父类的有参数构造函数
        // 子类提供无参数构造函数，并在其中调用父类构造函数
    }
}
```

总的来说，继承和构造函数的关系取决于父类的构造函数情况。在设计时，需要考虑如何在子类中正确地初始化父类的状态。

6 函数方法重载

函数方法的重载是指在同一个类中可以定义多个方法，它们具有相同的方法名但具有不同的参数列表。编译器根据方法的参数数量、类型或顺序来选择合适的方法。

下面计算器示例代码也是一个比较经典的案例了，可以使用重载来实现任意加减乘除的运算。

```
public class Calculator {
    // 重载方法
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public String add(String a, String b) {
        return a + b;
    }
```

```
    public static void main(String[] args) {
        Calculator myCalculator = new Calculator();
        System.out.println(myCalculator.add(2, 3));
        System.out.println(myCalculator.add(2.5, 3.5));
        System.out.println(myCalculator.add("Hello", " World"));
    }
}
```

7 构造函数中的重载

基础概念：

构造函数的重载是指在同一个类中定义多个构造函数，它们具有相同的名称但参数列表不同。通过构造函数的重载，可以提供多种初始化对象的方式。

解决的问题：

构造函数的重载解决了不同场景下对象初始化需求不同的问题。通过提供多个构造函数，使得用户能够选择适合自己需求的初始化方式。

代码案例：

```
public class Person {
    private String name;
    private int age;

    // 构造函数重载
    public Person() {
        this.name = "Unknown";
        this.age = 0;
    }

    public Person(String name) {
        this.name = name;
        this.age = 0;
    }

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

    // 其他类成员和方法...
}
```

三 一些其他知识

1 HashMap

基础概念

`HashMap` 是 Java 中常用的集合类之一，用于存储Key-Value键值对的集合。它实现了 `Map` 接口，用于存储键值对。`HashMap` 的核心思想是通过散列算法将键映射到存储桶，提高查找效率。基本操作的时间复杂度为 O(1)。然而，需要注意 `HashMap` 不是线程安全的，如果在多线程环境中使用，可以考虑使用 `ConcurrentHashMap`。

在 Java 中，键值对是一种常见的数据结构，通常用于表示关联关系。键值对包含两部分：键（key）和值（value）。键是唯一的，通过键可以访问对应的值。

在 HashMap 中，最常用的就是 get 和 put 了，通过名字也能知道，一个是获取键值，一个是存入键值。

代码案例

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {

    public static void main(String[] args) {
        // 创建HashMap实例
        Map<String, Integer> hashMap = new HashMap<>();

        // 添加键值对
        hashMap.put("Java", 1);
        hashMap.put("Python", 2);
        hashMap.put("JavaScript", 3);

        // 获取值
        int javaValue = hashMap.get("Java");
        System.out.println("Value for Java: " + javaValue);

        // 遍历键值对
        for (Map.Entry<String, Integer> entry : hashMap.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
        }
    }
}
```

2 StringBuilder

基础概念

`StringBuilder` 是 `java.lang` 包中的一个类，用于在单线程环境下对字符串进行可变操作，避免了使用 `String` 类时的不断创建新字符串的开销。它提供了一系列方法用于修改字符串内容，是一个可变的字符序列。

代码案例

```
public class StringBuilderExample {

    public static void main(String[] args) {
        StringBuilder stringBuilder = new StringBuilder("Hello");
```

```
        // 追加字符串
        stringBuilder.append(" World");

        // 插入字符串
        stringBuilder.insert(5, ", Java");

        // 替换字符串
        stringBuilder.replace(6, 11, "GPT");

        // 删除字符串
        stringBuilder.delete(12, 17);

        // 输出结果
        System.out.println(stringBuilder.toString());
    }
}
```

3 StringBuffer

基础概念

`StringBuffer` 与 `StringBuilder` 类似，也是可变的字符序列。主要区别在于 `StringBuffer` 是线程安全的，因此在多线程环境中更适用。然而，由于同步的开销，`StringBuilder` 在单线程情况下可能更高效。

代码案例

```
public class StringBufferExample {

    public static void main(String[] args) {
        StringBuffer stringBuffer = new StringBuffer("Hello");

        // 追加字符串
        stringBuffer.append(" World");

        // 插入字符串
        stringBuffer.insert(5, ", Java");

        // 替换字符串
        stringBuffer.replace(6, 11, "GPT");

        // 删除字符串
        stringBuffer.delete(12, 17);

        // 输出结果
        System.out.println(stringBuffer.toString());
    }
}
```

4 IO 流

基础概念

输入输出流（IO流）是 Java 中用于处理输入和输出的机制。它分为字节流和字符流，以及输入流和输出流。常见的 IO 类有 `FileInputStream`、`FileOutputStream`、`BufferedReader`、`BufferedWriter` 等。

可以进行文件读取，网络操作，缓冲操作读取字节流，对象序列化等操作，也是比较重要的。在第一阶段后面几个章节我们也会常接触的。

代码案例

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class IOExample {

    public static void main(String[] args) {
        try {
            // 读取文件
            BufferedReader reader = new BufferedReader(new
FileReader("input.txt"));
            String line = reader.readLine();
            while (line != null) {
                System.out.println(line);
                line = reader.readLine();
            }
            reader.close();

            // 写入文件
            FileWriter writer = new FileWriter("output.txt");
            writer.write("Hello, IO!");
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

5 Object

基础概念

`Object` 类是 Java 中所有类的根类，每个类都是 `Object` 类的子类。它定义了一些基本的方法，如 `toString`、`equals` 和 `hashCode`。在 Java 中，所有对象都可以被赋值给 `Object` 类型的变量。

```
https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html
```

`toString()` 方法：

- 用于返回对象的字符串表示。默认情况下，`toString()` 返回类名后跟对象的哈希码。

`equals(Object obj)` 方法：

- 用于比较两个对象是否相等。默认实现是比较对象的引用地址，但通常在子类中会被重写以根据业务逻辑判断对象是否相等。

`hashCode()` **方法：**

- 返回对象的哈希码。哈希码用于在哈希表等数据结构中快速查找对象。

`getClass()` **方法：**

- 返回对象的运行时类，即对象所属的类。

`notify()`、`notifyAll()` **和** `wait()` **方法：**

- 用于线程间的协调和通信。这些方法通常与多线程编程有关，用于实现线程的等待和唤醒机制。

`finalize()` **方法：**

- 在垃圾回收器清理对象之前调用。子类可以重写此方法以执行资源清理等操作。

`clone()` **方法：**

- 创建并返回一个对象的副本。默认情况下，`clone()` 方法执行的是浅拷贝，但可以在子类中重写以实现深拷贝。

`getClass()` **方法：**

- 返回对象的运行时类，即对象所属的类。

`wait()`，`notify()`，`notifyAll()`：

- 用于线程间的协调和通信，通常与多线程编程相关。