

Class Creation

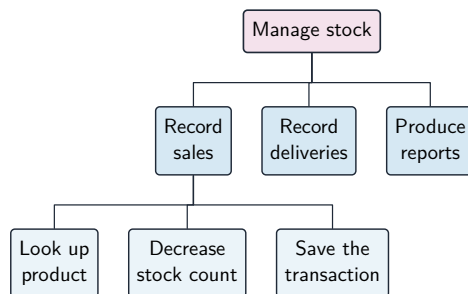
AP Computer Science A

Abstraction and Program Design



A jigsaw in progress: classes and methods are modular pieces of a larger program design

Abstraction 抽象 means hiding detail behind a simple interface – you use a `String` without knowing how it stores characters. Good design breaks a problem into classes, each responsible for one idea. This topic is about writing **your own** classes.



Decomposing a program into modules and sub-modules

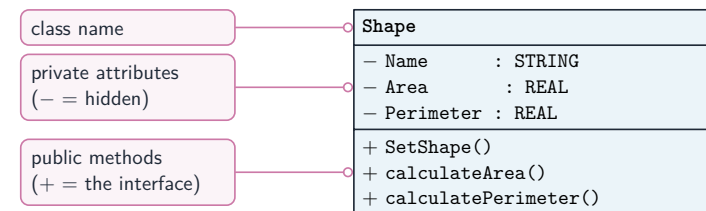
The Impact of Program Design

Design choices affect whether code is correct, readable, and reusable. **Encapsulation** 封装 – keeping data private and exposing it only through methods – protects an object’s state from misuse and lets you change the inside without breaking users of the class. Thoughtful naming, single-purpose methods, and testing reduce bugs.

Design also carries responsibility beyond the code. **System reliability** 系统可靠性 - a program performing its tasks as expected, without failure - is something programmers should maximise through careful design and testing. Programs have real impacts on society, the economy, and culture that can be **both beneficial and harmful**. And creating programs raises **legal and intellectual-property** 知识产权 concerns: programmers often reuse code published as **open source** 开源 and free to use, but must respect its licence and give credit rather than copy others’ work as their own.

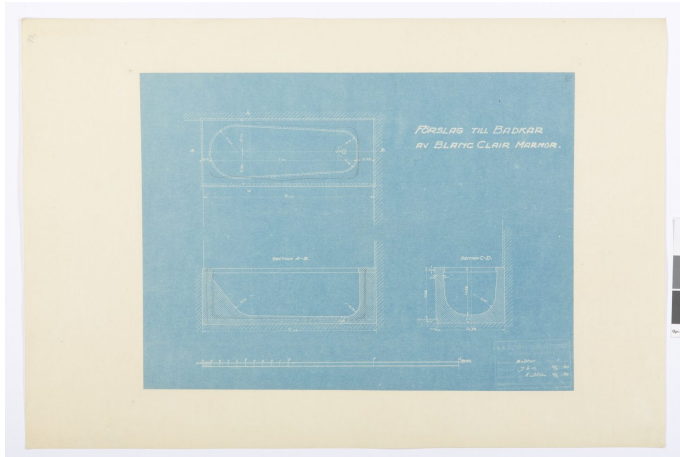
The Anatomy of a Class

A class has three parts: **instance variables** 实例变量 (fields – the object’s data), **constructors** (build objects), and **methods** (behavior). Fields are usually **private**; methods are usually **public**:



A class diagram: private attributes and public methods

```
public class Student {  
    private String name;        // instance variable  
    private int score;  
  
    public Student(String n, int s) { // constructor  
        name = n;  
        score = s;  
    }  
    public int getScore() { return score; } // accessor  
}
```



A blueprint: a class is a template that defines how objects of that type are built

Constructors

A **constructor** 构造函数 has the **same name as the class** and no return type. It runs when you write **new**, and its job is to initialize the fields. A class can have several constructors with different parameter lists (**overloading** 重载); a **no-argument** constructor sets defaults.

Methods: How to Write Them

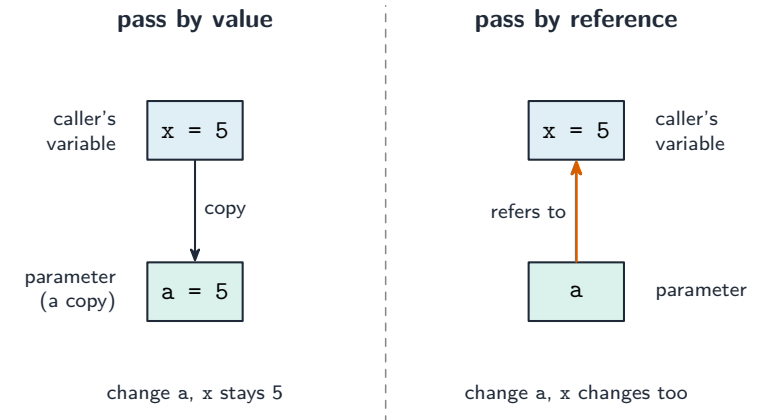
A method has a signature, a return type, and a body. An **accessor (getter)** 访问器 returns information without changing the object; a **mutator (setter)** 修改器 changes a field. A method returning a value must have a **return** of the right type on every path; a **void** method returns nothing.

```
public void setScore(int s) { score = s; } // mutator
public String toString() { return name + ": " + score; }
```

Passing and Returning References of an Object

When you pass an object to a method, Java copies the **reference**, so the method acts on the **same** object – changes to its fields are visible to the caller. (Primitives are copied by value, so changes to them are not.) A method can also **return** a reference

to an object. Because a **String** is immutable, passing one is safe; passing a mutable object lets a method change it.



Java always passes **by value** (left): the method gets a copy of the reference. True pass by reference (right) — which Java does not have — would let a method reassign the caller's own variable.

Exam skill: know that mutating an object's fields inside a method affects the original, but **reassigning** the parameter (**param = new...**) does not affect the caller.

Worked example. Suppose **s** is a **Student** with score 50, and we call **tweak(s)**:

```
public static void tweak(Student a) {
    a.setScore(100); // (1) mutates the shared object
    a = new Student("Z", 0); // (2) repoints the local copy only
    a.setScore(5); // (3) changes only the new local object
}
```

Line (1) changes the object **s** points to, so the caller now sees 100. Line (2) makes the method's own copy of the reference point at a fresh object – the caller's **s** is untouched – and line (3) affects only that new object. After the call, **s.getScore()** is **100**: the mutation stuck, the reassignment did not.

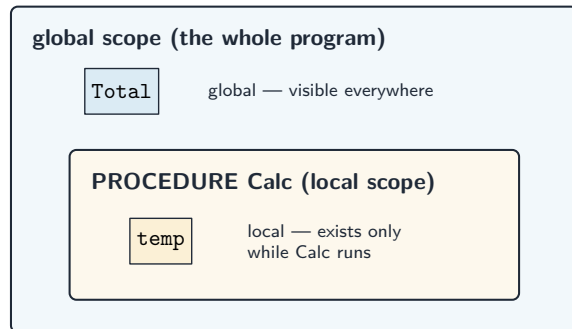
Class Variables and Class Methods

A **static (class) variable** 类变量, marked **static**, is shared by **all** objects of the class – one copy total (e.g. a counter of how many objects exist). A **static method**

belongs to the class and cannot use instance fields directly. Access them by class name: `Student.getCount()`.

Scope and Access

Scope 作用域 is where a name is visible. A **local variable** declared in a method exists only inside it; a **parameter** exists only in its method; an **instance variable** is visible throughout the object. Access modifiers control visibility across classes: **private** (this class only) versus **public** (anywhere). Local variables **shadow** fields of the same name – a source of bugs.



A global variable is visible everywhere; a local variable only inside its block

The this Keyword

this is a reference to the **current object**. Use it to tell a field apart from a parameter with the same name, or to call another method of the same object:

```
public Student(String name, int score) {
    this.name = name;           // this.name is the field; name is the
    ↪ parameter
    this.score = score;
}
```

Exam skill: when a constructor or setter's parameter has the same name as a field, you **must** write `this.field = param` – without `this`, the assignment does nothing useful.

Exam tips

- Design with methods and classes: encapsulate data as **private** fields and expose behaviour through public methods.
- Know the difference between an **object** and its **class**, and that objects are passed **by value** — the parameter gets a copy of the reference, so a method can change the object's state, but reassigning the parameter does not affect the caller (Java has no pass-by-reference).
- Traverse arrays and **ArrayLists** safely — size is **length** vs **.size()**, and removing during a loop shifts indices.
- **Trace** a **recursive** method to determine its result: find the base case first, then follow each recursive call to its return value (writing recursive code is outside the exam's scope).
- Recognise **inheritance** vocabulary — superclass, subclass, method overriding, and that every class is a subclass of **Object** (designing and implementing inheritance is outside the exam's scope).