Chapter 6 Objects and Classes

OO Programming Concepts

Object-oriented programming (OOP) involves ***programming using objects***. An ***object*** represents an ***entity in the real world*** that can be distinctly identified. For example, a *student*, a *desk*, a *circle*, a *button*, and even a loan can all be viewed as objects. An object has a unique identity, ***state***, and ***behaviors***. The ***state*** of an object consists of a set of ***data fields*** (also known as ***properties***) with their current values. The ***behavior*** of an object is defined by a set of ***methods***.

Objects

An object has both a *state* and *behavior*. The state *defines* the object, and the behavior defines *what the object does*.

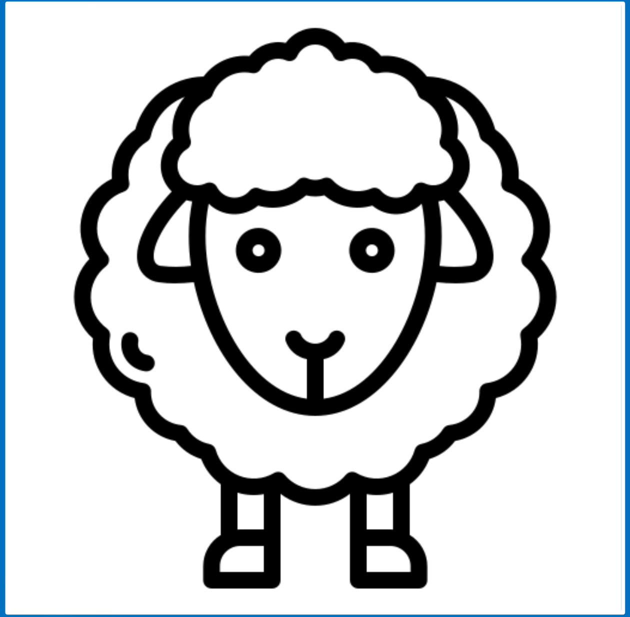
Object: a *car*

state: red, seats, etc.

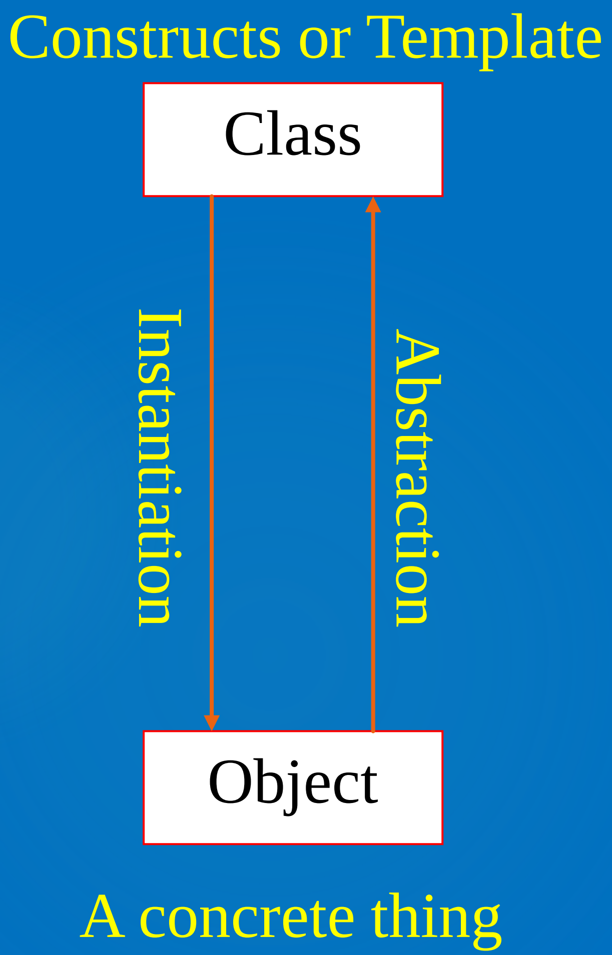
behavior: driving and running.

Classes

Classes are ***constructs*** that define objects of the ***same type***. A Java class uses ***variables*** to define ***data fields*** and ***methods*** to define ***behaviors***. Additionally, a class provides a special type of methods, known as ***constructors***, which are invoked to construct objects from the class.



Classes vs Objects



Class: Dog

Create Instance

Object: Jane

Properties	**Methods**
Color	*Sit*
Eye Color	*Lay Down*
Height	*Shake*
Length	
Weight	

Properties	**Methods**
Color: White and Black	*Sit*
Eye Color: Brown	*Lay Down*
Height: 40cm	*Shake*
Length: 90cm	
Weight: 20kg	

Class Structure in Java

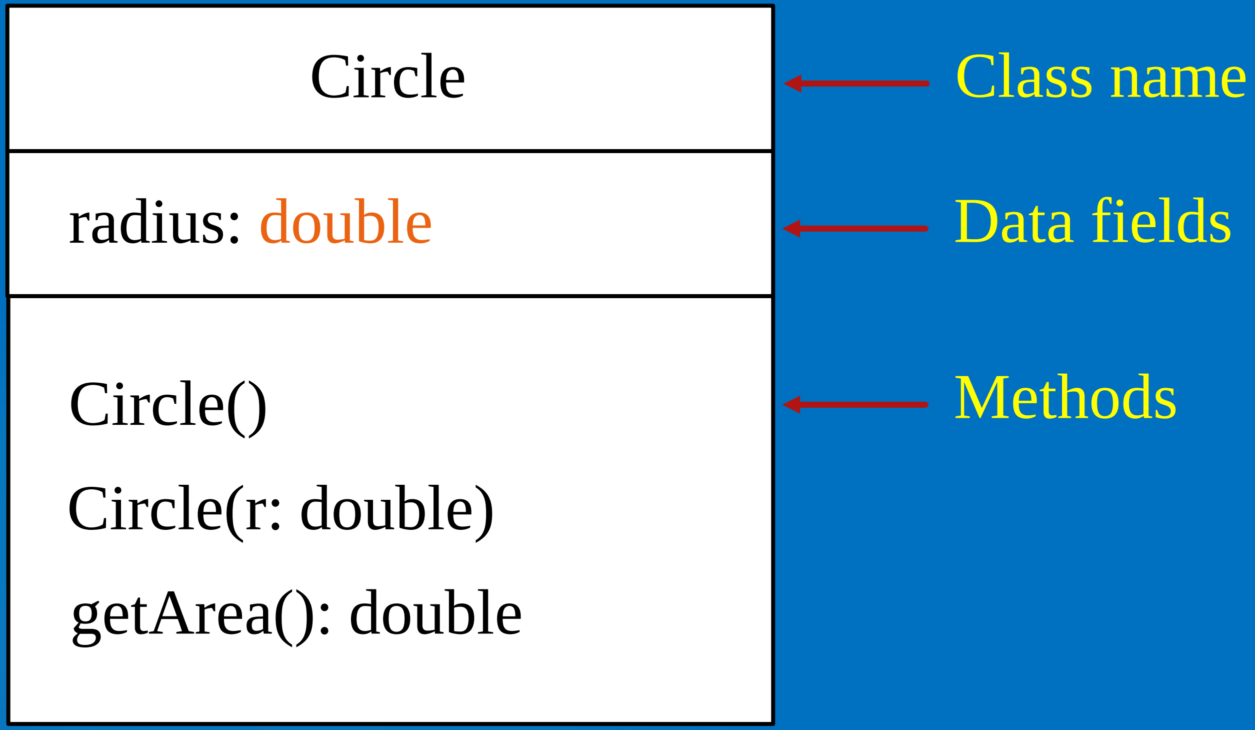
```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;
  /** Construct a circle object */
  Circle() {
  }
  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }
  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}
```

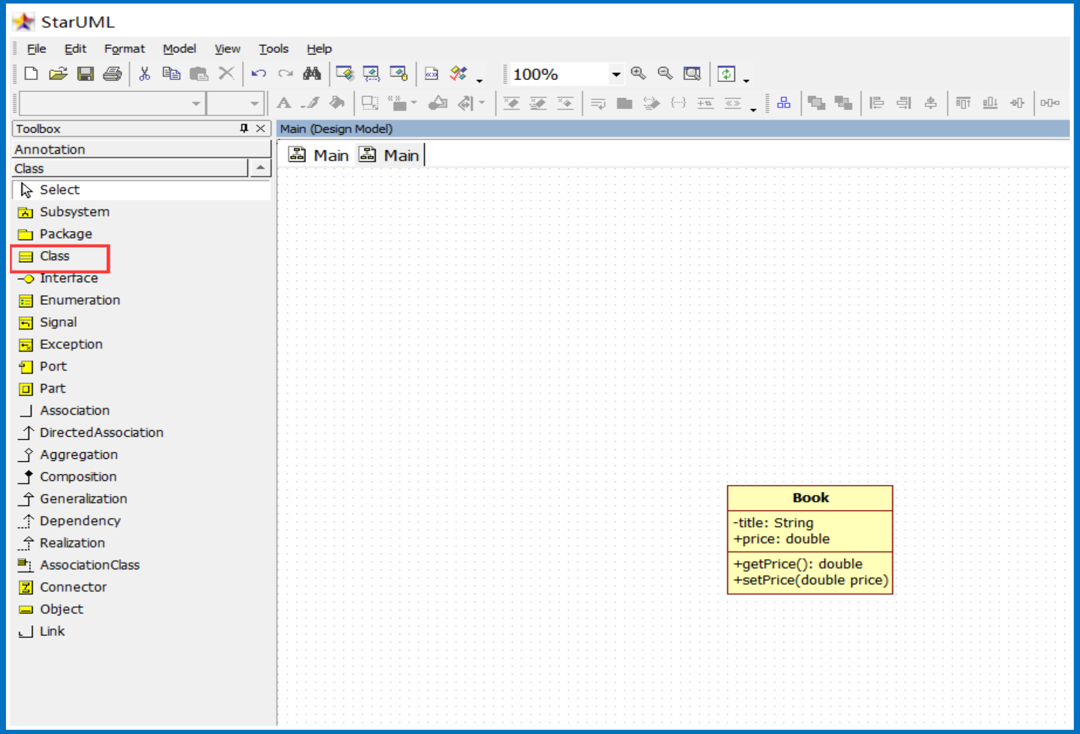
Data fields

Constructors

Method

Class UML Diagram





Draw Class Diagram Using StarUML

Constructor

Constructors are a special kind of methods that are invoked to *construct* objects.

- Constructors must have the *same name* as the class itself.
- Constructors *do not have a return type—not even void*.
- Constructors are *invoked* using the *new* operator when an object is created. Constructors play the role of initializing objects.

```java
// Constructors for constructing circles
public Circle() {}
public Circle(double radius) {
  this.radius = radius;
}
```

Default Constructor

A class may be declared *without* constructors. In this case, a no-arg *constructor* with an *empty body* is implicitly declared in the class. This constructor, called a *default constructor*, is provided automatically *only if no constructors* are *explicitly declared* in the class.

```
class Circle{
  double radius = 1.0;
  // Default constructor: public Circle(){}
  public double getArea() {
    return Math.PI * Math.pow(radius, 2);
  }
}
```

Constructor Overload

```
class Circle{
  double radius = 1.0;
  /* When other constructors are declared, the default no-arg
     constructor will not be implicitly declared
  */
  public Circle(){}
  // Overload Constructor
  public Circle(double radius){
    this.radius = radius;
  }
}
```

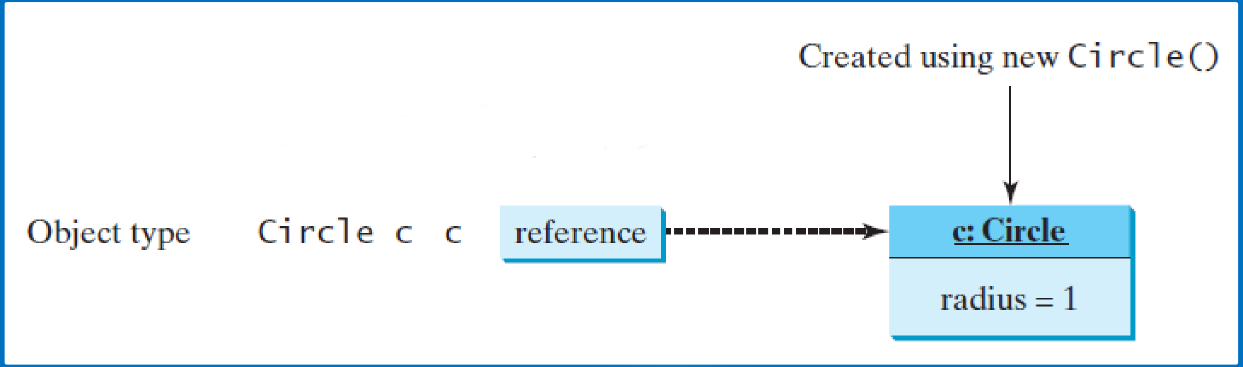
Example: Class Defining and Object Creating

```
public class Test{
  public static void main(String[] args){
    Circle c1 = new Circle(2.0);
    System.out.printf("The area of the circle is %4.2f", c1.getArea());
  }
}
class Circle{
  double radius = 1.0;
  public Circle() {}
  public Circle(double radius) {
    this.radius = radius;
  }
  public double getArea() {
    return Math.PI * Math.pow(radius, 2);
  }
}
```

Reference Variables and Reference Types

Objects are accessed via the object's *reference variables*, which contain references to the objects. [***Note1***: An *object reference variable* and an *object are different*, but most of the time the *distinction* can be *ignored*. Therefore, it is fine, for simplicity, to say that ***myCircle*** is a ***Circle*** object rather than use the longer-winded description that ***myCircle*** is a variable that contains a reference to a ***Circle*** object.] [***Note2***: Arrays are treated as objects in Java. Arrays are created using the new operator. An array variable is actually a variable that contains a reference to an array.]

```
Circle c = new Circle(1.0);
```



Accessing an Object’s Data and Methods

In OOP terminology, an object’s *member* refers to its data fields and methods. After an object is created, its data can be accessed and its methods can be invoked using the *dot operator (.)*, also known as the *object member access operator*:

```
Circle c = new Circle();
// Referencing the object’s data: objectRefVar.dataField
c.radius;
// Invoking the object’s methods: objectRefVar.methodName(arguments)
c.getArea();
```

How to access the members of an *anonymous object*.

```
System.out.println("Area is " + new Circle(5).getArea());
```

Default Values of Class Data fields

The *data fields* can be of *reference types*. If a data field of a reference type does not reference any object, the data field holds a special Java value, *null*. The *default value* of a data field is *null for a reference type*, *0 for a numeric type*, *false for a boolean type*, and *\u0000 for a char type*. However, Java assigns *no default value to a local variable* inside a method.

```
class Student {
  String name; // name has the default value null
  int age; // age has the default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // gender has default value '\u0000'
}
```

Class Members and Instance Members

A class may contain *static members* and *instance members*, and the static members are declared with the keyword *static*. There are some differences between *class members* and *instance members*:

(1) Static variables are *shared by all* the instances of the class while instance variables belong to a *specific* instance.
(2) Static variables are accessed by *class name* while instance variables are accessed by the reference variable of the object.
(3) A *static method* *cannot* access *instance members* of the class while a instance method can access both static members and instance members.
(4) *Static constants* are *final variables shared by all* the instances of the class.

Example: Countable Class

```
public class Client {
  public static void main(String[] x) {
    Circle c1 = new Circle(1.0);
    Circle c2 = new Circle(2.0);
    System.out.println(c1.radius); // 1.0
    System.out.println(c2.radius); // 2.0
    System.out.println(Circle.numberOfCircle); // 2
    System.out.println(Circle.numberOfCircle); // 2
  }
}
class Circle{
  double radius; // instance variable
  static int numberOfCircle; // class variable
  final static double PI = 3.14; // class constant
  public Circle(double r){ numberOfCircle++; radius = r;}
}
```

Example: The Math Class

```
// The source code of the Math class
public final class Math {
  public static final double E = 2.7182818284590452354;
  public static final double PI = 3.14159265358979323846;
  public static long abs(long a) {
    return (a < 0) ? -a : a;
  }
}
// The use of the Math class
public static void main(String[] x) {
  System.out.println(Math.E); //2.718281828459045
  System.out.println(Math.PI); //3.141592653589793
  System.out.println(Math.abs(-90)); //90
}
```

Example: Class methods can only access class variables

```
class Circle{
  double radius;
  static int numberOfCircle;
  final static double PI = 3.14;
  public Circle(double r){ numberOfCircle++; radius = r;}
  public static int getNumberOfCircle(){
    return numberOfCircle; // Access a class variable (Correct)
  }
  public static int getR(){
    return radius; // Access a instance variable (Error)
  }
}
```

Exercises

What is wrong with each of the following programs?

```
public class ShowErrors {
  public static main(String[] args) {
    ShowErrors t = new ShowErrors(5);
  }
}
```

```
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors();
    t.x();
  }
}
```

```
public class ShowErrors {
  public static void main(String[] args) {
    C c = new C(5.0);
    System.out.println(c.value);
  }
}

class C {
  int value = 2;
}
```

Exercises

What is wrong in the following code?

```
class Test {
  public static void main(String[] args) {
    A a = new A();
    a.print();
  }
}
class A {
  String s;
  A(String newS) {
    s = newS;
  }
  public void print() {
    System.out.print(s);
  }
}
```

Exercises

What is the output of the following code?

```java
public class A {
  boolean x;

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

Exercises

What is the output of the following code?

```
public class MyObject {
  public static int x = 7;
  public int y = 3;
  public static void main(String[] args) {
    MyObject a = new MyObject();
    MyObject b = new MyObject();
    a.y = 5; b.y = 6; a.x = 1; b.x = 2;
    System.out.printf("a.y=%d,b.y=%d,a.x=%d,b.x=%d\n",a.y,b.y,a.x,b.x);
    System.out.println(MyObject.x);
  }
}
```

Exercises

Suppose that the class **F** is defined in (a). Let **f** be an instance of **F**. Which of the statements in (b) are correct?

```
public class F {
  int i;
  static String s;
  void imethod() {
  }
  static void smethod() {
  }
}
```

(a)

```
System.out.println(f.i);
System.out.println(f.s);
f.imethod();
f.smethod();
System.out.println(F.i);
System.out.println(F.s);
F.imethod();
F.smethod();
```

(b)

Exercises

Add the static keyword in the place of ? if appropriate.

```
public class Test {
  int count;
  public (1)? void main(String[] args) {

  }
  public (2)? int getCount() {
    return count;
  }
  public (3)? int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
    result *= i;
    return result;
  }
}
```

Exercises

Can you invoke an instance method or reference an instance variable from a static method? Can you invoke a static method or reference a static variable from an instance method? What is wrong in the following code?

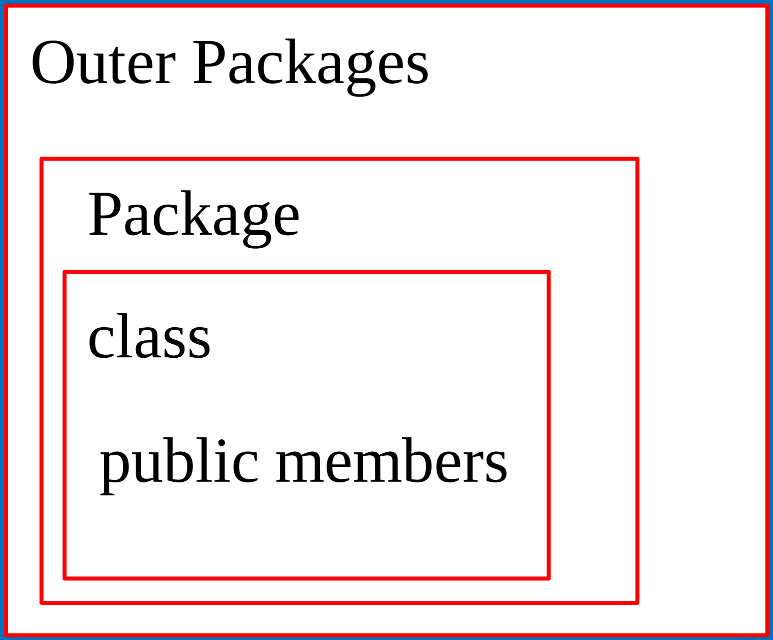
```
public class C {
  public static void main(String[] args) {
    method1();
  }
  public void method1() {
    method2();
  }
  public static void method2() {
    System.out.println("What is radius " + c.getRadius());
  }
}
```

Visibility Modifiers

Visibility modifiers can be used to specify the visibility of a *class* and its *members*. The modifiers *public*, *protected*, and *private* are placed in front of each definition for each member in your class, whether it’s a *field* or a *method*. Each access specifier only controls the access for that particular definition.

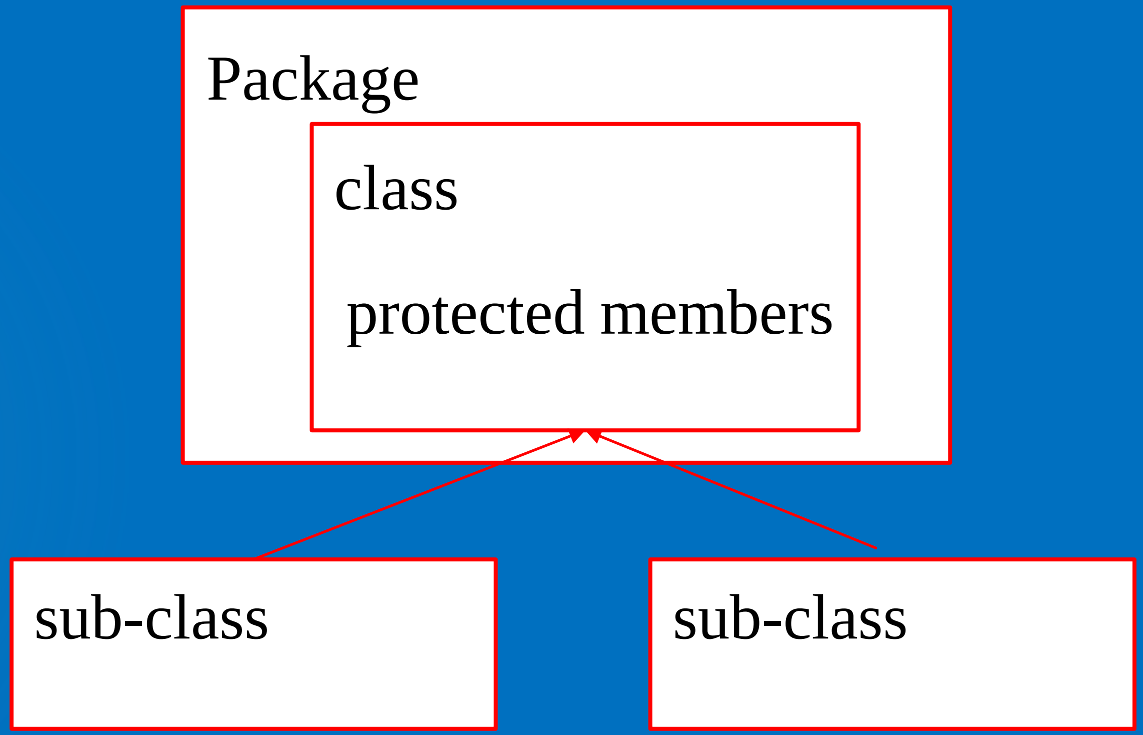
Visibility Modifiers

Public: The *public* keyword is an visibility modifier used for *classes*, *attributes*, *methods* and *constructors*, making them accessible by *any other class*.



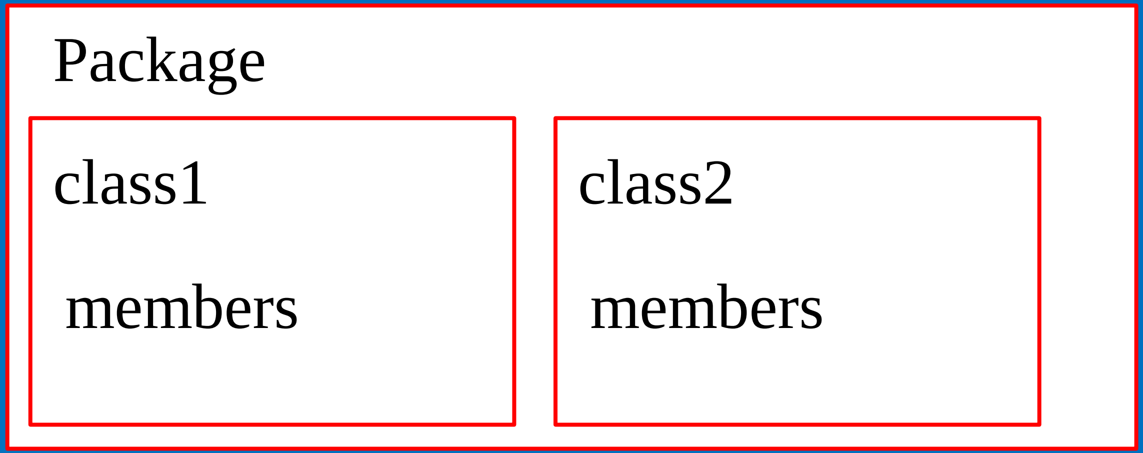
Visibility Modifiers

Protected: The *protected* keyword is an visibility modifier used for *attributes*, *methods* and *constructors*, making them accessible in the *same package* and *subclasses*.



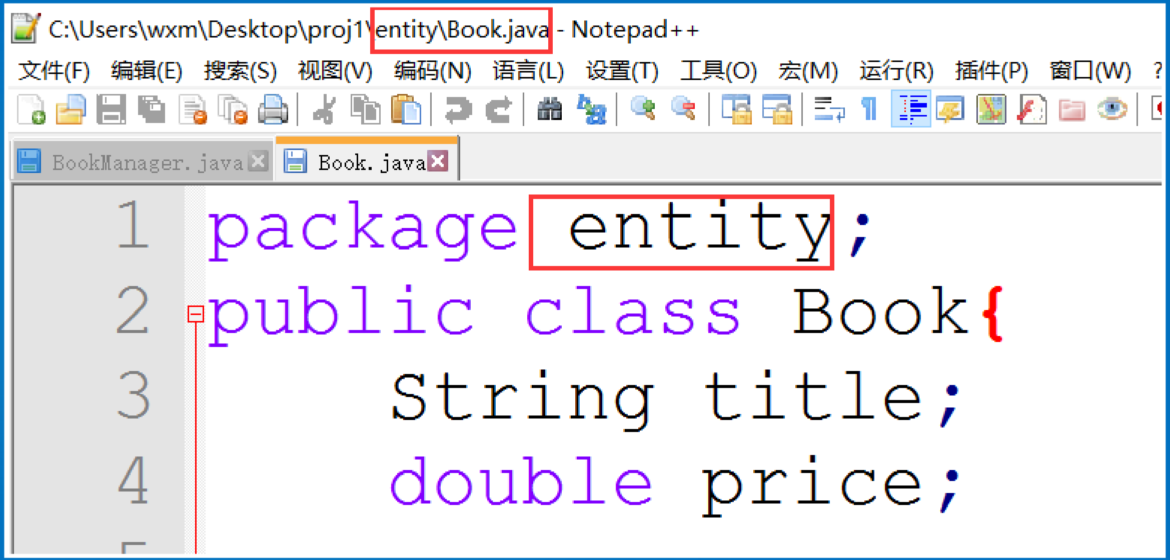
Visibility Modifiers

Default: Package access allows you to group related classes together in a *package* so that they can easily interact with each other. Package access is a *default* setting when there is no modifier before members.



Supplementary: Package and Import

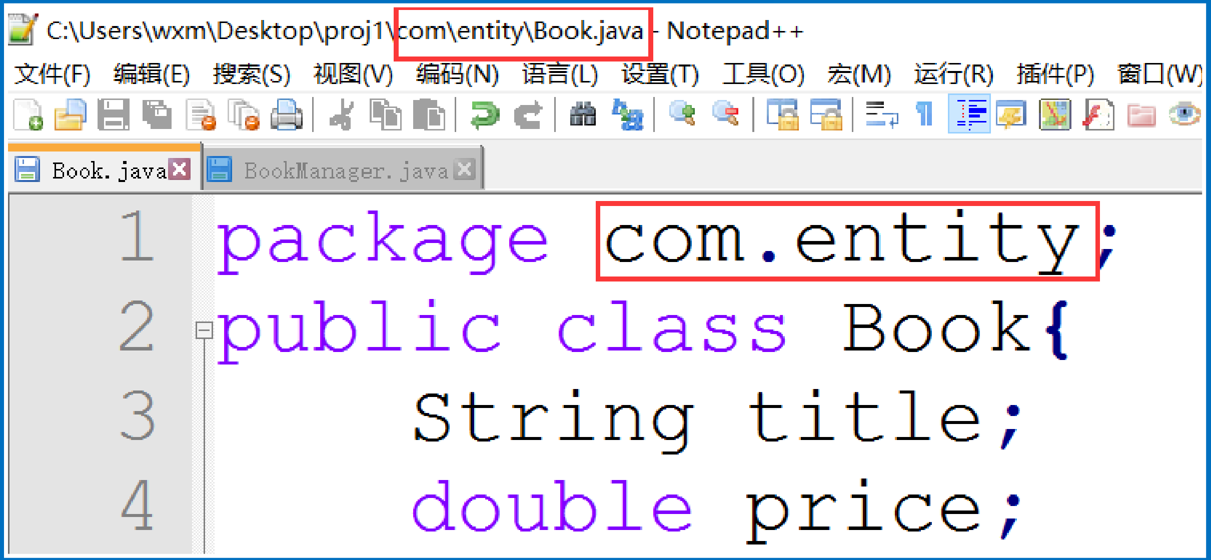
A package contains a group of classes, organized together under a single *namespace*. The syntax is like "`package pkg1[. pkg2[. pkg3…]];`". The path of class file and the package name must be consistent.



C:\Users\wxm\Desktop\proj1\entity\Book.java - Notepad++

文件(F) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(T) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W) ?
BookManager.java
Book.java

```
package entity;
public class Book{
    String title;
    double price;
```



C:\Users\wxm\Desktop\proj1\com\entity\Book.java - Notepad++

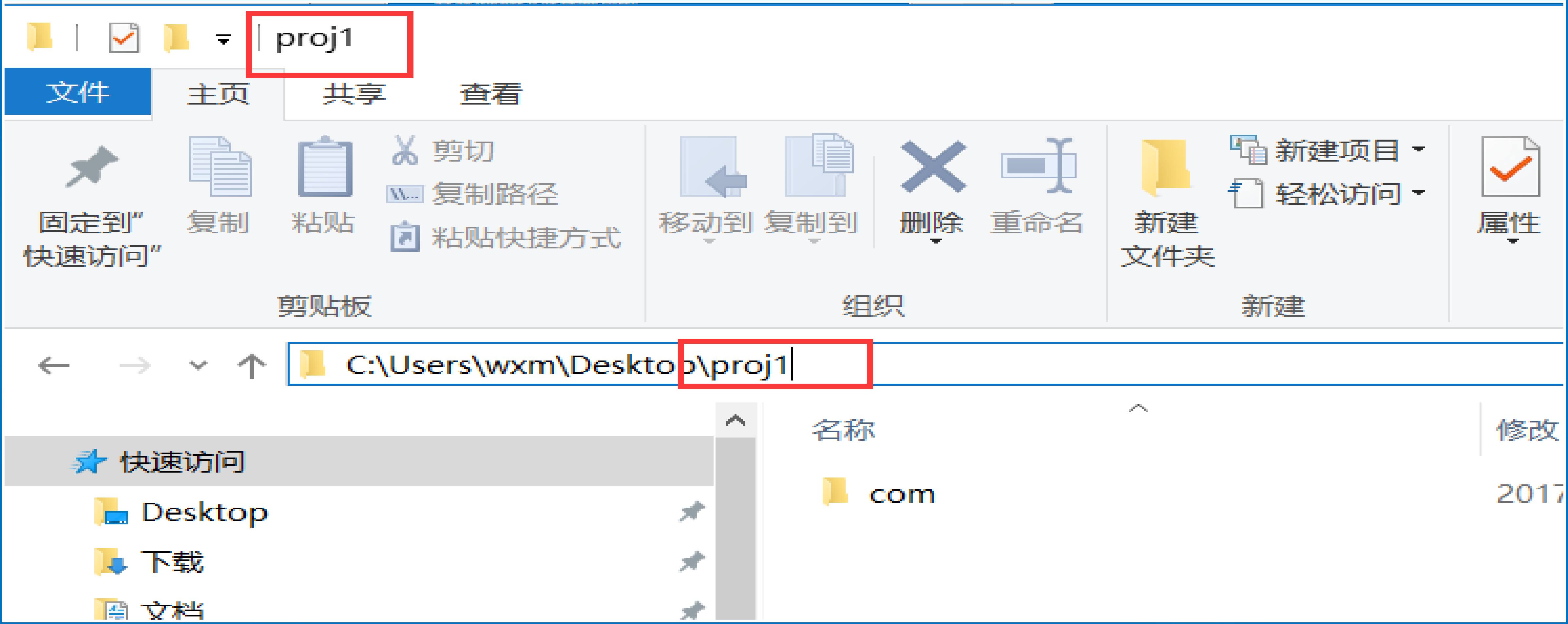
文件(F) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(T) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W)
Book.java
BookManager.java

```
package com.entity;
public class Book{
    String title;
    double price;
```

Supplementary: Package and Import

How to compile and run java program when it is organized with package:

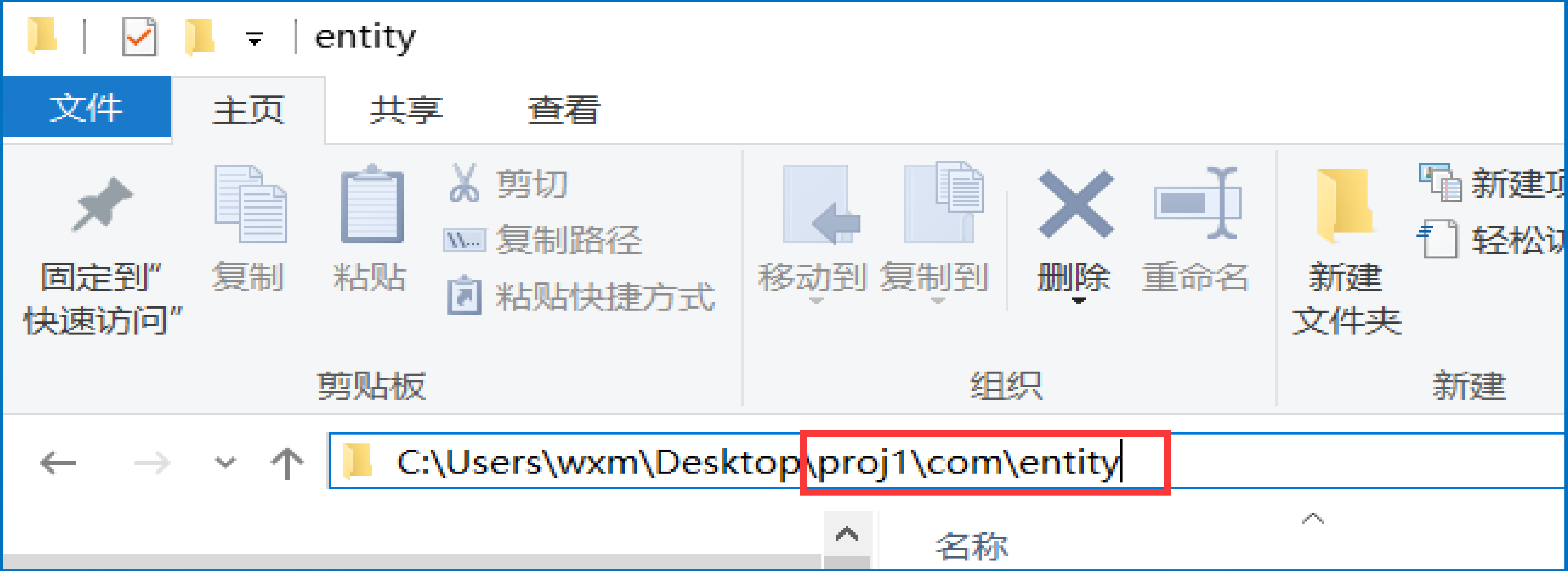
1. Create a folder as the project root directory



C:\Users\wxm\Desktop\proj1

Supplementary: Package and Import

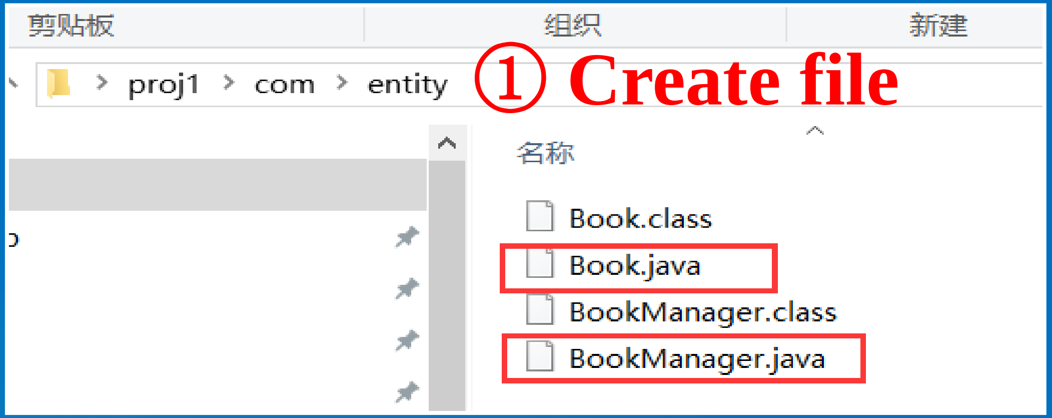
2. Create folders as package directories. For example, com/entity is created for organizing entity classes.

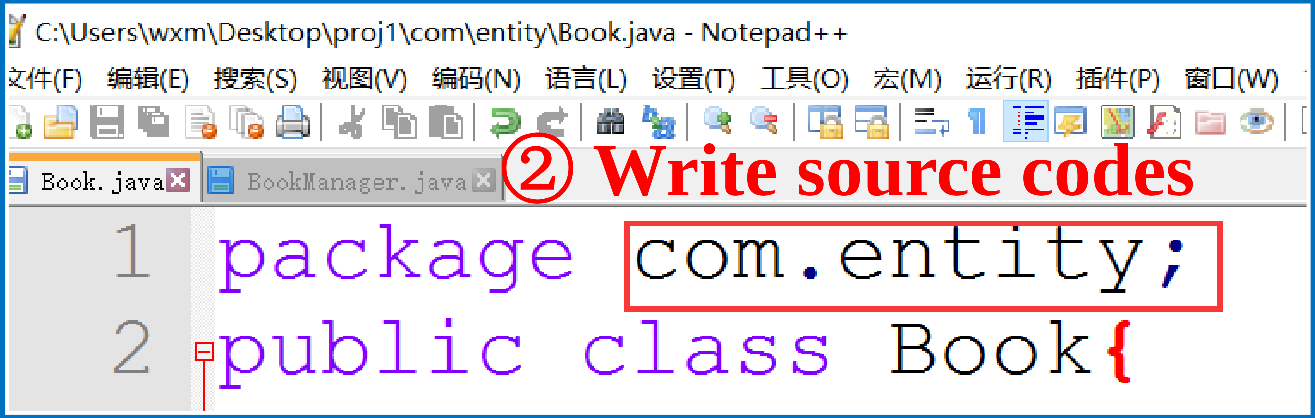


C:\Users\wxm\Desktop\proj1\com\entity

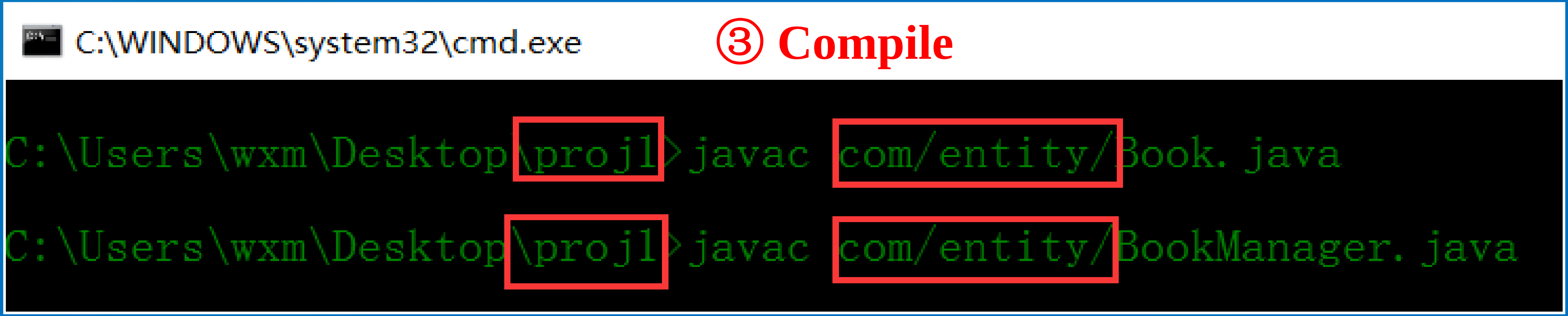
Supplementary: Package and Import

3. Create source files and compile them.




C:\Users\wxm\Desktop\proj1\com\entity\Book.java - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(T) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W)
Book.java
BookManager.java
② Write source codes

```
package com.entity;
public class Book{
```

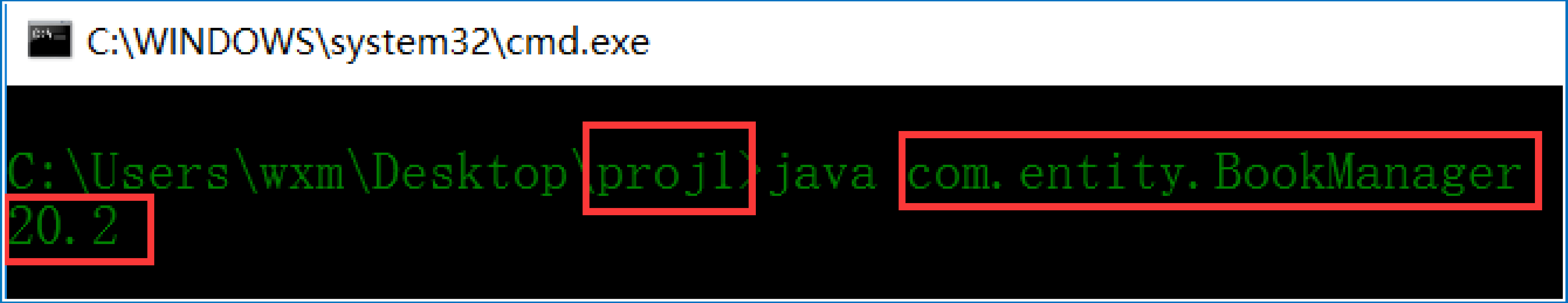


```
C:\Users\wxm\Desktop\proj1>javac com/entity/Book.java

C:\Users\wxm\Desktop\proj1>javac com/entity/BookManager.java
```

Supplementary: Package and Import

4. Run the program if there exists a class with a main method.

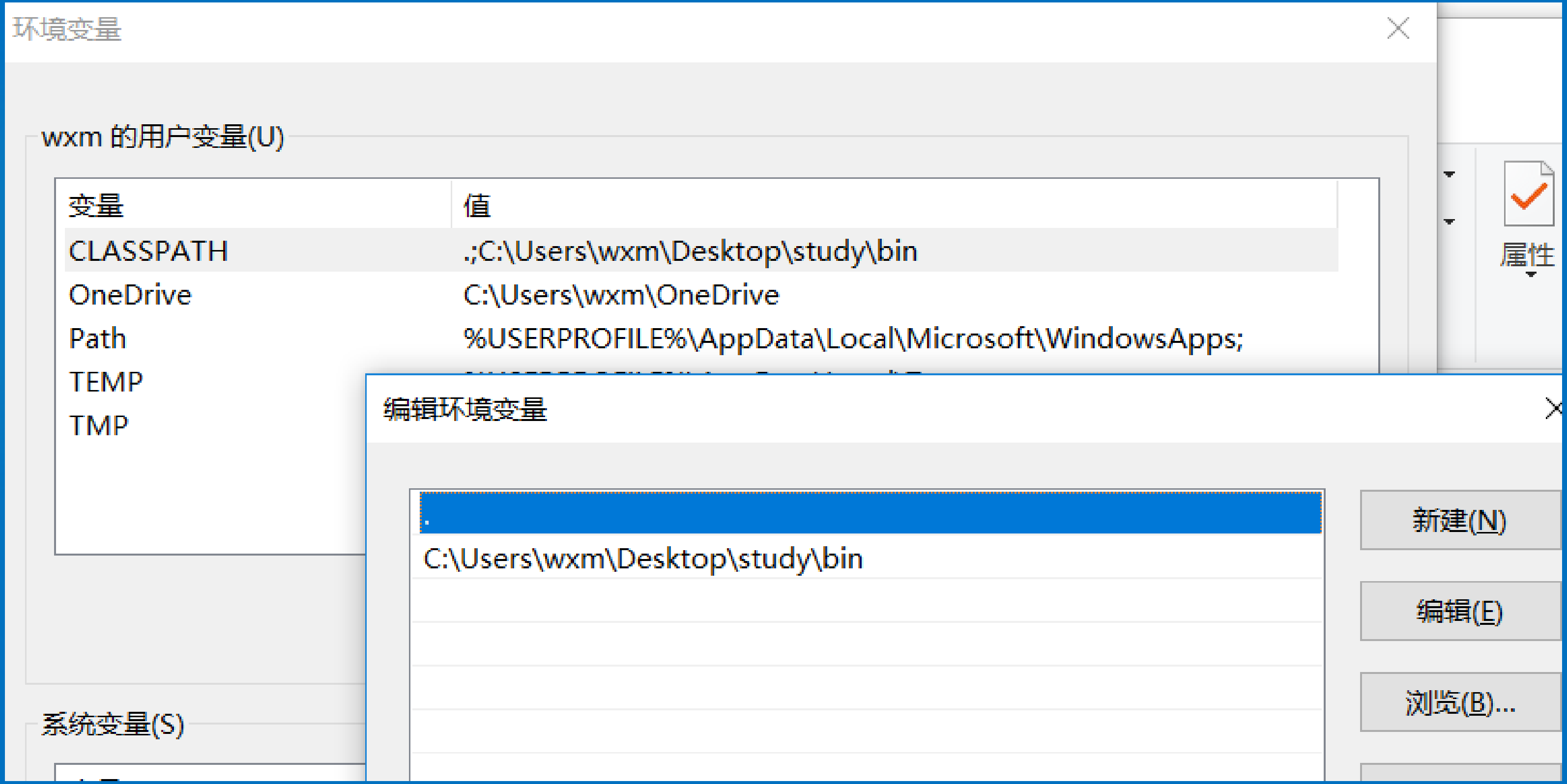


```
C:\Users\wxm\Desktop\proj1>java com.entity.BookManager
20.2
```

Supplementary: Package and Import

Compile the source file to the specified directory

```
javac -d ../bin com/entity/Book.java
java com.entity.Book
```


环境变量

wxm 的用户变量(U)

变量	值
CLASSPATH	.;C:\Users\wxm\Desktop\study\bin
OneDrive	C:\Users\wxm\OneDrive
Path	%USERPROFILE%\AppData\Local\Microsoft\WindowsApps;
TEMP	
TMP	

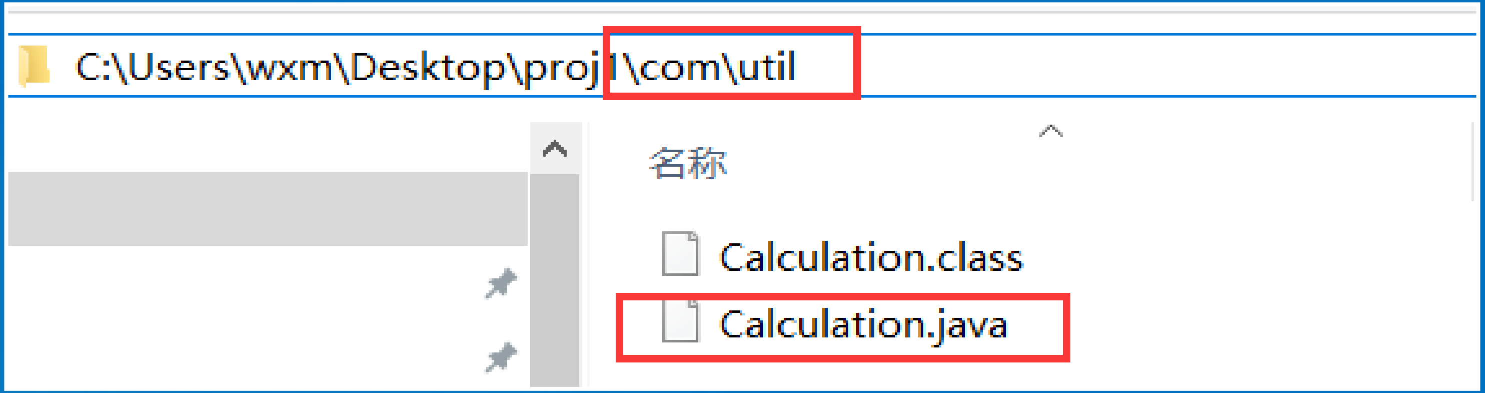
系统变量(S)

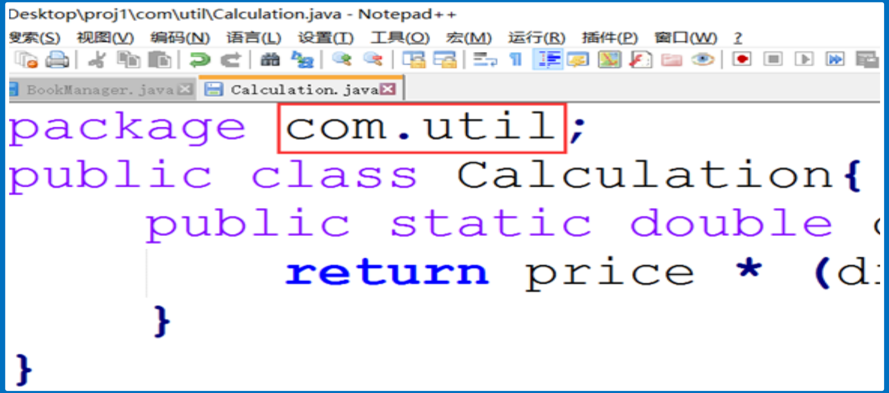
编辑环境变量

```
.
C:\Users\wxm\Desktop\study\bin
```

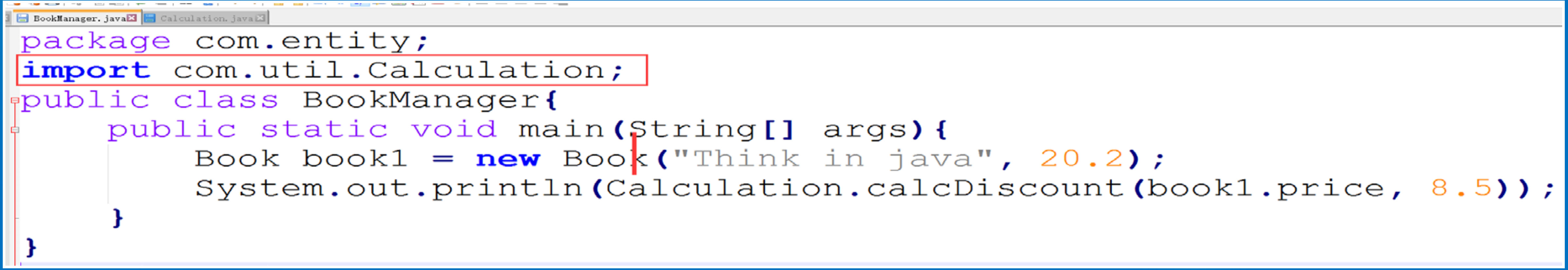
Supplementary: Package and Import

What if class A need class B but they are not in the same package. For example, package *com.util* is created for organizing tool classes. Use import keyword to use the Calculation class in the BookManager class.





```
package com.util;
public class Calculation{
    public static double c
        return price * (d
    }
}
```



```
package com.entity;
import com.util.Calculation;
public class BookManager{
    public static void main(String[] args){
        Book book1 = new Book("Think in java", 20.2);
        System.out.println(Calculation.calcDiscount(book1.price, 8.5));
    }
}
```

Visibility Modifiers

Private: The *private* keyword is an visibility modifier used for *attributes*, *methods* and *constructors*, making them only accessible *within the declared class*.

class

private members

Examples

The **private** modifier makes methods and data fields accessible only from within its own class. The following codes illustrate how a public, default, and private data field or method in class **C1** can be accessed from a class **C2** in the same package and from a class **C3** in a different package.

```
package p1;
public class C1 {
  public int x;
  int y;
  private int z;
  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;
public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;
    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p2;
public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;
    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

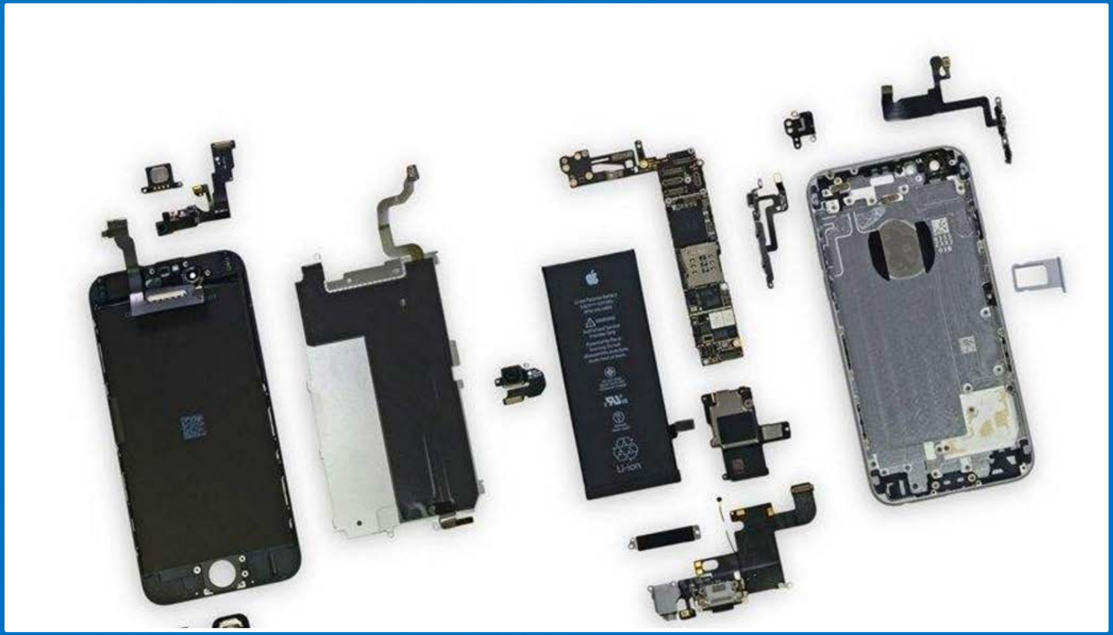
Data Field Encapsulation

Making data fields private *protects* data and makes the class easy to maintain. In following code, the data fields *radius* and *numberOfCircles* in the *Circle* class can be modified directly. This is not a good practice: (1) data may be tampered with; (2) the class becomes difficult to maintain and vulnerable to bugs. Suppose you want to modify the Circle class to ensure that the *radius* is nonnegative after other programs have already used the class. You have to change not only the circle class but also the programs that use it. To prevent direct modifications of data fields, you should declare the data fields *private*, using the *private modifier*. This is known as *data field encapsulation*.

```
class Circle{
  double radius; //private double radius;
  static int numberOfCircles; // private static int numberOfCircles
  final static double PI = 3.14;
  public Circle(double r){ numberOfCircle++; radius = r;}
}
```

Supplementary: Encapsulation (OO Feature)

- Bind together the data and functions that manipulate the data
- Hide the data inside a class, preventing unauthorized access
- Provide publicly accessible methods for data access



Exercises

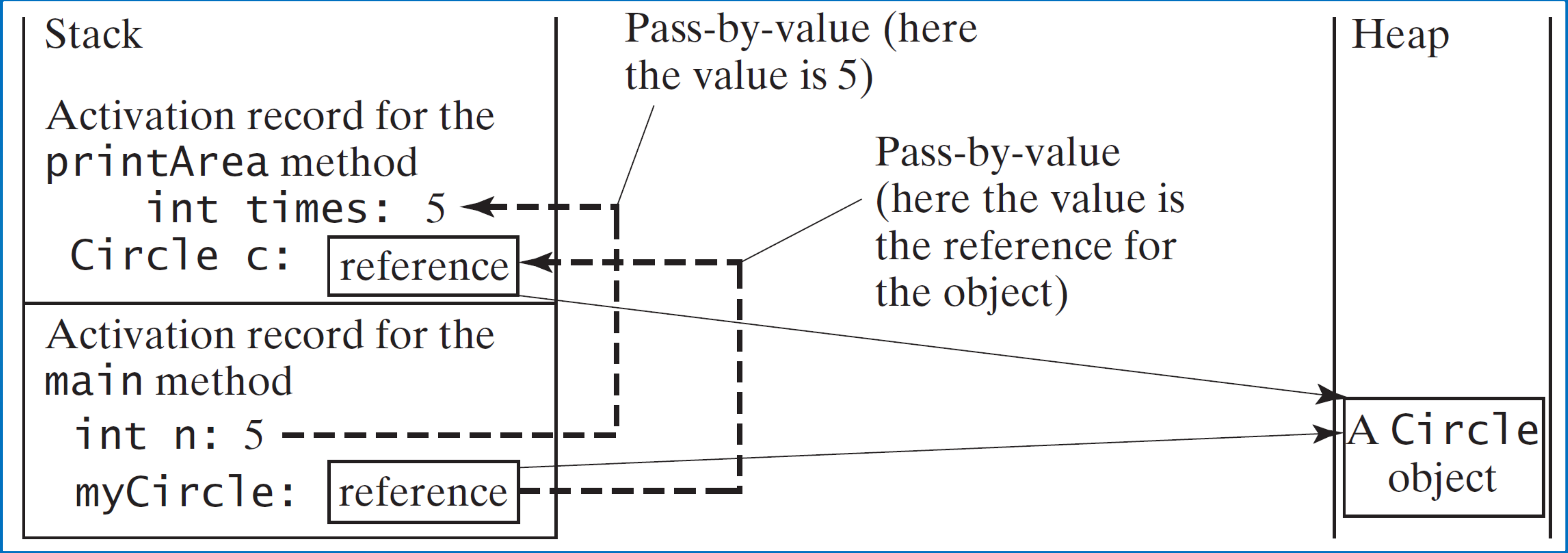
In the following code, *radius* is *private* in the *Circle* class, and *myCircle* is an *object* of the *Circle* class. Does the *highlighted* code cause any problems? If so, explain why.

```
public class Circle {
  private double radius = 1;
    /** Find the area of this circle */
    public double getArea() {
    return radius * radius * Math.PI;
  }
  public static void main(String[] args) {
    Circle myCircle = new Circle();
    System.out.println("Radius is " + myCircle.radius);
  }
}
```

Passing Objects to Methods

Passing an object to a method is to *pass the reference of the object*.

TestPassObject


Stack
Activation record for the printArea method
int times: 5
Circle c: reference
Activation record for the main method
int n: 5
myCircle: reference
Pass-by-value (here the value is 5)
Pass-by-value (here the value is the reference for the object)
Heap
A Circle object

Exercises

Describe the difference between passing a parameter of a *primitive type* and passing a parameter of a *reference type*. Show the output of the following programs:

```java
public class Test {
  public static void main(String[] args) {
    Count myCount = new Count();
    int times = 0;
    for (int i = 0; i < 100; i++)
      increment(myCount, times);
    System.out.println("count is " + myCount.count);
    System.out.println("times is " + times);
  }
  public static void increment(Count c, int times) {
    c.count++;
    times++;
  }
}
```

```java
public class Count {
  public int count;
  public Count(int c) {
    count = c;
  }
  public Count() {
    count = 1;
  }
}
```

Exercises

Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    int[] a = {1, 2};
    swap(a[0], a[1]);
    System.out.println("a[0] = " + a[0] + " a[1] = " + a[1]);
  }
  public static void swap(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
  }
}
```

Exercises

Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    int[] a = {1, 2};
    swap(a);
    System.out.println("a[0] = " + a[0] + " a[1] = " + a[1]);
  }
  public static void swap(int[] a) {
    int temp = a[0];
    a[0] = a[1];
    a[1] = temp;
  }
}
```

Exercises

Show the output of the following code:

```java
public class Test {
  public static void main(String[] args) {
    T t = new T();
    swap(t);
    System.out.println("e1 = " + t.e1 + " e2 = " + t.e2);
  }
  public static void swap(T t) {
    int temp = t.e1;
    t.e1 = t.e2; t.e2 = temp;
  }
}
class T {
  int e1 = 1;
  int e2 = 2;
}
```

Exercises

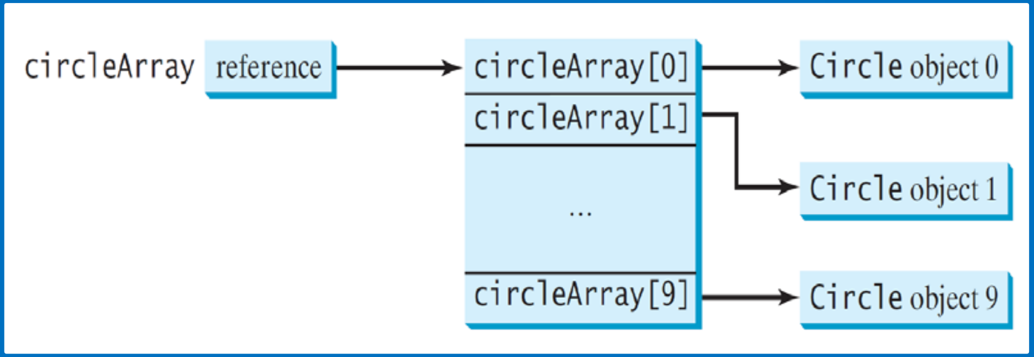
Show the output of the following code:

```java
public class Test {
  public static void main(String[] args) {
    T t1 = new T(); T t2 = new T();
    System.out.println("t1's i = " + t1.i + " and j = " + t1.j);
    System.out.println("t2's i = " + t2.i + " and j = " + t2.j);
  }
}
class T {
  static int i = 0;
  int j = 0;
  T() {
    i++;
    j = 1;
  }
}
```

Array of Objects

An array can hold objects as well as primitive type values. When an array of objects is created using the new operator, each element in the array is a reference variable with a default value of *null*.

```
Circle[] circleArray = new Circle[10];
for (int i = 0; i < circleArray.length; i++)
{
  circleArray[i] = new Circle();
}
```



Exercises

What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    java.util.Date[] dates = new java.util.Date[10];
    System.out.println(dates[0]);
    System.out.println(dates[0].toString());
  }
}
```

The Scope of Variables

The scope of *instance* and *static variables* is the *entire class*, *regardless* of *where* the variables are *declared*. A class's variables and methods can appear in any order in the class, as shown in (a). The *exception* is when a data field is *initialized based on* a reference to *another data field*. In such cases, the other data field must be declared first, as shown in (b).

```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }
  private double radius = 1;
}
```

(a)

```
public class F {

  private int i ;

  private int j = i + 1;

}
```

(b)

Exercises

Show the output of the following code:

```java
public class Test {
  private static int i = 0;
  private static int j = 0;
  public static void main(String[] args) {
    int i = 2;
    int k = 3;
    {
      int j = 3;
      System.out.println("i + j is " + i + j);
    }
    k = i + j;
    System.out.println("k is " + k);
    System.out.println("j is " + j);
  }
}
```

The *this* Reference

The *this* keyword is the name of a reference that an object can use to refer to *itself*. You can use the *this* keyword to reference the object's *instance members*. For example, the following code in (a) uses *this* to reference the object's radius and invokes its *getArea()* method explicitly. The *this* reference is normally *omitted*, as shown in (b).

```java
public class Circle {
  private double radius;
  ...
  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
  public String toString() {
    return "radius: " + this.radius
    + "area: " + this.getArea() ;
  }
}
```

(a)

```java
public class Circle {
  private double radius;
  ...
  public double getArea() {
    return radius * radius * Math.PI;
  }
  public String toString() {
    return "radius: " + radius
    + "area: " + getArea() ;
  }
}
```

(b)

Use *this* reference hidden data fields

The *this* keyword can be used to reference a class's *hidden data fields*. For example, a *data-field name* is often used as the *parameter name* in a setter method for the data field. In this case, the data field is hidden in the setter method. You need to reference the hidden data-field name in the method in order to set a new value to it. A hidden static variable can be accessed simply by using the *ClassName.staticVariable* reference. A hidden instance variable can be accessed by using the keyword *this*.

```
public class F {
  private int i = 5;
  private static double k = 0;
  public void setI(int i) {
    this.i = i;
  }
    public static void setK(double k) {
    F.k = k;
  }
}
```

Use *this* invoke constructor

The *this* keyword can be used to *invoke another constructor* of the same class. For example, you can rewrite the Circle class as follows:

```
public class Circle {
  private double radius;
  public Circle(double radius) {
    this.radius = radius;
  }
  public Circle() {
    this(1.0);
  }
  ...
}
```

Caution

this keyword can not exist in *class methods* (*static methods*).

```
public class Circle {
  public static int numberOfCircles;
  public double radius;

  public static void printRadius(){
    System.out.println(this.radius); //Error
  }
}
```

Exercises

What is wrong in the following code?

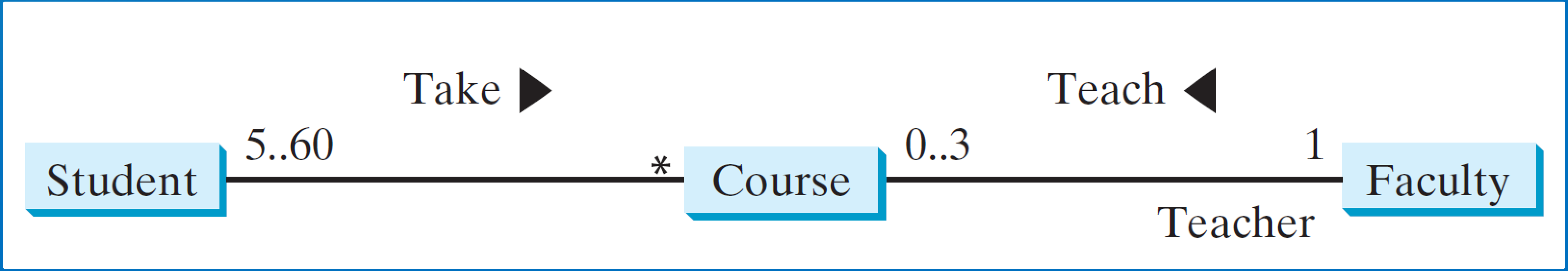
```java
public class C {
  private int p;
  public C() {
    System.out.println("C's no-arg constructor invoked");
    this(0);
  }
  public C(int p) {
    p = p;
  }
  public void setP(int p) {
    p = p;
  }
}
```

Object-Oriented Thinking

Object Relationships

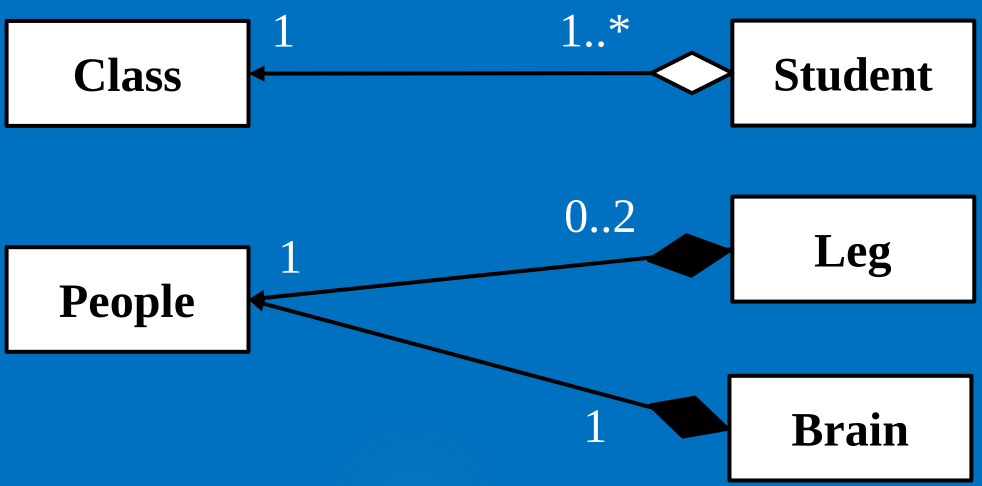
To design classes, you need to explore the relationships among objects. The common relationships among classes are *association*, *aggregation*, *composition*, and *inheritance*.

Association is a general binary relationship that describes an activity between two objects. For example, a *student taking a course* is an association between the *student* and the *course*, and a *teacher teaching a course* is an association between the *teacher* and the *course*.



Object Relationships

Aggregation is a special form of association that represents an *ownership relationship* between two objects. *Aggregation* models *has-a relationships*. The *owner object* is called an *aggregating object*, and its class is called an *aggregating class*. The *subject object* is called an *aggregated object*, and its class is called an *aggregated class*. An object can be owned by several other aggregating objects. If an object is *exclusively owned* by an aggregating object, the relationship between the object and its aggregating object is referred to as a *composition*.



Example: Design a Class Using Composition

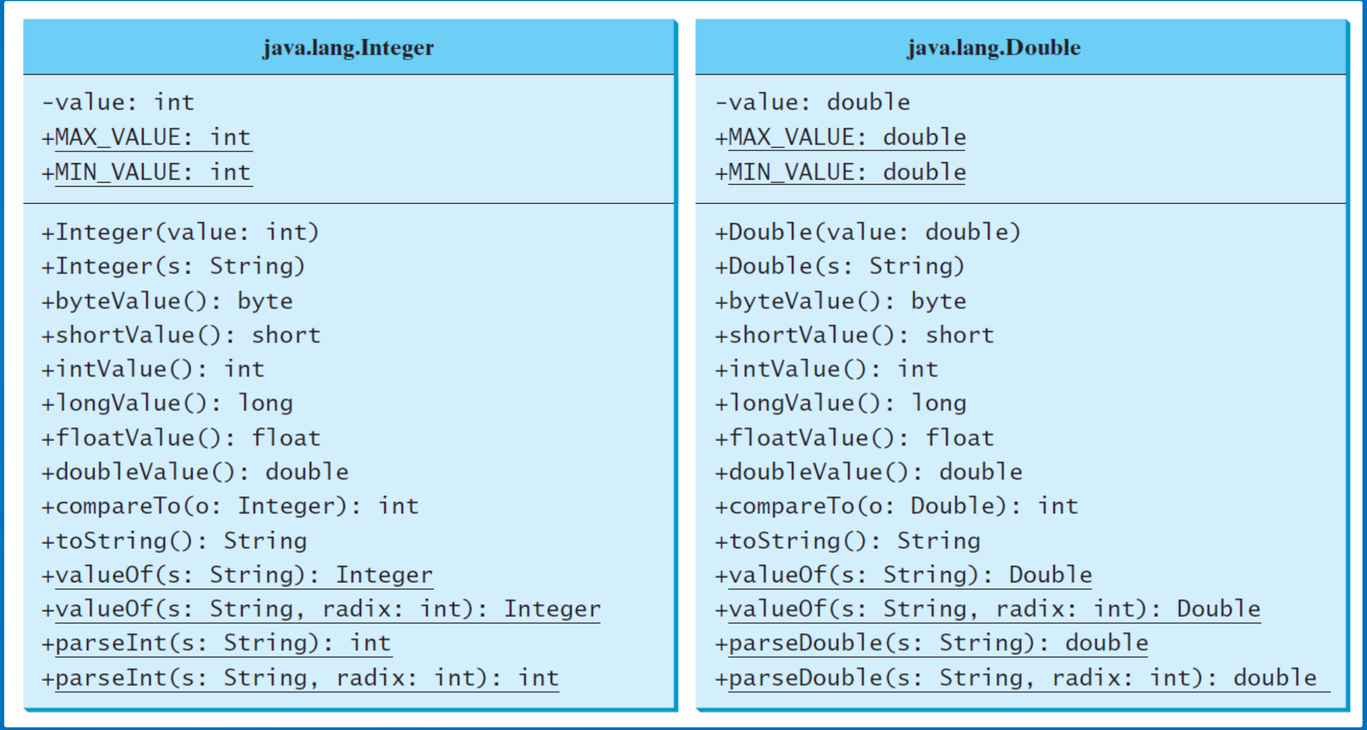
A computer consists of memory, CPU and other components.

Example: Design a Class Using Composition

```
public class Computer{
  private Memory[] mList;
  private CPU cpu;
  public void computerInfo(){
    System.out.println("CPU: " + cpu.getVersion());
    for(int i = 0; i < mList.length; i++){
      System.out.println("Memory[%d]: %s", i, mList[i].getVersion());
    }
  }
}
class Memory {
  private String version;
}
class CPU {
  private String version;
}
```

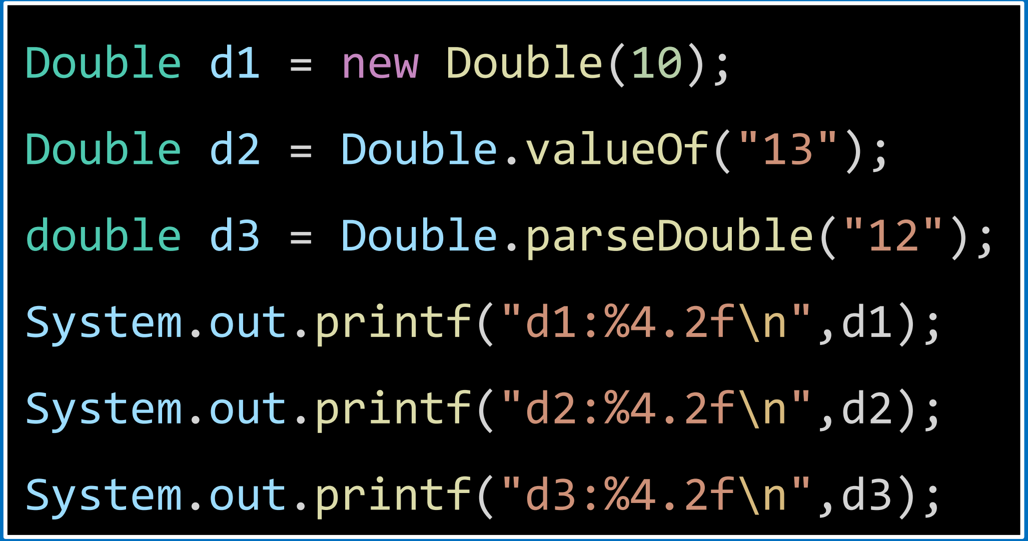
Wrapper Classes of Primitive Data Types

A primitive type value is not an object, but it can be wrapped in an object using a *wrapper class* in the Java API. Java provides *Boolean*, *Character*, *Double*, *Float*, *Byte*, *Short*, *Integer*, and *Long* wrapper classes in the *java.lang* package for primitive data types. The wrapper classes provide *constructors*, *constants*, and *conversion methods* for manipulating various data types.



```
java.lang.Integer
-value: int
+MAX_VALUE: int
+MIN_VALUE: int
+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Integer): int
+toString(): String
+valueOf(s: String): Integer
+valueOf(s: String, radix: int): Integer
+parseInt(s: String): int
+parseInt(s: String, radix: int): int
```

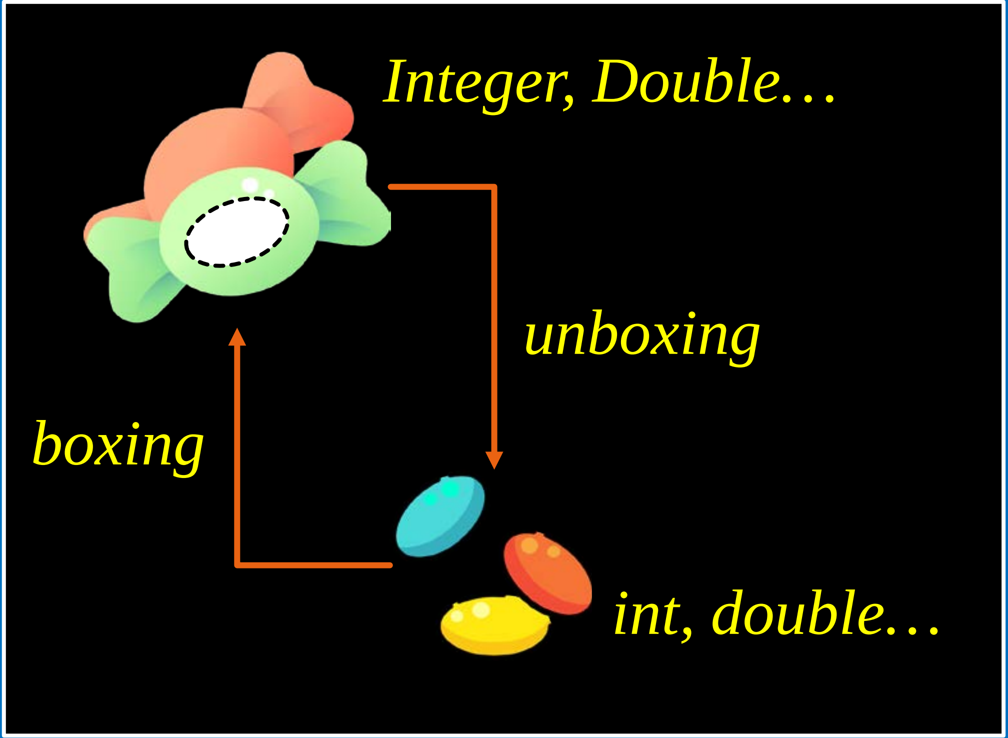
```
java.lang.Double
-value: double
+MAX_VALUE: double
+MIN_VALUE: double
+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+valueOf(s: String, radix: int): Double
+parseDouble(s: String): double
+parseDouble(s: String, radix: int): double
```



```java
Double d1 = new Double(10);
Double d2 = Double.valueOf("13");
double d3 = Double.parseDouble("12");
System.out.printf("d1:%4.2f\n",d1);
System.out.printf("d2:%4.2f\n",d2);
System.out.printf("d3:%4.2f\n",d3);
```

Auto-boxing and Auto-unboxing

A *primitive* type value can be *automatically converted* to an object using a *wrapper* class, and vice versa, depending on the context. Converting a primitive value to a wrapper object is called *boxing*. The reverse conversion is called *unboxing*.



```
// auto-boxing
Double d1 = 12.12;
Integer[] list = {1, 2, 3};
// auto-unboxing
double d2 = d1 * 2.0;
System.out.println(d1 * 5);
System.out.println(list[0]+list[1]+list[2]);
// auto-unboxing and auto-boxing
Double d3 = d1 * 3.0;
```

The *BigInteger* and *BigDecimal* Classes

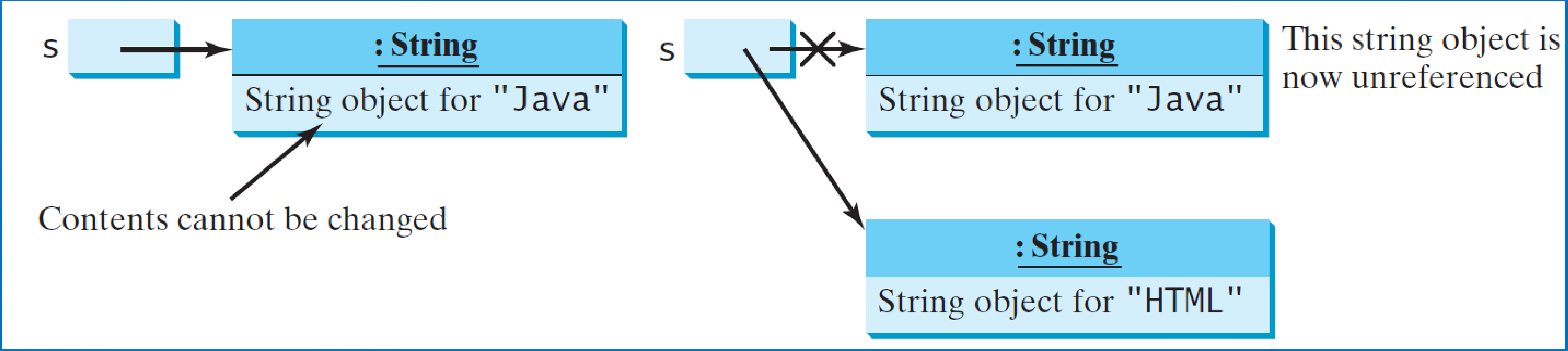
The *BigInteger* and *BigDecimal* classes can be used to represent integers or decimal numbers of any size and precision.

```
public class Client {
  public static void main(String[] args) {
    BigInteger a = new BigInteger("9223372036854775807");
    BigInteger b = new BigInteger("2");
    BigInteger c = a.multiply(b); // 9223372036854775807 * 2
    System.out.println(c);
    BigDecimal x = new BigDecimal(1.0);
    BigDecimal y = new BigDecimal(3);
    BigDecimal z = x.divide(y, 20, BigDecimal.ROUND_UP);
    System.out.println(z);
  }
}
```

Immutable Strings and Interned Strings

A *String* object is immutable; its contents cannot be changed. Does the following code change the contents of the string? The answer is *no*. Strings are *immutable*; once created, their contents cannot be changed.

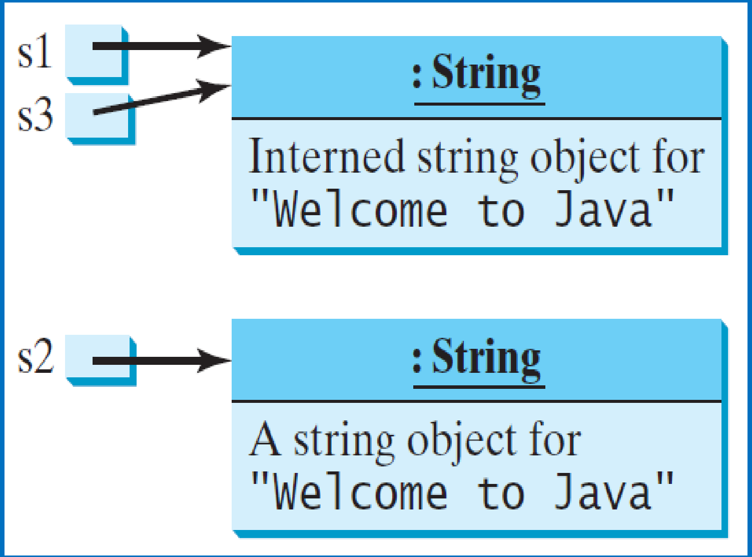
```
String s = "Java";
s = "HTML";
```



Immutable Strings and Interned Strings

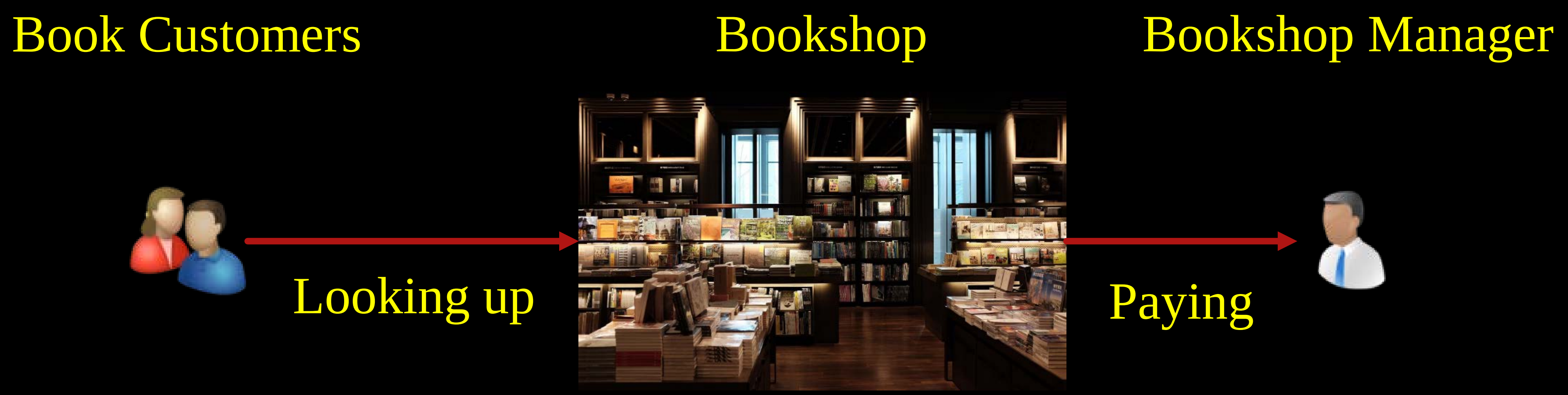
Because strings are *immutable* and are *ubiquitous* in programming, the JVM uses a unique instance for *string literals* with the same character sequence in order to *improve efficiency* and *save memory*. Such an instance is called an *interned string*. For example, the following statements:

```java
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

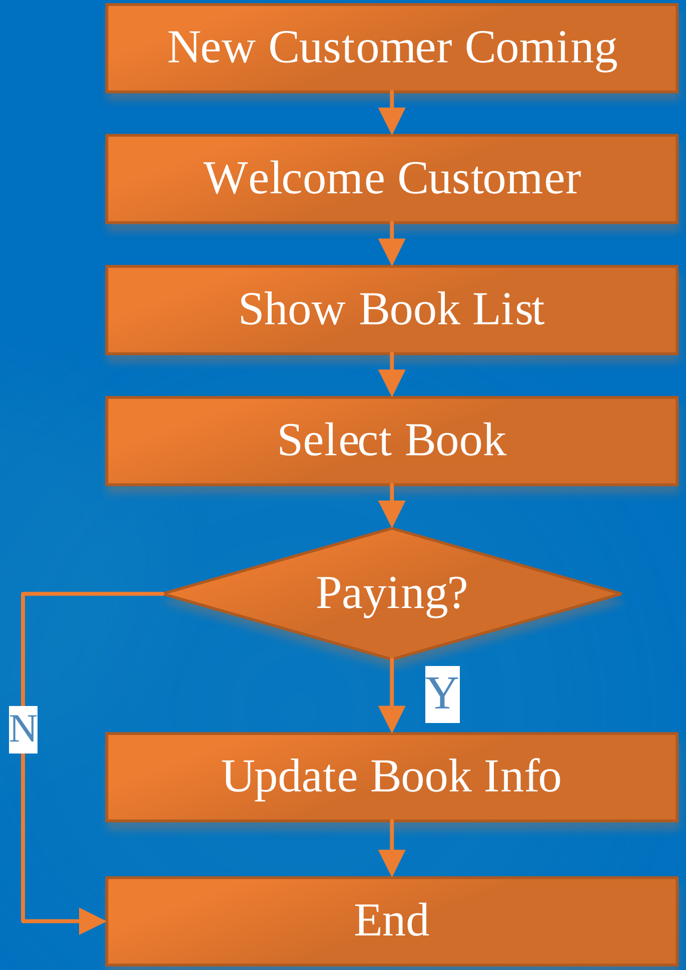


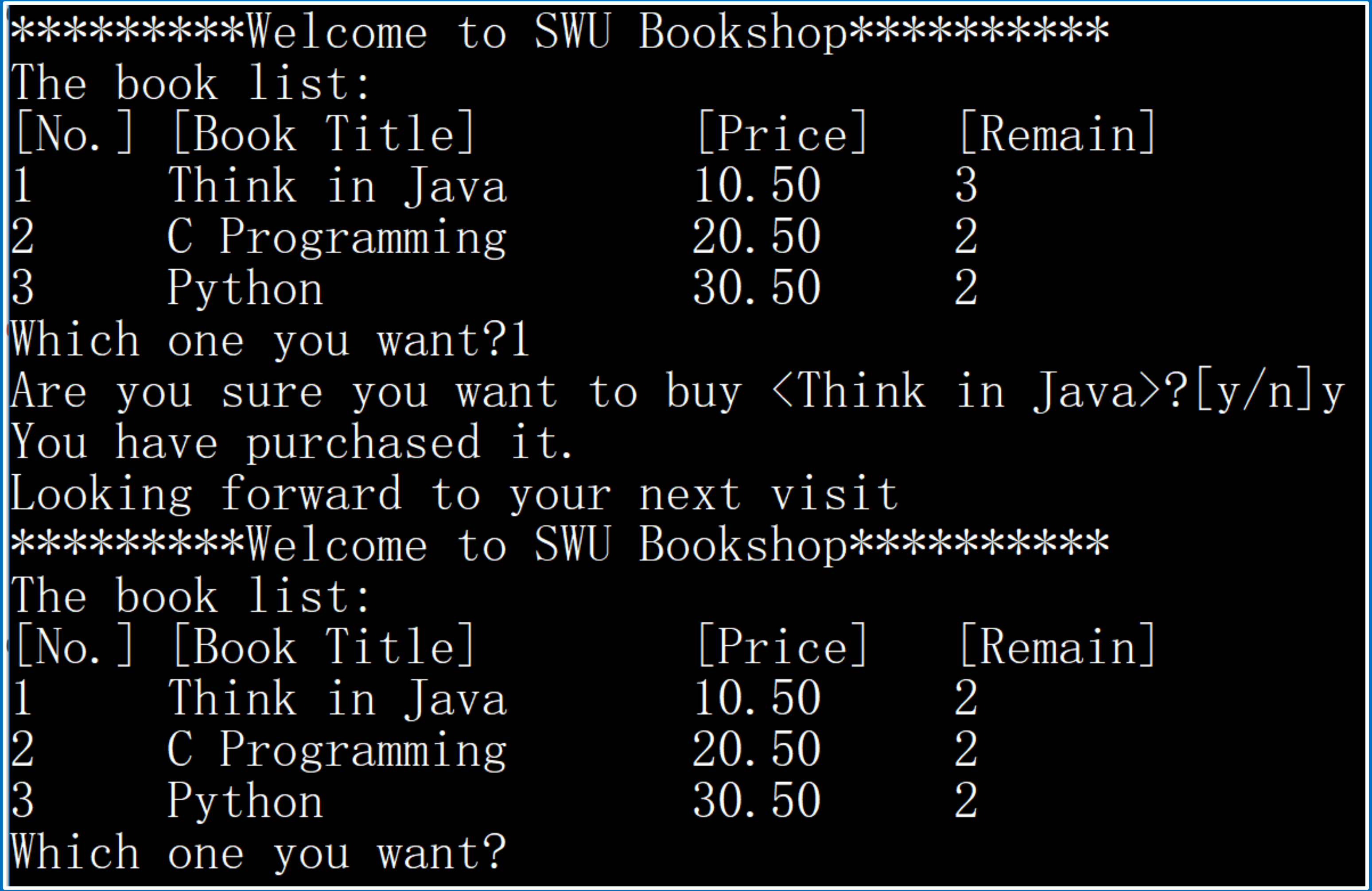
Procedural Paradigm vs Object-oriented Paradigm

A Case: Write a program to simulate the process of buying a book in a bookstore.



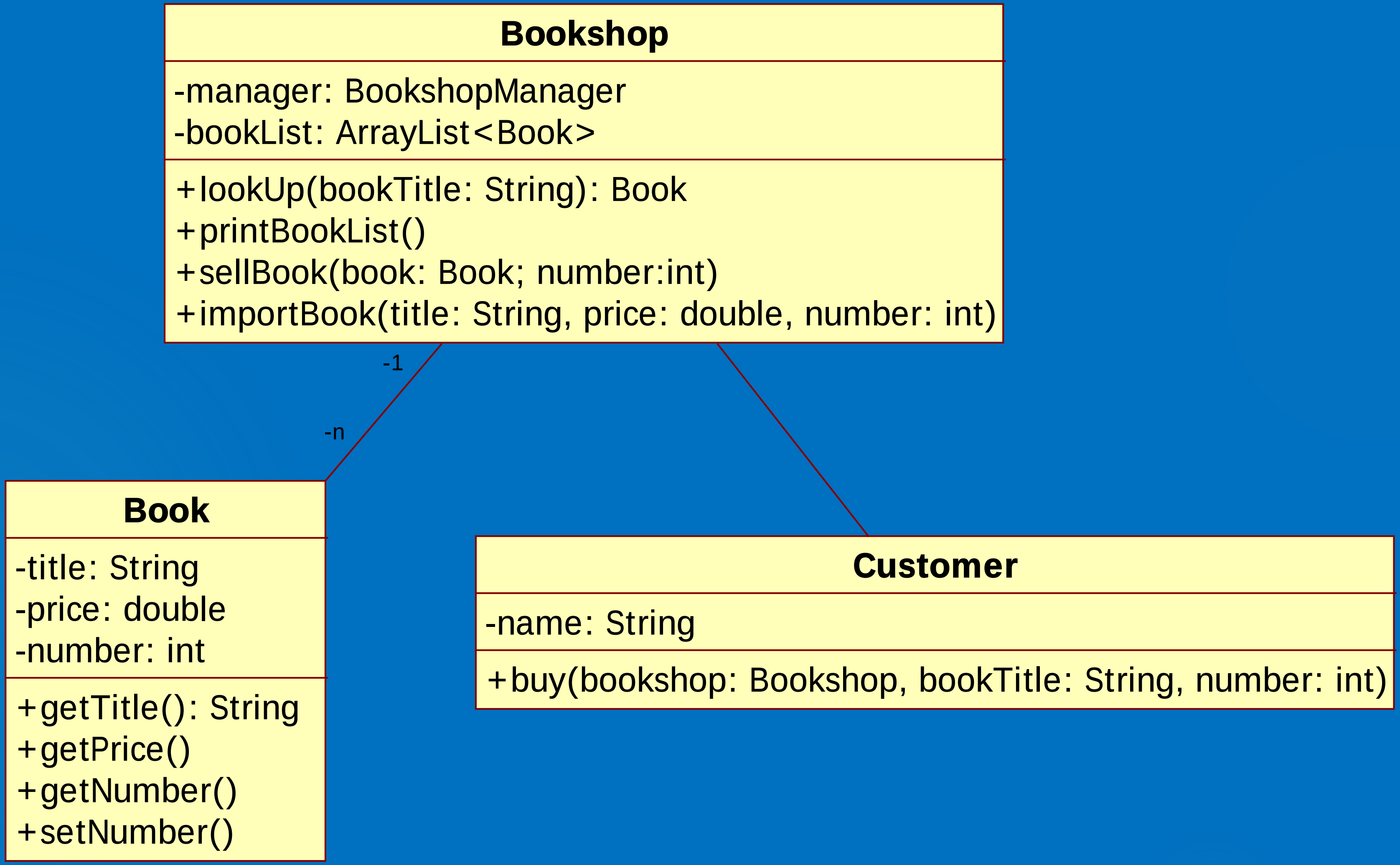
Procedural Programming





```
*********Welcome to SWU Bookshop**********
The book list:
[No.] [Book Title]          [Price]   [Remain]
1     Think in Java         10.50     3
2     C Programming         20.50     2
3     Python                30.50     2
Which one you want?1
Are you sure you want to buy <Think in Java>?[y/n]y
You have purchased it.
Looking forward to your next visit
*********Welcome to SWU Bookshop**********
The book list:
[No.] [Book Title]          [Price]   [Remain]
1     Think in Java         10.50     2
2     C Programming         20.50     2
3     Python                30.50     2
Which one you want?
```

Object-Oriented programming


Bookshop
-manager: BookshopManager
-bookList: ArrayList<Book>
+lookUp(bookTitle: String): Book
+printBookList()
+sellBook(book: Book; number:int)
+importBook(title: String, price: double, number: int)
-1
-n
Book
-title: String
-price: double
-number: int
+getTitle(): String
+getPrice()
+getNumber()
+setNumber()
Customer
-name: String
+buy(bookshop: Bookshop, bookTitle: String, number: int)