

《面向对象程序先导》

Lec2-编写类与单元测试

北京航空航天大学计算机学院

吴际

2023.9.15

大纲

- Java程序的类
 - 构造方法
 - 查询方法
 - 修改方法
- 对象引用与方法调用
 - 容器初步了解
- Java程序的单元测试
 - Junit介绍
 - 编写Junit测试
 - 工具配置和使用
- 本周作业介绍

C语言与Java语言

- C 语言
 - 面向过程语言，函数为编程单位
 - 程序功能由函数及其调用来实现
- Java语言
 - 面向对象语言，类为编程单位
 - 程序功能由类及其协作来实现
- 都有复杂性控制问题
 - 规模控制
 - 逻辑控制
 - ==》形成层次化设计

类

- 类是一种自定义数据类型，包含数据以及操作数据的方法
 - 相当于对结构体和操作结构体函数的封装
- 类的定义方式是结构化的
 - `public class className {`
 - `//属性定义部分`
 - `//方法定义部分`
 - `}`
- 属性和方法在类中的出现位置是自由的，处于同一个层次
 - 方法中定义的变量是**局部变量**，不是类的属性
- 类中应该放哪些属性和方法？
 - 你的程序需要这个类干什么？→数据+方法
 - 例：银行管理系统需要银行账号类，来管理与账号相关的一些数据和必要操作行为

方法中声明的变量与类中声明的变量有什么区别？

Java程序中的主类

- 一个 Java 程序可以包含多个类，其中有一个类拥有 `main` 方法，酷似 C 的主函数
 - `public static void main(String[] args){...}`
- 这个 `main` 方法就是**程序入口**，拥有这种 `main` 方法的类称为主类
 - 主类一般不设置属性
 - `main`方法常见流程
 - 构建业务类对象
 - 获取输入
 - 调用业务类对象处理获得的输入
 - 打印输出处理结果

`main`方法一般不做具体的业务处理，而是把业务数据和处理封装成单独的业务类！

第一次手搓Java程序

- 假设你有一个银行账号，你想要打印输出余额，然后存500元后再打印输出余额。
 - 银行账号类为BankAccount，假设已存在
 - `public class HelloBank{`
 - `public static void main(String[] args){`
 - `BankAccount myAccount = new BankAccount("/小李");`
 - `System.out.println("The balance is: " + myAccount.getBalance());`
 - `myAccount.deposit(500);`
 - `System.out.println("The balance is: " + myAccount.getBalance());`

System是java.lang包中的类，专门用于输入和输出，out是它的静态成员，println是out对象提供的方法

println调用参数中的 '+' 是什么意思？

实现BankAccount业务类

- 银行账号类管理个人银行相关业务数据，并提供相关的操作
 - 属性：账户名、账号、银行名称、余额
 - 方法：查看余额、存款、取款、（付利息）、（查看交易历史记录）
- 请同学们手动来实现这个类

```
public class BankAccount{  
    private String name;  
    private String ID;  
    private String bankBranch;  
    private int balance;  
    public int getBalance(){return balance;}  
    public String getAccountName(){return name;}  
}
```

```
    public boolean withdraw(int amount){  
        if(amount > balance) return false;  
        //记录本次交易  
        balance -= amount;  
        //执行取钱  
        return true;  
    }  
    public boolean deposit(int amount){  
        ??  
    }  
}
```

存取款方法可能会遇到什么异常？

类的属性和方法识别与编写

- 遵循业务需要
- 从业务角度分析属性的取值范围约束
- 从业务角度分析方法调用参数的范围限制
- 从业务角度分析对象状况的可能异常
- 分析程序执行时可能出现的异常情况

类成员的可见性

- 可见性(visibility)不是人感知的光学可见性，而是类中方法能否访问另一个对象或类中成员的规则
 - public：任意外部对象都能访问
 - protected：任意本类对象或子类对象都能访问
 - private：只有本类对象才能访问

```
public class BankAccount{  
    public boolean MergeAccount(BankAccount ba){  
        if(ba.name.equals(this.name)){...};  
        else return false;  
    }  
}
```

能否使用
(ba.name == this.name) ?

类的属性

- BankAccount类的属性支撑了它的业务功能
 - 存取款、看余额等
- 如何知道一个类该有哪些属性？
 - 复杂的问题
 - 原则1：你为什么需要这个类，属性来回答
 - 原则2：依据需求（指导书）
 - 原则3：与方法相匹配，刚刚好才是真的好

构造方法

- 构造方法用来初始化对象属性
- 何时调用构造方法?
 - `BankAccount b = new BankAccount("***");`
- Java语言默认为每个类“配”一个无参数的构造方法
 - 所有属性都按照默认值来初始化
- 一旦用户自己定义了构造方法，默认的那个便失效
 - `new` 操作时可能会报编译错误
- 需要哪些构造方法
 - 必要性原则

查询方法

- 用来查询一个对象所管理的私有化属性数据 (getter)
- 也可以返回对象数据适当变换或计算后的结果
 - toString方法
- 查询方法不会修改对象属性
- 问题：是否所有private属性数据都提供查询方法？

修改方法

- 用于更改对象私有属性数据的方法
 - withdraw和deposit方法
 - 一旦出错，难以恢复
- 修改方法对于一个类实现其业务目标而言具有决定性
 - 银行账号之所以是账号，核心不是查询，而是能够存取款
- 一定要清楚修改方法成功执行前提条件和执行效果
 - 成功存款的前提条件：账号状态正常、存款额大于0
 - 成功存款的执行效果：账号状态正常、余额增加量等于存款额
- 假设BankAccount提供一个setBalance(amount)方法，是否合理？

重名的方法

- Java允许在一个类中定义多个重名的方法
 - 前提：这些方法的参数列表必须有差异：类型序列和个数
 - `int func(int,int); boolean func(int,int)`，这两个func是否可以在一个类中？
 - `int func(int,String); int func(int,int)`，这两个func是否可以在一个类中？
- 面向对象术语：重载
 - 多个方法重名，说明这些方法具有相近的功能
 - 一个方法名“背着”多个相近但又有差异的功能
 - ==》多态效果

重名的方法

- 提供多个构造方法是经常出现的
应用场景，用来做不同的对象初始化
 - 注意：初始化的完整性！
- 业务的必要性
 - 现实世界：ATM直接取现金、通过微信取款
 - 逻辑世界：需要提供多个withdraw方法吗？

```
public class BankAccount {  
    String ID;  
    int balance;  
    //构造函数的重载  
    public BankAccount(String id) {  
        this.ID = id;  
        this.balance = 0;  
    }  
    //参数数量不同  
    public BankAccount(String id, int balance){  
        this.ID = id;  
        this.balance = balance;  
    }  
    //参数类型不同  
    public BankAccount(String balanceStr) {  
        this.balance = Integer.parseInt(balanceStr);  
    }  
}
```

创建对象

- 设计好类之后，就可以创建对象
 - 是不是所有的类都可以实例化对象？
- 创建对象要选择合适的构造方法
 - 取决于掌握了多少可用来初始化对象的数据
- 对象创建后一定要有一个变量引用它
 - `public void func(...){`
 - `new BankAccount ("xyz");`
 - `}`
- 如果未创建对象会如何？

```
public class Secret{  
    private String secrets;  
    private Secret (String s){  
        secrets = s;  
    }  
}
```

```
public void func (...){  
    BankAccount ba;  
    ba.deposit(300);  
}
```


对象引用

- 程序中使用类声明的一个变量
 - 相当于C语言的指针
 - 默认是null
- 通过对象引用可以访问所引用对象的属性和方法
- 要确保对象引用的类型与所指向实际对象的类型一致
 - `BankAccount ba = new String("sss");`
 - `BankAccount ba = new Object();`
 - `Object ba = new String("sss");`
 - `Object ba = new BankAccount("xyz");`

方法调用的参数传递

- Java中的方法调用与C中的函数调用本质一致
 - 通过栈来管理
 - 调用时把被调用函数压栈，返回时弹栈
- C程序函数调用传参
 - 值还是引用？

```
typedef struct _point{int x, y;} point;  
void swap(point p){  
    int temp = p.x;  
    p.x = p.y;  
    p.y = temp;  
}
```

C 中的结构体是值，传递参数后拷贝了一份，形参值的改变不会导致实参值变化

方法调用的参数传递

- Java中调用参数传递的是值还是引用？

```
public class Point { int x; int y; }
public class MainClass {
    public static void swap1(int x, int y){
        int temp=x; x=y; y=temp;
    }
    public static void swap2(Point p){
        int temp=p.x; p.x=p.y; p.y=temp;
    }
    public static void main(String[] args){
        Point p; p.x =1; p.y =2;
        swap1(p.x, p.y);
        swap2(p);
    }
}
```

swap1和swap2哪个
能实现x与y的交换？

值与引用

- Java中除了基础类型，所有使用类所声明的变量都是引用
 - 用户自定义类、Java类库定义的类
 - 泛指class、enum、interface
- Java使用任意类型声明的数组变量也是引用
 - `int[] arr;`
 - `BankAccount [] balist;`
- 参数调用时
 - 基础类型直接传递值（即拷贝）
 - 引用类型传递引用本身（可理解为对象地址）

```
swap3(Point p1, Point p2){  
    Point temp = p1;  
    p1 = p2;  
    p2 = temp;  
}
```

Java 和 C 都是采用值传递 (pass-by-value)。当传的是基本类型变量时，传值拷贝，对变量的修改不影响原变量；当传的是引用类型时，引用地址拷贝

对象引用作为属性

- 在定义类的属性时，难以只使用基础类型
 - 使用一个类来声明对象引用（作为属性成员）
 - 在构造方法中调用相应类型的构造方法来设置该属性成员
 - ```
public class Box{
 • private Ball ball;
 • Box(...){...ball = new Ball(...);}
 • }
```
- 此时这两个类之间产生了依赖（关联）
- 有时一个类需要管理多个其他类型的业务对象
  - 枚举/静态数组/动态数组/容器

# 对象容器

- Java提供的一整套解决方案
- 用来管理数量不定且动态变化的一组对象
- 提供了灵活的遍历和插入方法
- 常见的容器有 ArrayList 和 HashMap
  - ArrayList 相当于不定长数组，可以根据需要在运行期自动扩展长度
  - HashMap 相当于散列表，可极速检索到所需对象
- 今天的作业指导书和下节课会详细介绍这些容器

# ArrayList的初步了解

- ArrayList提供插入、检索、克隆和删除操作
  - add, get, clone, remove
- ArrayList存储对象引用，而不是对象值
  - 如果要存int怎么办？
- ArrayList<Bottle>表明这个容器中只能存放Bottle类型的对象
  - 也可以是Bottle的子类对象
- 一种新的for语句，遍历容器中的对象
  - for(Bottle b:bottleArray){...}

```
class Adventurer {
 private int id;
 private String name;
 private ArrayList<Bottle> bottleArray;
 private HashMap<Integer, Bottle> bottleMap;
}
```

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

# HashMap的初步了解

- HashMap 是一个散列表
  - 存储键值对 (key-value) 映射，key和value都是对象引用
- ArrayList中对象引用有序存放
- HashMap 无序，不会记录插入的顺序
  - 通过key对象来映射到相应的value对象
- 插入操作： `bottleMap.put(b.getId(),b)`
- 读取操作： `Bottle b = bottleMap.get(12345)`
- 判断是否存在某个对象： `containsKey(b.getId())` 或 `containsValue(b)`
- 删除操作： `remove(b.getId())`或`remove(b.getId(), b)`

```
class Adventurer {
 private int id;
 private String name;
 private ArrayList<Bottle> bottleArray;
 private HashMap<Integer, Bottle> bottleMap;
}
```



# 类是Java的编程和测试单元

- 一个类在语法上必须完整，否则无法编译
- 如何发现一个类的设计与实现是否有错？
- 测试是发现程序bug的工程化方法
  - 使用最为广泛
  - 静态测试：检查代码逻辑来发现问题
  - 动态测试：执行代码来发现问题
- 单元测试是一种动态测试
  - Java：针对类

# Java程序的单元测试

- 单元测试与调试
- 共同点
  - 都需要了解和结合代码逻辑
  - 都需要执行代码
- 不同点
  - 测试的目标是发现bug
  - 调试的目标是定位和修复bug
- 手段不同
  - 测试：设计输入数据
  - 调试：设置断点

# Java程序的单元测试

- 原则上每个类都应进行单元测试
- 隔离式测试
  - 当一个类依赖于其他类时，首先需要对被依赖的类进行测试，然后逐层向上
  - 可以快速定位问题范围
- Junit是一个所有Java开发者都应掌握的单元测试方法及工具

# JUnit 的基本功能

- 1.测试用例管理：** JUnit允许创建一系列测试用例，每个测试用例包含一个或多个测试方法
- 2.断言(assert)：** JUnit提供了多种断言，用于检查被测方法的返回值是否符合预期
- 3.异常检查：** JUnit可以检查被测方法是否会抛出预期的异常
- 4.测试流程管理：** JUnit提供了标记法来定义测试流程管理，例如 @Before 和 @After 用于在测试方法执行前后执行特定的操作

# Junit 使用方法

- 为 BankAccount 类编写Junit测试
- 测试类也是一个类：通常为被测试的类名 + Test
  - BankAccountTest
- 测试方法通常和待测方法同名，带有@Test标记
  - 带有@Test标记的方法可以直接执行
  - 可以多个测试方法，相互独立
- setUp() 方法中创建了一个新的 BankAccount 实例
  - 使用@Before标记，在任意的@Test标记方法运行前执行
- tearDown善后操作
  - 使用@After标记，释放资源和善后处理，所有的@Test方法运行后运行

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class BankAccountTest {
 private BankAccount ba;

 @Before
 public void setUp() {
 ba = new BankAccount("1234567890");
 }

 @Test
 public void deposit(){
 ba.deposit(50);
 assertEquals(ba.getBalance(),50);
 }

 @After
 public void tearDown() {
 System.err.println("Test pass!");
 }
}
```

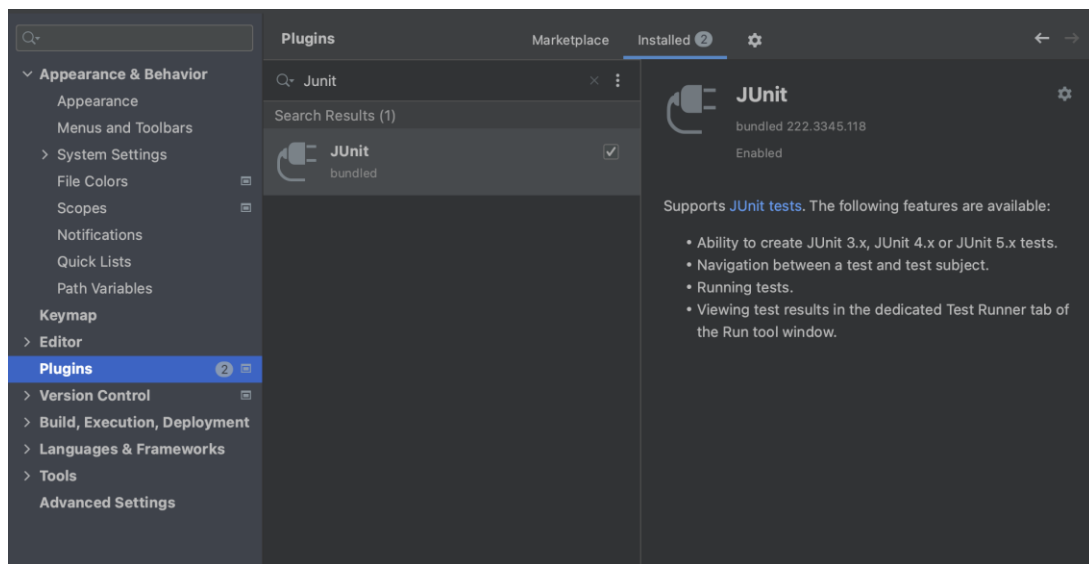
# Junit 使用方法

- 如果不能保证测试方法的独立性  
将会产生错误的测试结果
  - withdraw测试方法有什么问题?
  - 如何修改
- 编写综合的测试
  - 调用多个被测方法进行综合测试
- Junit提供了多种assert断言

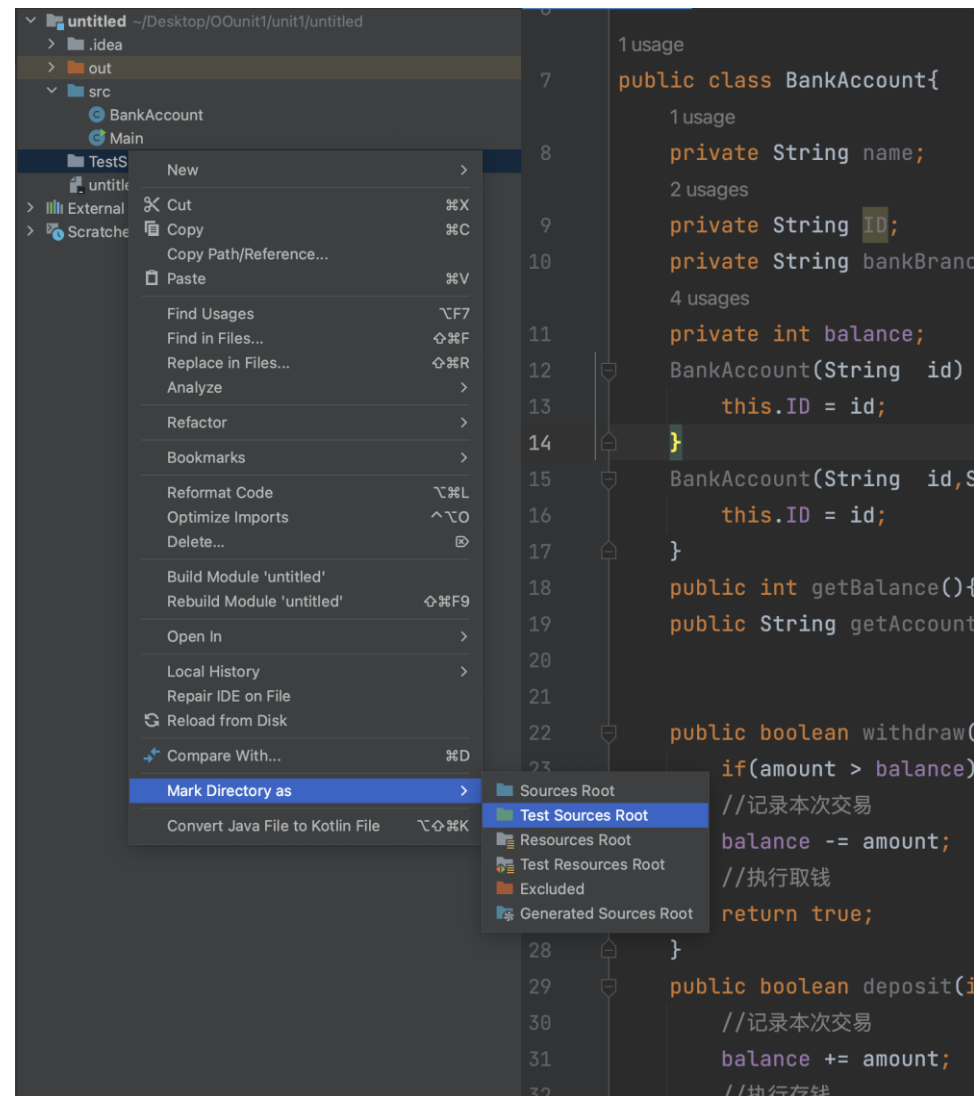
|             |                                                                                                                                                                                                     |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| static void | <code>assertEquals(double expected, double actual)</code><br><b>Deprecated. Use <code>assertEquals(double expected, double actual, double delta)</code> instead</b>                                 |
| static void | <code>assertEquals(double expected, double actual, double delta)</code><br>Asserts that two doubles are equal to within a positive delta.                                                           |
| static void | <code>assertEquals(float expected, float actual, float delta)</code><br>Asserts that two floats are equal to within a positive delta.                                                               |
| static void | <code>assertEquals(long expected, long actual)</code><br>Asserts that two longs are equal.                                                                                                          |
| static void | <code>assertEquals(Object[] expecteds, Object[] actuals)</code><br><b>Deprecated. use <code>assertArrayEquals</code></b>                                                                            |
| static void | <code>assertEquals(Object expected, Object actual)</code><br>Asserts that two objects are equal.                                                                                                    |
| static void | <code>assertEquals(String message, double expected, double actual)</code><br><b>Deprecated. Use <code>assertEquals(String message, double expected, double actual, double delta)</code> instead</b> |
| static void | <code>assertEquals(String message, double expected, double actual, double delta)</code><br>Asserts that two doubles are equal to within a positive delta.                                           |
| static void | <code>assertEquals(String message, float expected, float actual, float delta)</code><br>Asserts that two floats are equal to within a positive delta.                                               |
| static void | <code>assertEquals(String message, long expected, long actual)</code><br>Asserts that two longs are equal.                                                                                          |
| static void | <code>assertEquals(String message, Object[] expecteds, Object[] actuals)</code><br><b>Deprecated. use <code>assertArrayEquals</code></b>                                                            |
| static void | <code>assertEquals(String message, Object expected, Object actual)</code><br>Asserts that two objects are equal.                                                                                    |
| static void | <code>assertFalse(boolean condition)</code><br>Asserts that a condition is false.                                                                                                                   |
| static void | <code>assertFalse(String message, boolean condition)</code><br>Asserts that a condition is false.                                                                                                   |
| static void | <code>assertNotEquals(double unexpected, double actual, double delta)</code><br>Asserts that two doubles are <b>not</b> equal to within a positive delta.                                           |
| static void | <code>assertNotEquals(float unexpected, float actual, float delta)</code><br>Asserts that two floats are <b>not</b> equal to within a positive delta.                                               |
| static void | <code>assertNotEquals(long unexpected, long actual)</code><br>Asserts that two longs are <b>not</b> equals.                                                                                         |
| static void | <code>assertNotEquals(Object unexpected, Object actual)</code><br>Asserts that two objects are <b>not</b> equals.                                                                                   |
| static void | <code>assertNotEquals(String message, double unexpected, double actual, double delta)</code><br>Asserts that two doubles are <b>not</b> equal to within a positive delta.                           |
| static void | <code>assertNotEquals(String message, float unexpected, float actual, float delta)</code><br>Asserts that two floats are <b>not</b> equal to within a positive delta.                               |

# 在IDEA中使用JUnit

## 1.在IDEA中安装JUnit插件（一般默认内置）

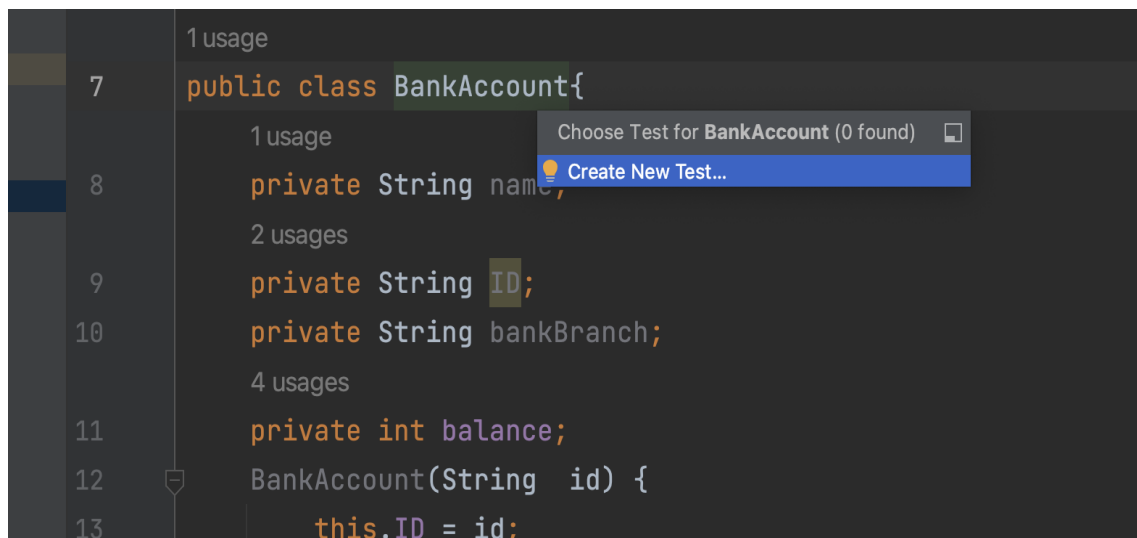


## 2.创建测试文件夹(存储单元测试文件)

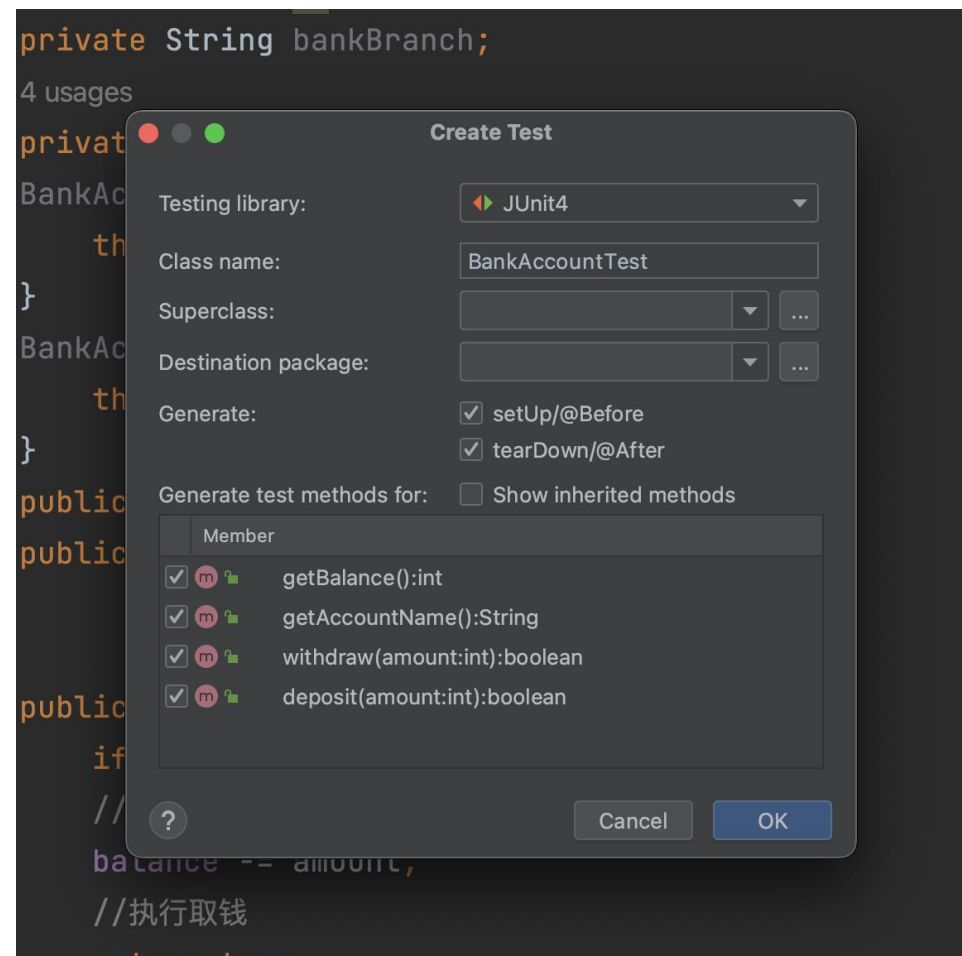


# 在IDEA中使用JUnit

3.ctrl+shift+t选中要测试的类，创建单元测试



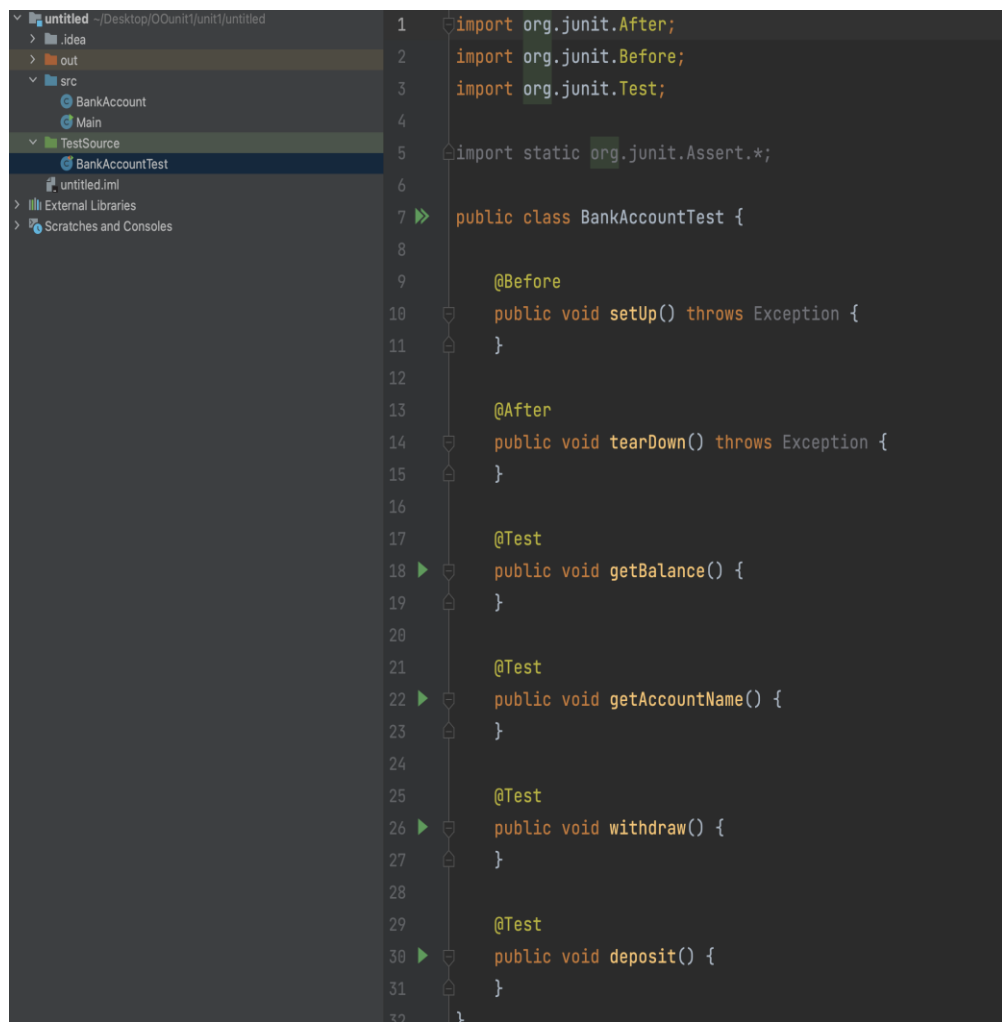
4.勾选测试设置，选择是否自动生成Before、After函数，选择要测试的函数，点击确认会自动生成测试文件





# 在IDEA中使用JUnit

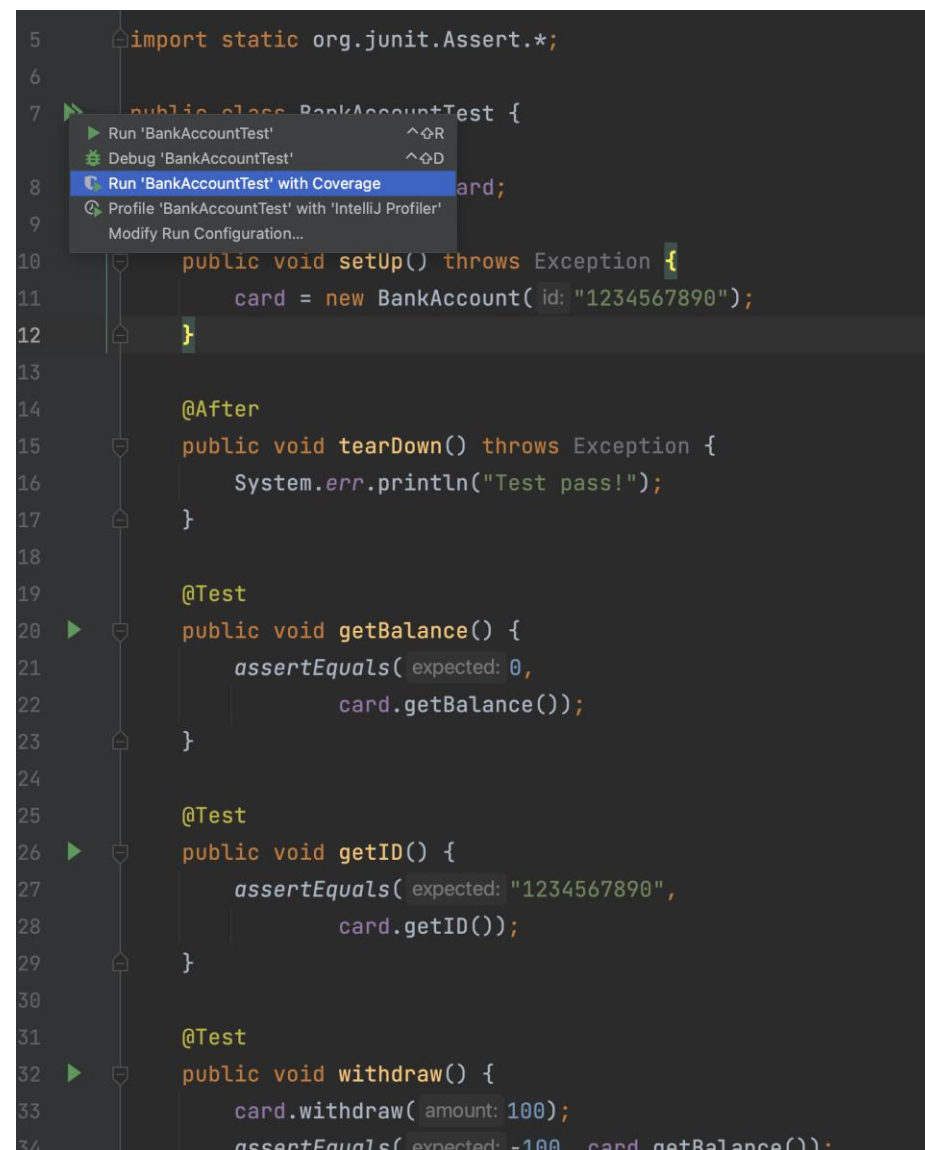
5. 可以看到测试文件已经自动生成，需要补全测试代码



The screenshot shows the IntelliJ IDEA interface. On the left, the 'Project' tool window displays the project structure: 'untitled' (root), '.idea', 'out', 'src' (containing 'BankAccount' and 'Main'), 'TestSource' (containing 'BankAccountTest'), 'untitled.iml', 'External Libraries', and 'Scratches and Consoles'. The 'BankAccountTest.java' file is selected in the 'TestSource' folder. The main editor shows the code for 'BankAccountTest.java':

```
1 import org.junit.After;
2 import org.junit.Before;
3 import org.junit.Test;
4
5 import static org.junit.Assert.*;
6
7 public class BankAccountTest {
8
9 @Before
10 public void setUp() throws Exception {
11 }
12
13 @After
14 public void tearDown() throws Exception {
15 }
16
17 @Test
18 public void getBalance() {
19 }
20
21 @Test
22 public void getAccountName() {
23 }
24
25 @Test
26 public void withdraw() {
27 }
28
29 @Test
30 public void deposit() {
31 }
32 }
```

6. 手动补全自己想进行的测试代码后，点击使用覆盖率运行



The screenshot shows the IntelliJ IDEA interface with the 'BankAccountTest.java' file open. A context menu is visible over the code, with the option 'Run BankAccountTest with Coverage' highlighted. The code in the background is as follows:

```
5 import static org.junit.Assert.*;
6
7 public class BankAccountTest {
8 @Before
9 public void setUp() throws Exception {
10 card = new BankAccount(id: "1234567890");
11 }
12
13 @After
14 public void tearDown() throws Exception {
15 System.err.println("Test pass!");
16 }
17
18 @Test
19 public void getBalance() {
20 assertEquals(expected: 0,
21 card.getBalance());
22 }
23
24 @Test
25 public void getID() {
26 assertEquals(expected: "1234567890",
27 card.getID());
28 }
29
30 @Test
31 public void withdraw() {
32 card.withdraw(amount: 100);
33 assertEquals(expected: -100, card.getBalance());
34 }
35 }
```

# 在IDE中查看单元测试覆盖率

The screenshot displays the IntelliJ IDEA IDE with the `BankAccountTest.java` file open. The `src` directory shows 50% classes and 57% lines covered. The `TestSource` directory shows 83% methods and 61% lines covered for `BankAccountTest`. The `BankAccount` class has 100% methods and 61% lines covered. The `Main` class has 0% methods and 0% lines covered.

The Coverage tool window (top right) shows the following data:

| Element     | Class, %   | Method, %   | Line, %     |
|-------------|------------|-------------|-------------|
| all         | 50% (2/4)  | 71% (10/14) | 57% (16/28) |
| BankAccount | 100% (1/1) | 83% (5/6)   | 61% (8/13)  |
| Main        | 0% (0/1)   | 0% (0/1)    | 0% (0/1)    |

The test results window (bottom) shows the following data:

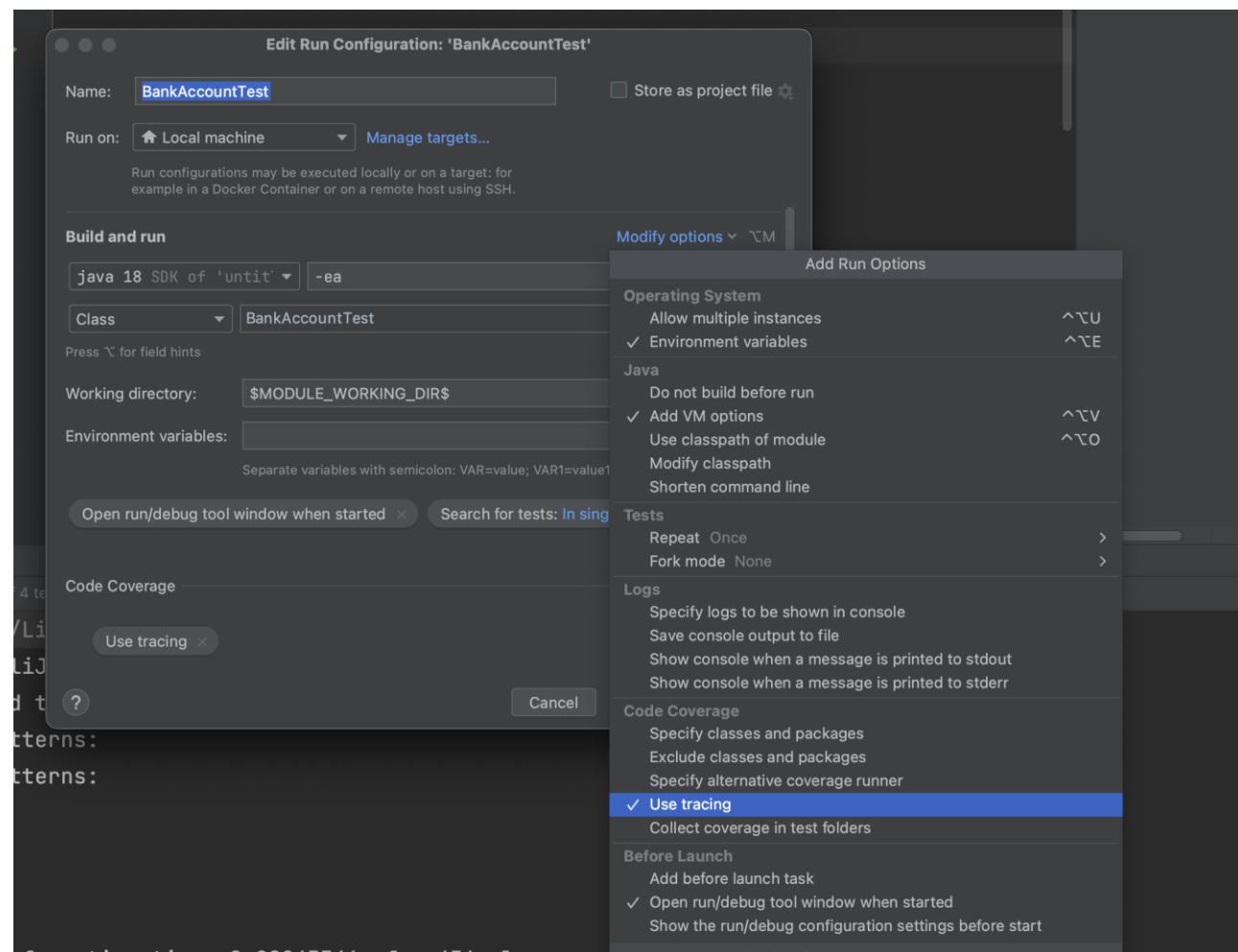
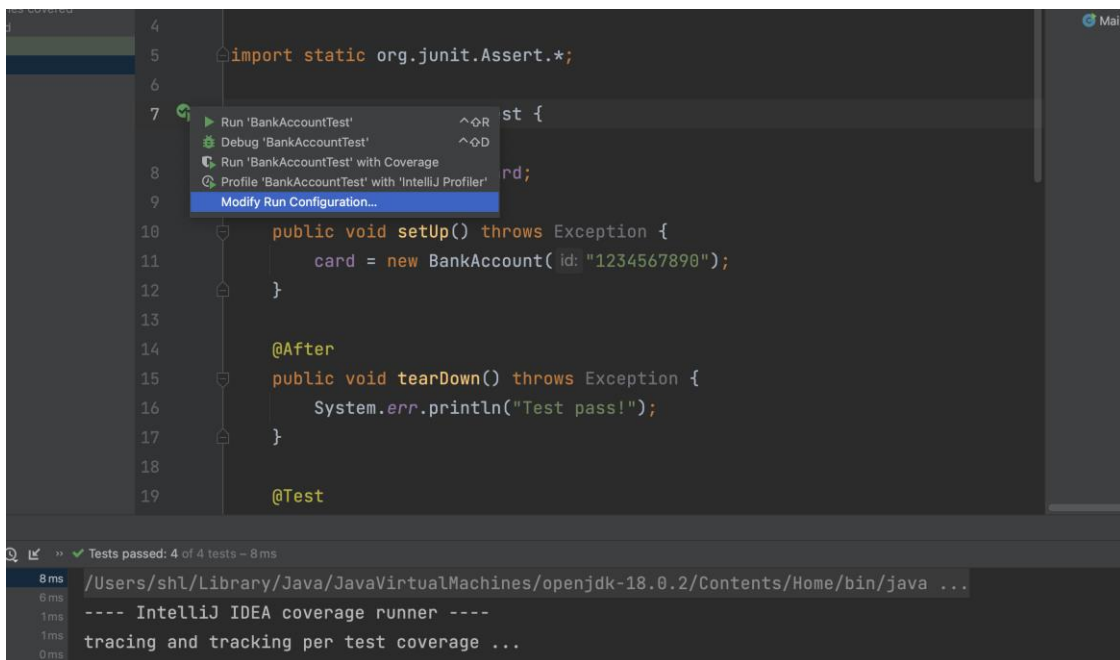
| Test            | Time |
|-----------------|------|
| BankAccountTest | 5 ms |
| withdraw        | 4 ms |
| getID           | 1 ms |
| getBalance      | 0 ms |
| deposit         | 0 ms |

The test results window also shows the following output:

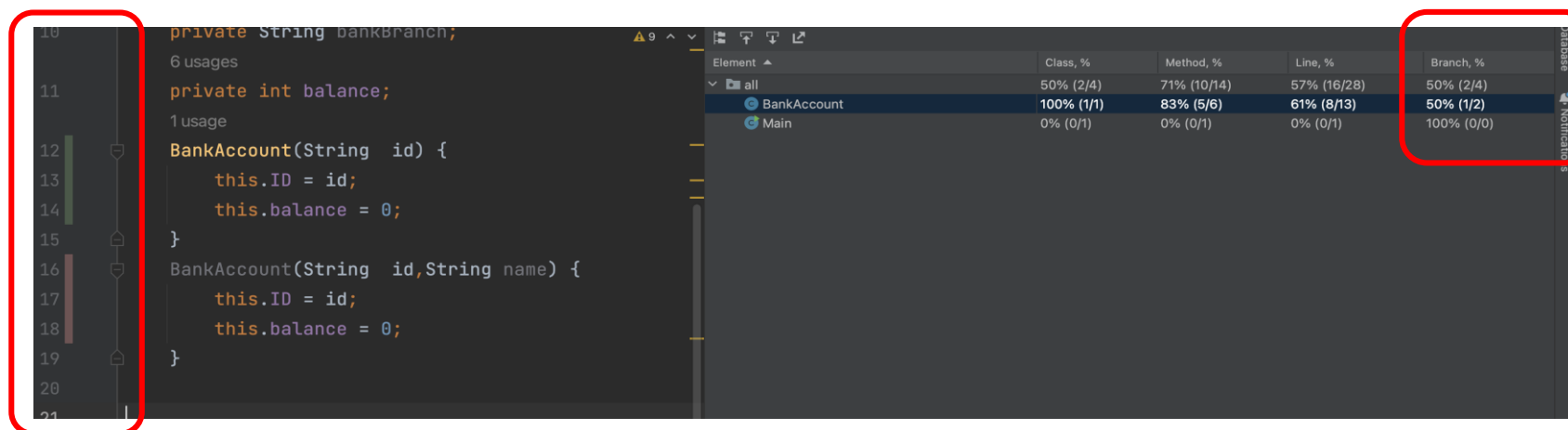
```
Tests passed: 4 of 4 tests - 5 ms
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
exclude patterns:
Test pass!
Test pass!
```

# 在IDE中添加分支覆盖率

7. 设置单元测试配置，勾选Use tracing，重新运行单元测试



# 在IDE中添加单元测试分支覆盖率



分支覆盖率

- ✓ 函数覆盖率：定义的函数中有多少被调用
- ✓ 语句覆盖率：程序中的语句有多少被执行
- ✓ 分支覆盖率：有多少控制结构的分支（例如if语句）被执行（常用）
- ✓ 条件覆盖率：有多少布尔子表达式被测试为真值和假值
- ✓ 行覆盖率：有多少行源代码被测试过

# 本周作业介绍

- 迭代开发是主流的开发方式
  - 需求持续变化，设计持续改进，测试始终护航！
- 熟悉风格检查+中测+强测的玩法了吗？
- 一个穿越到魔法大陆上的冒险者，在旅途中收集各种道具，使用各种装备，招募其他冒险者加入队伍，提升作战水平和经验
- 本次作业实现冒险者，管理好冒险者持有的装备
  - 编写类和创建对象，使用基本容器来管理对象
  - 提供了基础的输入处理代码String[]→ArrayList<String>
- 编写JUnit测试并运行（按照指导书的要求来编写！）