Chapter 4.2: Encapsulation

Object Oriented Programming with Java

Information hiding | Pre- and postconditions | Access modifiers | Getters and setters | Interface and implementation | Immutable classes | Packages and imports

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_04_02_Encapsulation.html

INTRODUCTION

Learning Outcomes

Explain encapsulation

Show with examples how a class bundles data with the methods that use it, hides internal details (information hiding) and separates “what” from “how”.

Write method contracts

State a method's **precondition** and **postcondition** in a comment and check them in code.

Choose the right access

`public`, `private`, `protected` and package access: use a table to say who can reach each.

Write getters and setters

Make fields `private` and write validating accessors; use the `this.field = field` idiom.

Separate interface from implementation

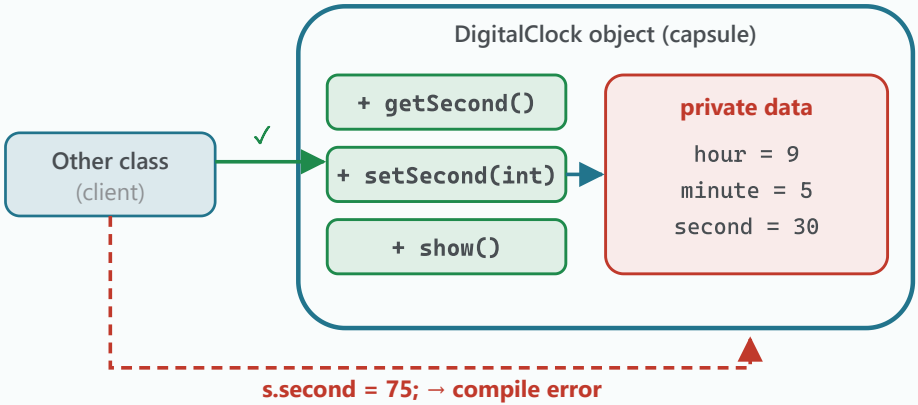
Change a class's internals without changing its outside; meet `interface` and `implements`.

Use packages

`package` and `import`: group classes, compile and run them, and resolve class name clashes.

Encapsulation

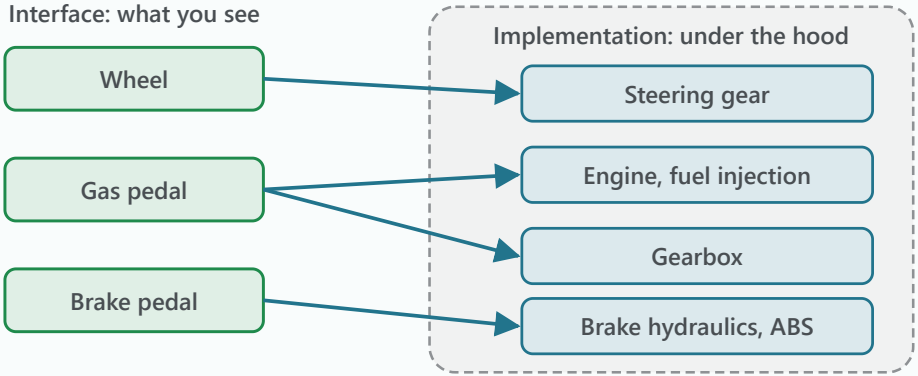
- **Encapsulation**: bundling an object's **data** (fields) and the **methods** that work on it into one unit, the class.
- The data is **hidden** (`private`); outside code can call only the **methods** the class offers.
- Analogy: a medicine capsule. You never see the powder inside; you just swallow the capsule. Likewise, a class is the **guardian** of its own data: an invalid value (e.g. second 75) cannot get in.



Encapsulation is one of the four pillars of object-oriented programming (the others: inheritance, polymorphism, abstraction).

Information Hiding

- **Information hiding** is also called **abstraction**. A design should separate **what** is done from **how** it is done.
- A method should be usable without knowing its internals: knowing **what it does** should be enough.
- In a car we see and use the brake, the gas pedal and the steering wheel; we don't know their **inner mechanics**.



The same idea applies in code: we use `System.out.println` all the time without knowing how printing to the screen works.

Information Hiding: Code

```java
public class DataService {
    private DbConnection conn;
    // usable without knowing the internals.
    public void addData(String data) {
        conn = new DbConnection();
        conn.connect();
        conn.addData(data);
        conn.close();
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        DataService service = new DataService();
        service.addData("Ayşe");
    }
}
```

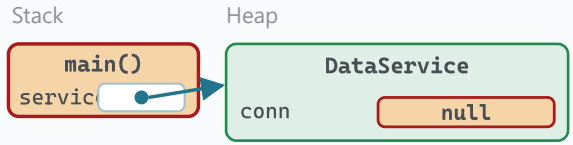
OUTPUT

```
Connection opened
Added: Ayşe
Connection closed
```

- `Main` only says `service.addData("Ayşe")`: it states **what** it wants.
- Opening the connection, writing the data and closing it (the **how**) are hidden inside `DataService`.
- The `conn` field is `private`: no outside code can see or change it.
- `DbConnection` is just a class that prints to the screen here; even with a real database driver, `Main` would not change.

Behind a Single Call: Step by Step

```
 1 public class DataService {
 2   private DbConnection conn;
 3   // usable without knowing the internals.
 4   public void addData(String data) {
 5     conn = new DbConnection();
 6     conn.connect();
 7     conn.addData(data);
 8     conn.close();
 9   }
10 }
11 public class Main {
12   public static void main(String[] args) {
▶      DataService service = new DataService();
14     service.addData("Ayşe");
15   }
16 }
```



`new DataService()` creates an object on the heap; `service` points to **it** (a reference). `conn` is still `null`.

Step 1 / 7

OUTPUT

CALL STACK

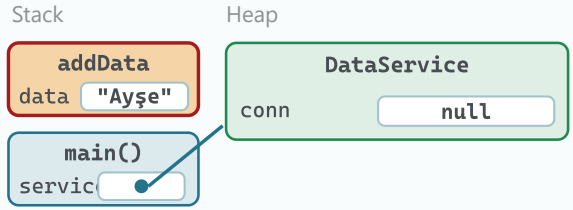
`main()`

VARIABLES

`service = →DataService`

Behind a Single Call: Step by Step

```
1  public class DataService {
2    private DbConnection conn;
3    // usable without knowing the internals.
▶    public void addData(String data) {
5      conn = new DbConnection();
6      conn.connect();
7      conn.addData(data);
8      conn.close();
9    }
10 }
11 public class Main {
12   public static void main(String[] args) {
13     DataService service = new DataService();
14     service.addData("Ayşe");
15   }
16 }
```



This call is all Main does. A new frame opens and the `data` parameter becomes "Ayşe".

Step 2 / 7

OUTPUT

VARIABLES

`data = "Ayşe"`

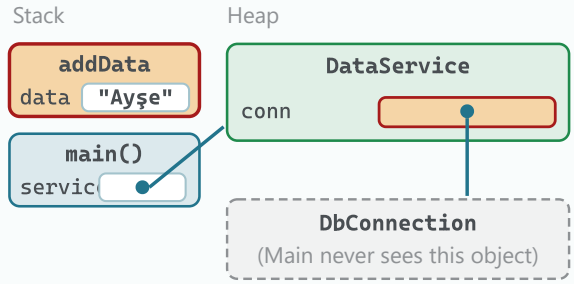
CALL STACK

`service.addData("Ayşe")`

`main()`

Behind a Single Call: Step by Step

```
1  public class DataService {
2    private DbConnection conn;
3    // usable without knowing the internals.
4    public void addData(String data) {
▶      conn = new DbConnection();
6      conn.connect();
7      conn.addData(data);
8      conn.close();
9    }
10 }
11 public class Main {
12   public static void main(String[] args) {
13     DataService service = new DataService();
14     service.addData("Ayşe");
15   }
16 }
```



The hidden work begins: a second object is created and the `private` `conn` field points to it.

Step 3 / 7

OUTPUT

VARIABLES

`data = "Ayşe"` `conn = →object`

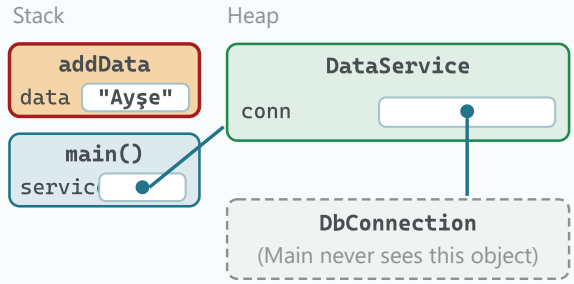
CALL STACK

`service.addData("Ayşe")`

`main()`

Behind a Single Call: Step by Step

```
1  public class DataService {
2    private DbConnection conn;
3    // usable without knowing the internals.
4    public void addData(String data) {
5      conn = new DbConnection();
▶      conn.connect();
7      conn.addData(data);
8      conn.close();
9    }
10 }
11 public class Main {
12   public static void main(String[] args) {
13     DataService service = new DataService();
14     service.addData("Ayşe");
15   }
16 }
```



`connect()` is called: three frames on the stack. `Main` knows nothing of it.

Step 4 / 7

OUTPUT

```
Connection opened
```

VARIABLES

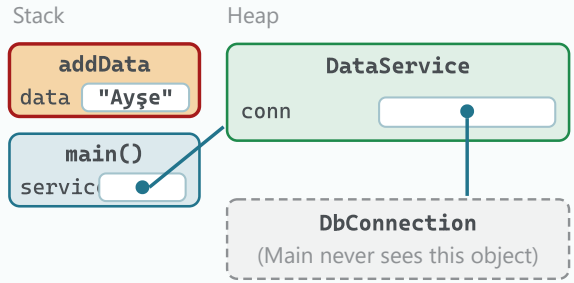
`data = "Ayşe"`

CALL STACK

- `conn.connect()`
- `service.addData("Ayşe")`
- `main()`

Behind a Single Call: Step by Step

```
1  public class DataService {
2    private DbConnection conn;
3    // usable without knowing the internals.
4    public void addData(String data) {
5      conn = new DbConnection();
6      conn.connect();
▶      conn.addData(data);
8      conn.close();
9    }
10 }
11 public class Main {
12   public static void main(String[] args) {
13     DataService service = new DataService();
14     service.addData("Ayşe");
15   }
16 }
```



The data goes to the connection object's `addData` method: same name, **different class**.

Step 5 / 7

OUTPUT

```
Connection opened
Added: Ayşe
```

VARIABLES

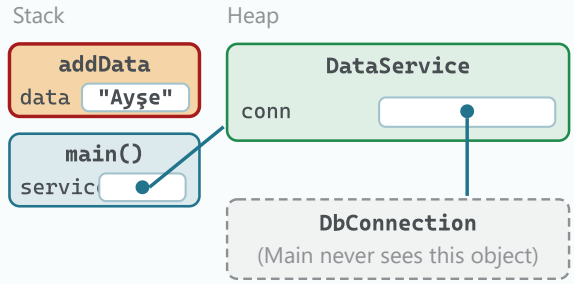
`data = "Ayşe"`

CALL STACK

- `conn.addData(data)`
- `service.addData("Ayşe")`
- `main()`

Behind a Single Call: Step by Step

```
1  public class DataService {
2    private DbConnection conn;
3    // usable without knowing the internals.
4    public void addData(String data) {
5      conn = new DbConnection();
6      conn.connect();
7      conn.addData(data);
▶      conn.close();
9    }
10 }
11 public class Main {
12   public static void main(String[] args) {
13     DataService service = new DataService();
14     service.addData("Ayşe");
15   }
16 }
```



The connection is closed. All three steps hide behind a single call.

Step 6 / 7

OUTPUT

```
Connection opened
Added: Ayşe
Connection closed
```

VARIABLES

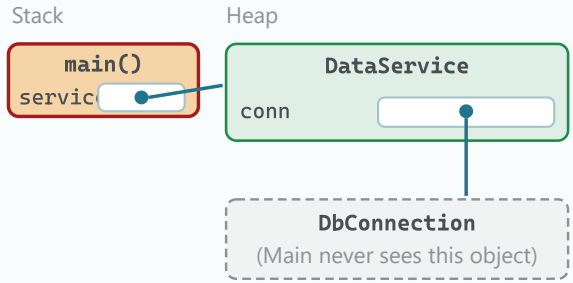
`data = "Ayşe"`

CALL STACK

- `conn.close()`
- `service.addData("Ayşe")`
- `main()`

Behind a Single Call: Step by Step

```
1  public class DataService {
2    private DbConnection conn;
3    // usable without knowing the internals.
4    public void addData(String data) {
5      conn = new DbConnection();
6      conn.connect();
7      conn.addData(data);
8      conn.close();
9    }
10 }
11 public class Main {
12   public static void main(String[] args) {
13     DataService service = new DataService();
14     service.addData("Ayşe");
▶    }
16 }
```



`main` resumes. The user wrote one line and never saw the details: **information hiding**.

Step 7 / 7

OUTPUT

```
Connection opened
Added: Ayşe
Connection closed
```

CALL STACK

`main()`

VARIABLES

`service = →DataService`

Preconditions and Postconditions

Precondition

A condition assumed true **before** the method is called. Ensuring it is the **caller's** responsibility.

```
public class DataService {
    // Precondition: data != null
    public void addData(String data) {
        // check whether data is null
        // ...
    }
}
```

Postcondition

If the precondition held, the state guaranteed to be true **after** the method finishes. Ensuring it is the **method's** responsibility.

```
public class DataService {
    // Postcondition: returns true if data was added
    public boolean addData(String data) {
        // check whether the data was added
        // ...
        return true;
    }
}
```

Together they form the method's **contract**: “Give me non-empty data, and I will add it and return `true`.” The user knows only the contract, not the body.

Contract: A Complete Example

```
public class DataService {
  private int count = 0;

  /**
   * Adds a record to the database.
   * Precondition: data is not null or empty.
   * Postcondition: returns true if added,
   *   and count grows by 1.
   */
  public boolean addData(String data) {
    if (data == null || data.isEmpty()) {
      return false;   // precondition broken
    }
    // ... add data to the database ...
    count++;
    return true;      // postcondition met
  }

  public int getCount() {
    return count;
  }
}
```

```
// in main:
DataService service = new DataService();
System.out.println(service.addData("Ali"));
System.out.println(service.addData(""));
System.out.println(service.addData(null));
System.out.println(service.getCount());
```

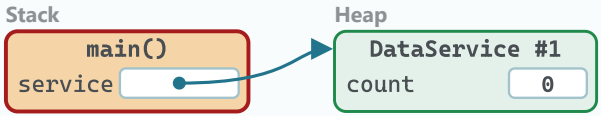
OUTPUT

```
true
false
false
1
```

- `/** ... */` is a **Javadoc** comment: it explains how to use the method.
- If the precondition is broken, the method adds nothing and returns `false`.

Contract: Code Trace

```
1  public boolean addData(String data) {
2      if (data == null || data.isEmpty()) {
3          return false;   // precondition broken
4      }
5      count++;
6      return true;        // postcondition met
7  }
8  // in main:
▶  DataService service = new DataService();
10 System.out.println(service.addData("Ayşe"));
11 System.out.println(service.addData(null));
12 System.out.println(service.addData(""));
13 System.out.println(service.addData("Can"));
14 System.out.println(service.getCount());
```



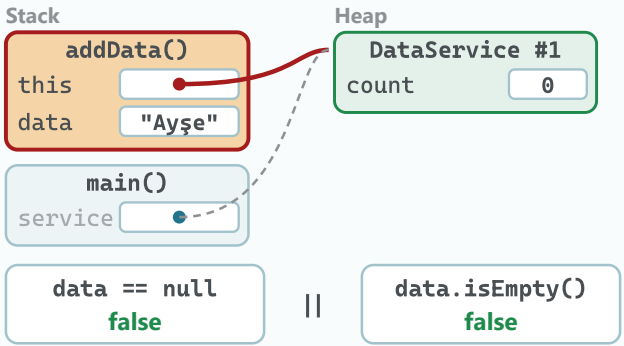
The object was just created: `count = 0`. The contract: `data` must be neither null nor empty.

Step 1 / 8

OUTPUT

Contract: Code Trace

```
1  public boolean addData(String data) {
▶      if (data == null || data.isEmpty()) {
3          return false;   // precondition broken
4      }
5      count++;
6      return true;        // postcondition met
7  }
8  // in main:
9  DataService service = new DataService();
▶  System.out.println(service.addData("Ayşe"));
11 System.out.println(service.addData(null));
12 System.out.println(service.addData(""));
13 System.out.println(service.addData("Can"));
14 System.out.println(service.getCount());
```



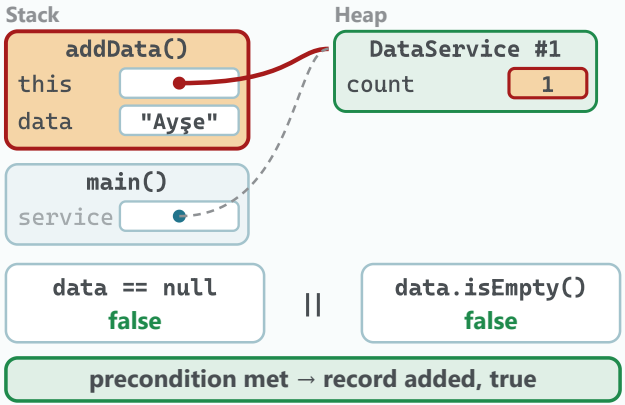
`"Ayşe"` is neither null nor empty: the condition is false; the `if` block is skipped.

Step 2 / 8

OUTPUT

Contract: Code Trace

```
1  public boolean addData(String data) {
2      if (data == null || data.isEmpty()) {
3          return false;    // precondition broken
4      }
▶      count++;
▶      return true;         // postcondition met
7  }
8  // in main:
9  DataService service = new DataService();
10 System.out.println(service.addData("Ayşe"));
11 System.out.println(service.addData(null));
12 System.out.println(service.addData(""));
13 System.out.println(service.addData("Can"));
14 System.out.println(service.getCount());
```



The postcondition holds: the counter goes up by 1 and **true** is returned.

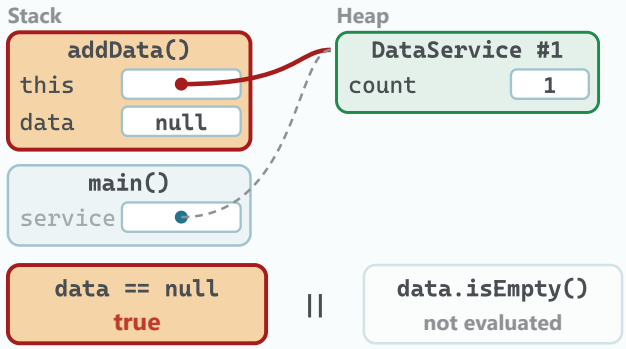
Step 3 / 8

OUTPUT

```
true
```

Contract: Code Trace

```
1  public boolean addData(String data) {
▶      if (data == null || data.isEmpty()) {
3          return false;   // precondition broken
4      }
5      count++;
6      return true;        // postcondition met
7  }
8  // in main:
9  DataService service = new DataService();
10 System.out.println(service.addData("Ayşe"));
▶  System.out.println(service.addData(null));
12 System.out.println(service.addData(""));
13 System.out.println(service.addData("Can"));
14 System.out.println(service.getCount());
```



`data == null` is true; `||` short-circuits. `null.isEmpty()` is never called, so no `NullPointerException` is thrown.

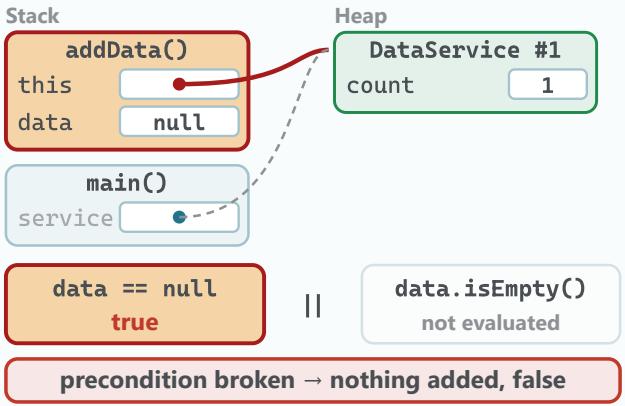
Step 4 / 8

OUTPUT

```
true
```

Contract: Code Trace

```
1  public boolean addData(String data) {
2      if (data == null || data.isEmpty()) {
▶          return false;   // precondition broken
4      }
5      count++;
6      return true;        // postcondition met
7  }
8  // in main:
9  DataService service = new DataService();
10 System.out.println(service.addData("Ayşe"));
11 System.out.println(service.addData(null));
12 System.out.println(service.addData(""));
13 System.out.println(service.addData("Can"));
14 System.out.println(service.getCount());
```



Precondition broken: without touching the object, the method returns **false**.

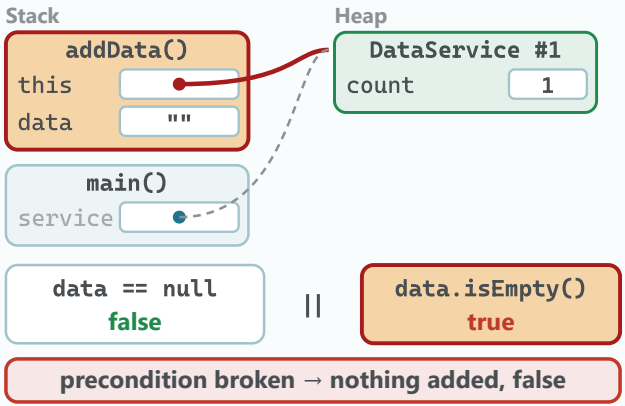
Step 5 / 8

OUTPUT

```
true
false
```

Contract: Code Trace

```
 1 public boolean addData(String data) {
 ▶     if (data == null || data.isEmpty()) {
 ▶         return false;   // precondition broken
 4     }
 5     count++;
 6     return true;        // postcondition met
 7 }
 8 // in main:
 9 DataService service = new DataService();
10 System.out.println(service.addData("Ayşe"));
11 System.out.println(service.addData(null));
   System.out.println(service.addData(""));
13 System.out.println(service.addData("Can"));
14 System.out.println(service.getCount());
```



`""` is not null but empty: the right side is true, so again **false**. The counter is unchanged.

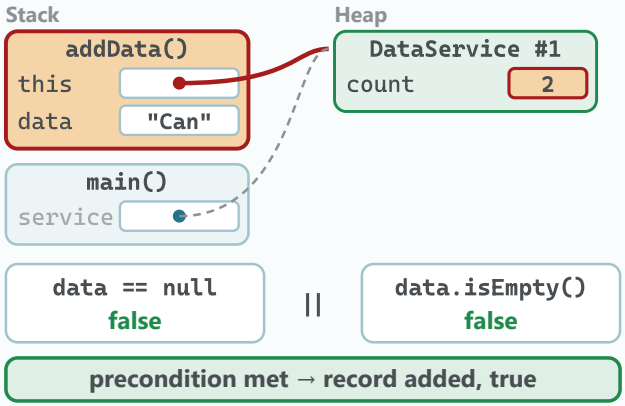
Step 6 / 8

OUTPUT

```
true
false
false
```

Contract: Code Trace

```
1  public boolean addData(String data) {
2      if (data == null || data.isEmpty()) {
3          return false;   // precondition broken
4      }
▶      count++;
▶      return true;        // postcondition met
7  }
8  // in main:
9  DataService service = new DataService();
10 System.out.println(service.addData("Ayşe"));
11 System.out.println(service.addData(null));
12 System.out.println(service.addData(""));
▶  System.out.println(service.addData("Can"));
14 System.out.println(service.getCount());
```



Valid data: the counter becomes 2, **true**.

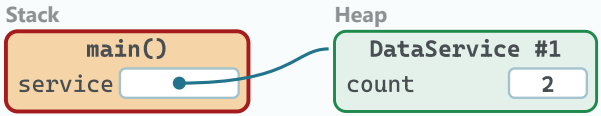
Step 7 / 8

OUTPUT

```
true
false
false
true
```

Contract: Code Trace

```
1  public boolean addData(String data) {
2      if (data == null || data.isEmpty()) {
3          return false;   // precondition broken
4      }
5      count++;
6      return true;        // postcondition met
7  }
8  // in main:
9  DataService service = new DataService();
10 System.out.println(service.addData("Ayşe"));
11 System.out.println(service.addData(null));
12 System.out.println(service.addData(""));
13 System.out.println(service.addData("Can"));
▶  System.out.println(service.getCount());
```



Of the four calls, only the two that honor the contract added a record: **2**. The object's state was never broken.

Step 8 / 8

OUTPUT

```
true
false
false
true
2
```

What If the Precondition Is Broken?

Approach	Example	What does the caller notice?
Ignore silently	`setSecond(75)` does nothing	Nothing; the bug may go unnoticed
Return a status	`addData(null)` → `false`	Only if it checks the return value
Throw an exception	`throw new IllegalArgumentException(...)`	Uncaught, it stops the program; the error is not lost

```java
public void setSecond(int second) {
    if (second < 0 || second > 59)
        throw new IllegalArgumentException(
            "second must be 0-59: " + second);
    this.second = second;
}
// in main:
DigitalClock s = new DigitalClock();
try {
    s.setSecond(30);
    s.setSecond(75);
    System.out.println("never reached");
} catch (IllegalArgumentException e) {
    System.out.println("Error: " + e.getMessage());
}
System.out.println(s.getSecond());
```

```
Error: second must be 0-59: 75
30
```

- The exception exposes the bad value **at the call site**; the object's state does not change (it stays 30).
- `public` methods: Java's usual choice for a bad argument is `IllegalArgumentException`.
- Document the chosen approach as part of the contract.

Writing the Contract with Javadoc

```java
/**
 * Withdraws money from the account.
 * @param amount sum to withdraw; {@code 0 < amount <= balance}
 * @return the balance after the call (old balance - amount)
 * @throws IllegalArgumentException if amount is out of range
 */
public double withdraw(double amount) {
    if (amount <= 0 || amount > balance)
        throw new IllegalArgumentException("Invalid amount: " + amount);
    balance -= amount;
    return balance;
}
```

Tags

`@param` : each parameter and its precondition.
`@return` : the return value, i.e. the postcondition.
`@throws` : what happens when the contract is broken.

Generating docs

`javadoc -d docs BankAccount.java` generates HTML docs from these comments. The Java library's online docs are built the same way.

What goes in?

Write **what** the method does and how to use it. **How** it does it stays out: that is the hidden implementation, and it may change.

For whom?

For the user, who never sees the body. A class's interface is its signatures plus these comments.

`assert`: Checking Invariants

```
class Counter {
    private int value;   // invariant: 0 <= value <= 9
    void increment() {
        value = value + 1;   // bug: 10 after 9
        assert value >= 0 && value <= 9 : "value=" + value;
    }
}
// in main:
Counter s = new Counter();
for (int i = 0; i < 10; i++) s.increment();
System.out.println("done");
```

`java Main`

```
done
```

`java -ea Main`

```
Exception in thread "main" java.lang.AssertionError:
value=10
        at Counter.increment(Main.java:5)
```

- `assert condition : message;` throws an `AssertionError` if the condition is false. It checks class invariants and postconditions **during development**.
- Assertions are **off by default**; `-ea` (enable assertions) turns them on. When they are off, the bug slips by silently: `done`.
- `public` method arguments are not checked with `assert`: when assertions are off, nothing is checked. Exceptions for arguments; for internal consistency, `assert`.

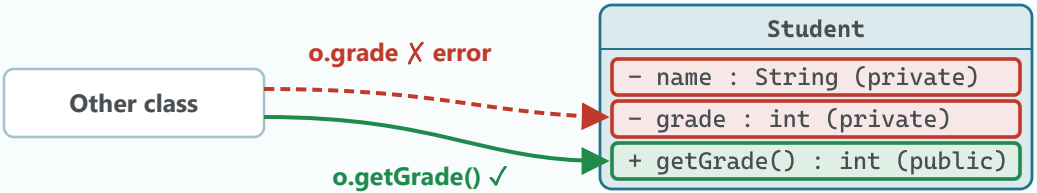
Access Modifiers

- **Access modifier**: a keyword that sets **who** can access a class, field or method.
- `public` : other classes can access it directly.
- `private` : accessible only from **within** the class; the compiler blocks outside access.
- Classes are usually `public` .
- Fields are usually `private` .
- Fields are read and changed through `public` **methods**.

```
public class Student {       // public class
    private String name;       // hidden field
    private int grade;         // hidden field

    public int getGrade() {    // public method
        return grade;
    }
}
```

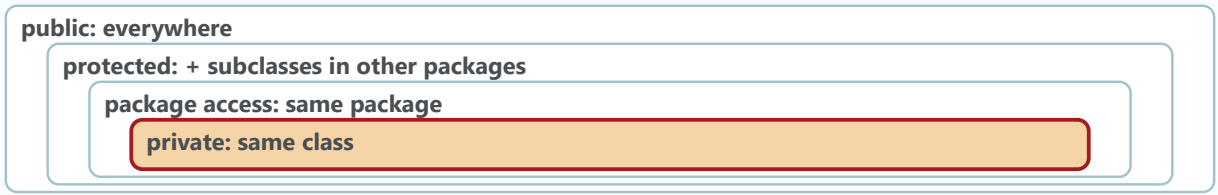
Rule: **fields hidden, behavior public**.



Four Access Levels

Modifier	Same class	Same package	Subclass (other package)	Everywhere
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
(none) package access	✓	✓	✗	✗
`private`	✓	✗	✗	✗

- With no modifier, **package access** (package-private, "default") applies: only classes in the same package have access.
- `protected` is for subclasses: a subclass in another package also has access.
- As a rule, prefer the most restrictive level: start with `private`, and open up only when truly needed.
- A top-level class can only be `public` or package-private.

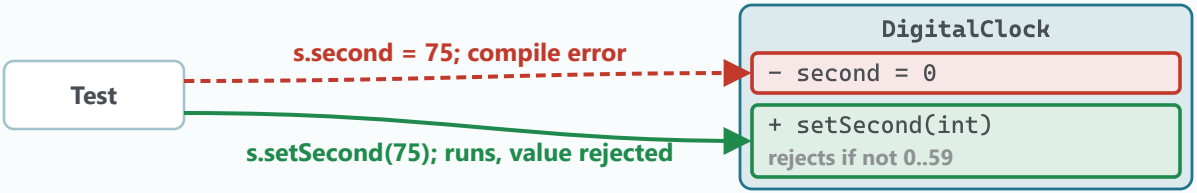


Trying to Access a private Field

```
public class Test {
    public static void main(String[] args) {
        DigitalClock s = new DigitalClock();
        s.second = 75;
    }
}
```

```
Test.java:4: error: second has private access
in DigitalClock
        s.second = 75;
         ^
1 error
```

- `second` is declared `private` in `DigitalClock`.
- Writing `s.second` from another class is a **compile error**: the program never runs.
- If the field were `public`, `s.second = 75;` would be silently accepted and the clock would fall into an **invalid** state.
- The fix: let the class's own method make the change: `s.setSecond(75);` The method rejects the invalid value.



`private` Is Per Class

```
class Ratio {
    private int num, den;
    Ratio(int num, int den) {
        this.num = num;
        this.den = den;
    }
    Ratio times(Ratio other) {
        // other's private fields: allowed
        return new Ratio(num * other.num,
                         den * other.den);
    }
    @Override public String toString() {
        return num + "/" + den;
    }
}
// in main:
Ratio k = new Ratio(2, 3).times(new Ratio(3, 4));
System.out.println(k);
```

```
6/12
```

- Access control works **per class, not per object**: `Ratio`'s code can read **every** `Ratio` object's `private` fields.
- `times`, `equals`, `compareTo`: methods that work on two objects need no getters for this reason.
- In another class, the same line does not compile:

```
error: num has private access in Ratio
```

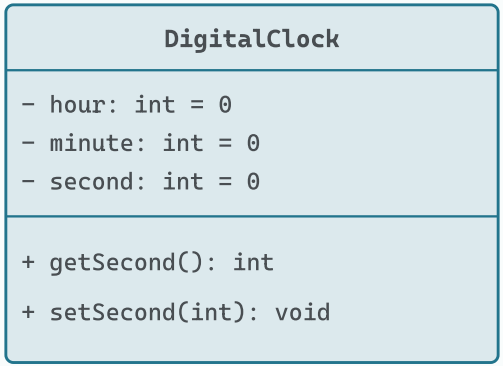
Encapsulation's boundary is the class's source code. The inside is one author's responsibility; it is the outside users who are kept out.

The DigitalClock Class

```java
public class DigitalClock {
    private int hour = 0;
    private int minute = 0;
    private int second = 0;

    public int getSecond() {
        return this.second;
    }

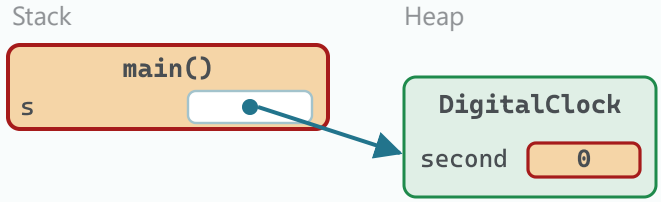
    public void setSecond(int second) {
        if (second >= 0 && second < 60) {
            this.second = second;
        }
    }
}
```



- **Getter** (accessor): returns the field's value.
- **Setter** (mutator): checks the new value **first**, and assigns it if valid.
- In UML, `-` means private, `+` means public.

Setter Protection: Step by Step

```
1  public class DigitalClock {
2    private int second = 0;
3    public int getSecond() {
4      return this.second;
5    }
6    public void setSecond(int second) {
7      if (second >= 0 && second < 60) {
8        this.second = second;
9      }
10   }
11 }
12 // in ClockTest.main:
▶ DigitalClock s = new DigitalClock();
14 s.setSecond(45);
15 s.setSecond(75);
16 System.out.println(s.getSecond());
```



The object is created; its `second` field starts at **0**. `s` points to the object.

Step 1 / 6

OUTPUT

CALL STACK

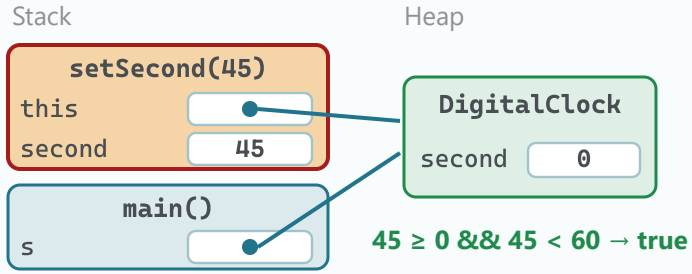
`main()`

VARIABLES

s	→DigitalCloc
=	k

Setter Protection: Step by Step

```
1  public class DigitalClock {
2    private int second = 0;
3    public int getSecond() {
4      return this.second;
5    }
6    public void setSecond(int second) {
▶      if (second >= 0 && second < 60) {
8        this.second = second;
9      }
10   }
11 }
12 // in ClockTest.main:
13 DigitalClock s = new DigitalClock();
14 s.setSecond(45);
15 s.setSecond(75);
16 System.out.println(s.getSecond());
```



`setSecond(45)` : parameter `second` = 45. `this` is the object the method was called on. The condition is true.

Step 2 / 6

OUTPUT

CALL STACK

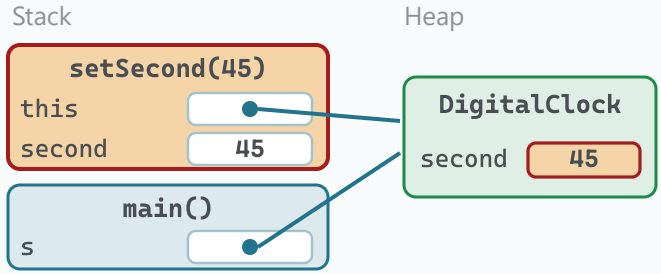
- `s.setSecond(45)`
- `main()`

VARIABLES

- `this = →object`
- `second = 45`

Setter Protection: Step by Step

```
1  public class DigitalClock {
2    private int second = 0;
3    public int getSecond() {
4      return this.second;
5    }
6    public void setSecond(int second) {
7      if (second >= 0 && second < 60) {
▶        this.second = second;
9      }
10   }
11 }
12 // in ClockTest.main:
13 DigitalClock s = new DigitalClock();
14 s.setSecond(45);
15 s.setSecond(75);
16 System.out.println(s.getSecond());
```



In `this.second = second;` the left `second` **is the object's field**, the right one **is the parameter**. The field becomes 45.

Step 3 / 6

OUTPUT

CALL STACK

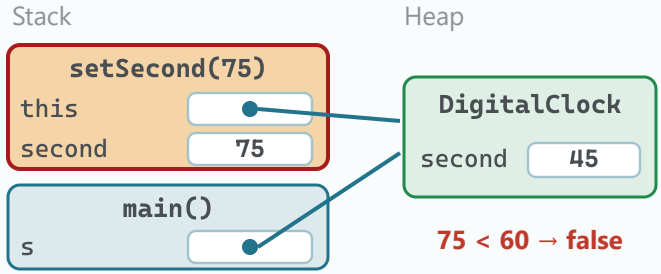
- `s.setSecond(45)`
- `main()`

VARIABLES

- `this = →object`
- `second = 45`

Setter Protection: Step by Step

```
1  public class DigitalClock {
2    private int second = 0;
3    public int getSecond() {
4      return this.second;
5    }
6    public void setSecond(int second) {
▶      if (second >= 0 && second < 60) {
8        this.second = second;
9      }
10   }
11 }
12 // in ClockTest.main:
13 DigitalClock s = new DigitalClock();
14 s.setSecond(45);
15 s.setSecond(75);
16 System.out.println(s.getSecond());
```



`setSecond(75)`: the condition is **false**, so the assignment is skipped. The invalid value cannot **get in**.

Step 4 / 6

OUTPUT

CALL STACK

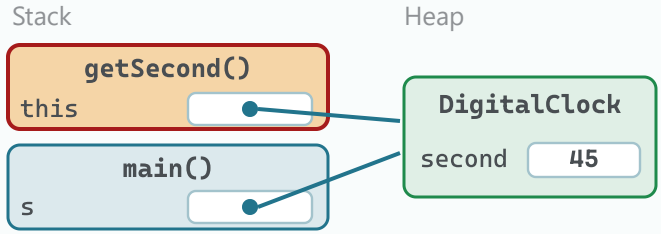
- `s.setSecond(75)`
- `main()`

VARIABLES

- `this = →object`
- `second = 75`

Setter Protection: Step by Step

```
1  public class DigitalClock {
2    private int second = 0;
3    public int getSecond() {
▶      return this.second;
5    }
6    public void setSecond(int second) {
7      if (second >= 0 && second < 60) {
8        this.second = second;
9      }
10   }
11 }
12 // in ClockTest.main:
13 DigitalClock s = new DigitalClock();
14 s.setSecond(45);
15 s.setSecond(75);
16 System.out.println(s.getSecond());
```



`getSecond()` returns the field's value: 45. The field is still valid.

Step 5 / 6

OUTPUT

CALL STACK

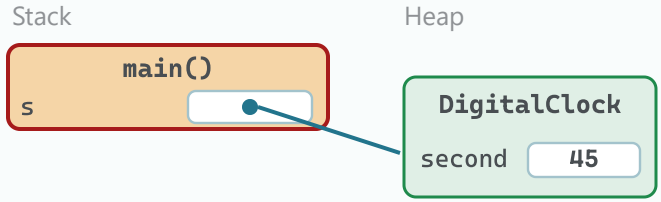
- `s.getSecond()`
- `main()`

VARIABLES

- `this = →object`
- `return = 45`

Setter Protection: Step by Step

```
1  public class DigitalClock {
2    private int second = 0;
3    public int getSecond() {
4      return this.second;
5    }
6    public void setSecond(int second) {
7      if (second >= 0 && second < 60) {
8        this.second = second;
9      }
10   }
11 }
12 // in ClockTest.main:
13 DigitalClock s = new DigitalClock();
14 s.setSecond(45);
15 s.setSecond(75);
▶  System.out.println(s.getSecond());
```



It prints **45**. The object's state stayed **valid** from start to finish: that is the point of encapsulation.

Step 6 / 6

OUTPUT

```
45
```

CALL STACK

```
main()
```

VARIABLES

s	→DigitalCloc
=	k

this and Shadowing

Correct

```
public void setSecond(int second) {
    if (second >= 0 && second < 60) {
        this.second = second;
    }
}
```

Wrong (but compiles!)

```
public class Box {
    private int width;
    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        width = width;
    }
}
```

- If a parameter has the same name as a field, then inside the method `second` **means the parameter**: the parameter **shadows** the field.
- `this` is the **very object the method was called on**. `this.second` is always the field.
- In the wrong version, `width = width;` assigns the parameter to itself; the field never changes. `k.setWidth(5)`, then `k.getWidth()`:

OUTPUT

```
0
```

The compiler reports nothing; the bug shows up only at run time. That is why the `this.` habit matters in setters.

Accessor Methods

```java
public class DigitalClock {
  private int hour = 0;       // 0..23
  private int minute = 0;     // 0..59
  private int second = 0;     // 0..59

  public int getHour() { return hour; }
  public int getMinute() { return minute; }
  public int getSecond() { return second; }

  public void setHour(int hour) {
    if (hour >= 0 && hour < 24) this.hour = hour;
  }
  public void setMinute(int minute) {
    if (minute >= 0 && minute < 60) this.minute = minute;
  }
  public void setSecond(int second) {
    if (second >= 0 && second < 60) this.second = second;
  }

  public void show() {
    System.out.printf("%02d:%02d:%02d%n",
        hour, minute, second);
  }
}
```

```java
DigitalClock s = new DigitalClock();
s.setHour(9);
s.setMinute(5);
s.setSecond(30);
s.show();
s.setHour(24);     // rejected
s.setMinute(-3);   // rejected
s.show();
```

OUTPUT

```
09:05:30
09:05:30
```

- Naming: `minute` → `getMinute()`, `setMinute(int)`.
- `boolean` fields use an accessor starting with `is`: `isActive()`.

Why Not a public Field?

public field

```
public class Clock {
    public int second;
}
// in another class:
s.second = -5;   // accepted!
```

- Any code can write any value.
- The rule (0–59) is left to every user; one slip and the object is broken.
- If the field's name or type changes, **every** class that uses it must change.

private field + method

```
public class Clock {
    private int second;
    public void setSecond(int s) {
        if (s >= 0 && s < 60) second = s;
    }
}
```

+ The rule is checked **in one place**, inside the class.
+ Read-only fields are possible: write only a getter.
+ The internals may change while method signatures stay the same.

Java 16+: for immutable classes that only carry data, you can write `record Point(int x, int y) { }`; the fields automatically become `private final`, and the accessors are named `x()` and `y()`.

Setter Protection: Interactive Demo

field second value 75 set via setter if field were public reset

```
s.setHour(9); s.setMinute(5); s.setSecond(30);
```

hour	minute	second
9	5	30

09:05:30

Pick a field and a value, then compare the two paths.

“set via setter” calls `s.setSecond(75)` : an invalid value is rejected. “if field were public” writes directly, as in `s.second = 75` : the object falls into an invalid state (red). Valid ranges: hour 0–23, minute and second 0–59.

BankAccount: Fields

```
public class BankAccount {
  private int accountNumber;  // account number
  private double balance;      // account balance
  private String accountType;   // "Checking" / "Savings"

  public BankAccount(int accountNumber,
      double balance, String accountType) {
    this.accountNumber = accountNumber;
    this.balance = balance;
    this.accountType = accountType;
  }

  public int getAccountNumber() {
    return accountNumber;
  }
  public double getBalance() {
    return balance;
  }
  public String getAccountType() {
    return accountType;
  }
  // ... deposit and withdraw methods
```

- All three fields are `private`: nobody can change the account number or balance directly.
- **The constructor** sets up the object with its initial values, using the `this.balance = balance` idiom.
- Getters only, no setters: `setBalance` does not exist, so the balance changes only through **transactions**.

EXAMPLE: BANK ACCOUNT

BankAccount: Deposit and Withdraw

```
// ...
// deposit
public void deposit(double amount) {
  if (amount > 0) {
    balance += amount;
    System.out.println(amount
        + " TL deposited. New balance: " + balance);
  } else {
    System.out.println("Invalid deposit amount.");
  }
}

// withdraw
public void withdraw(double amount) {
  if (amount <= balance && amount > 0) {
    balance -= amount;
    System.out.println(amount
        + " TL withdrawn. New balance: " + balance);
  } else if (amount > balance) {
    System.out.println("Insufficient balance.");
  } else {
    System.out.println("Invalid withdrawal amount.");
  }
}
```

- Each operation first checks its **precondition**: `amount > 0`, and for withdrawals also `amount ≤ balance`.
- If the condition fails, the balance **stays the same**; only a message is printed.
- The balance can never go negative: this is the class's **invariant**.

BankAccount: Interest

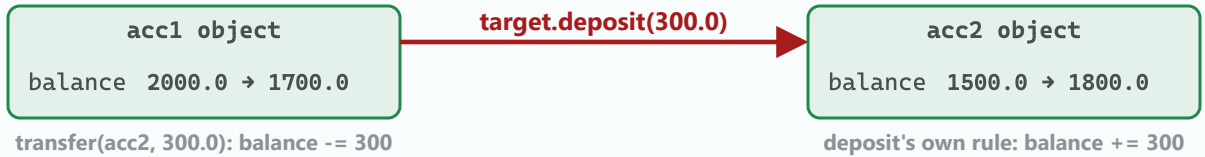
```
// ...
// add interest (savings only)
public void addInterest(double rate) {
  if (accountType.equals("Savings") && rate > 0) {
    double interest = balance * rate / 100;
    balance += interest;
    System.out.println("Adding interest: " + interest
        + " TL. New balance: " + balance);
  } else if (!accountType.equals("Savings")) {
    System.out.println(
        "Interest applies only to savings accounts.");
  } else {
    System.out.println("Invalid interest rate.");
  }
}
```

- Interest is added only to savings accounts, at a positive rate.
- Strings are compared not with `==`, but with `equals`.
- `interest` is a **local** variable: it exists only while the method runs and is not part of the object.

BankAccount: Transfer

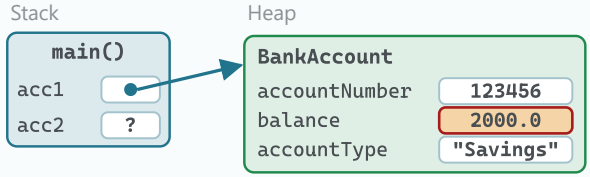
```
  // ...
  // transfer to another account
  public void transfer(BankAccount target,
                       double amount) {
    if (amount <= balance && amount > 0) {
      balance -= amount;
      target.deposit(amount);
      System.out.println(amount
          + " TL transferred. New balance: " + balance);
    } else {
      System.out.println(
          "Insufficient balance or invalid amount.");
    }
  }
}
```

- `transfer` calls a method on **another object** of the same class: `target`'s `deposit`.
- Writing directly to the target's `balance` field would also work (same class, and `private` is per class), but calling a method keeps the rules in one place.



Two Accounts, One Transfer: Step by Step

```
1  // in main:
▶  BankAccount acc1 =
▶      new BankAccount(123456, 2000.0, "Savings");
4  BankAccount acc2 =
5      new BankAccount(789012, 1500.0,
   "Checking");
6  acc1.deposit(500.0);
7  acc2.withdraw(200.0);
8  acc1.addInterest(5.0);
9  acc1.transfer(acc2, 300.0);
10 System.out.println(acc1.getBalance());
11 System.out.println(acc2.getBalance());
```



The constructor runs: the first object is built on the heap with balance 2000.0.

Step 1 / 9

OUTPUT

CALL STACK

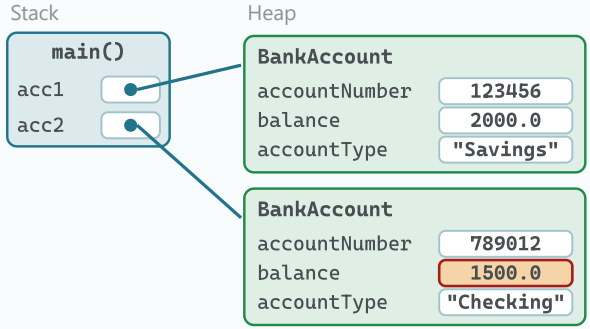
`main()`

VARIABLES

`acc1 = →BankAccount`

Two Accounts, One Transfer: Step by Step

```
1  // in main:
2  BankAccount acc1 =
3      new BankAccount(123456, 2000.0, "Savings");
▶  BankAccount acc2 =
▶      new BankAccount(789012, 1500.0,
   "Checking");
6  acc1.deposit(500.0);
7  acc2.withdraw(200.0);
8  acc1.addInterest(5.0);
9  acc1.transfer(acc2, 300.0);
10 System.out.println(acc1.getBalance());
11 System.out.println(acc2.getBalance());
```



A second, **independent** object. Each object has its own `balance` field.

Step 2 / 9

OUTPUT

CALL STACK

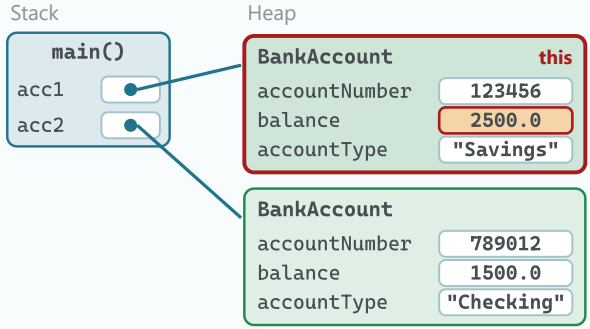
`main()`

VARIABLES

`acc1 = →BankAccount` `acc2 = →BankAccount`

Two Accounts, One Transfer: Step by Step

```
1  // in main:
2  BankAccount acc1 =
3      new BankAccount(123456, 2000.0, "Savings");
4  BankAccount acc2 =
5      new BankAccount(789012, 1500.0,
   "Checking");
▶  acc1.deposit(500.0);
7  acc2.withdraw(200.0);
8  acc1.addInterest(5.0);
9  acc1.transfer(acc2, 300.0);
10 System.out.println(acc1.getBalance());
11 System.out.println(acc2.getBalance());
```



`this` = acc1. `amount > 0` is true: `balance += amount` → 2500.0.

Step 3 / 9

OUTPUT

```
500.0 TL deposited. New balance: 2500.0
```

CALL STACK

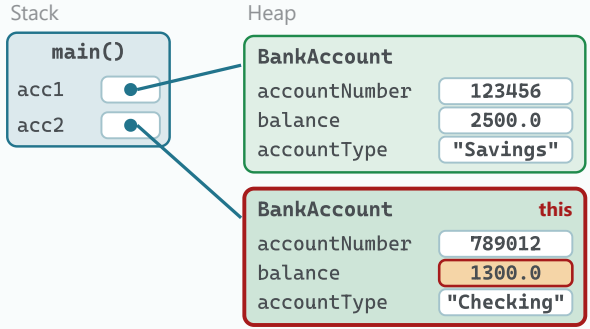
- `acc1.deposit(500.0)`
- `main()`

VARIABLES

`amount = 500.0`

Two Accounts, One Transfer: Step by Step

```
1  // in main:
2  BankAccount acc1 =
3      new BankAccount(123456, 2000.0, "Savings");
4  BankAccount acc2 =
5      new BankAccount(789012, 1500.0,
   "Checking");
6  acc1.deposit(500.0);
▶  acc2.withdraw(200.0);
8  acc1.addInterest(5.0);
9  acc1.transfer(acc2, 300.0);
10 System.out.println(acc1.getBalance());
11 System.out.println(acc2.getBalance());
```



This time `this` = acc2. 200 ≤ 1500: the balance becomes 1300.0; acc1 is unaffected.

Step 4 / 9

OUTPUT

```
500.0 TL deposited. New balance: 2500.0
200.0 TL withdrawn. New balance: 1300.0
```

CALL STACK

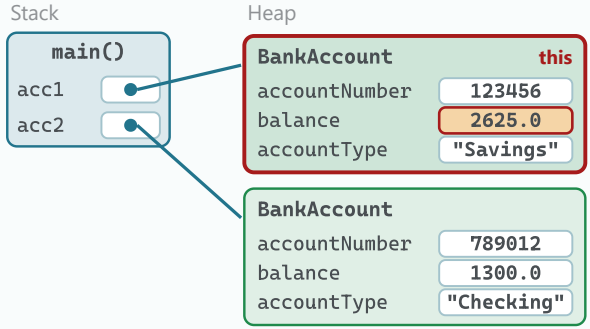
- `acc2.withdraw(200.0)`
- `main()`

VARIABLES

`amount = 200.0`

Two Accounts, One Transfer: Step by Step

```
1  // in main:
2  BankAccount acc1 =
3      new BankAccount(123456, 2000.0, "Savings");
4  BankAccount acc2 =
5      new BankAccount(789012, 1500.0,
   "Checking");
6  acc1.deposit(500.0);
7  acc2.withdraw(200.0);
▶  acc1.addInterest(5.0);
9  acc1.transfer(acc2, 300.0);
10 System.out.println(acc1.getBalance());
11 System.out.println(acc2.getBalance());
```



A savings account: `interest = 2500.0 × 5 / 100 = 125.0`, balance 2625.0.

Step 5 / 9

OUTPUT

```
500.0 TL deposited. New balance: 2500.0
200.0 TL withdrawn. New balance: 1300.0
Adding interest: 125.0 TL. New balance: 2625.0
```

CALL STACK

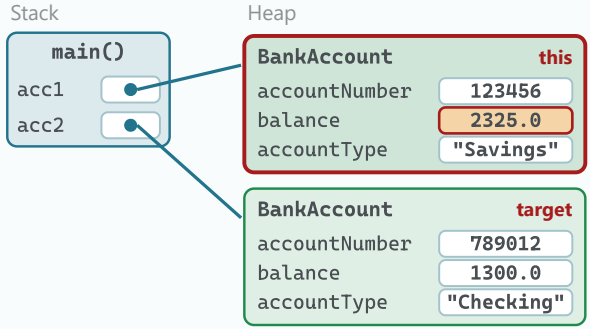
- `acc1.addInterest(5.0)`
- `main()`

VARIABLES

`rate = 5.0` `interest = 125.0`

Two Accounts, One Transfer: Step by Step

```
1  // in main:
2  BankAccount acc1 =
3      new BankAccount(123456, 2000.0, "Savings");
4  BankAccount acc2 =
5      new BankAccount(789012, 1500.0,
   "Checking");
6  acc1.deposit(500.0);
7  acc2.withdraw(200.0);
8  acc1.addInterest(5.0);
▶  acc1.transfer(acc2, 300.0);
10 System.out.println(acc1.getBalance());
11 System.out.println(acc2.getBalance());
```



`transfer` : the condition is true, so it first deducts from its own balance: 2625.0 − 300.0 = 2325.0.

Step 6 / 9

OUTPUT

```
500.0 TL deposited. New balance: 2500.0
200.0 TL withdrawn. New balance: 1300.0
Adding interest: 125.0 TL. New balance: 2625.0
```

CALL STACK

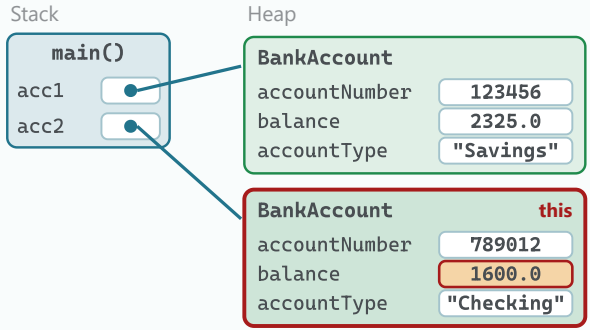
- `acc1.transfer(acc2, 300.0)`
- `main()`

VARIABLES

`amount = 300.0` `target = →acc2`

Two Accounts, One Transfer: Step by Step

```
1  // in main:
2  BankAccount acc1 =
3      new BankAccount(123456, 2000.0, "Savings");
4  BankAccount acc2 =
5      new BankAccount(789012, 1500.0,
   "Checking");
6  acc1.deposit(500.0);
7  acc2.withdraw(200.0);
8  acc1.addInterest(5.0);
▶  acc1.transfer(acc2, 300.0);
10 System.out.println(acc1.getBalance());
11 System.out.println(acc2.getBalance());
```



`target.deposit(amount)` : 3 frames on the stack; now `this` = acc2, balance 1600.0.

Step 7 / 9

OUTPUT

```
500.0 TL deposited. New balance: 2500.0
200.0 TL withdrawn. New balance: 1300.0
Adding interest: 125.0 TL. New balance: 2625.0
300.0 TL deposited. New balance: 1600.0
```

CALL STACK

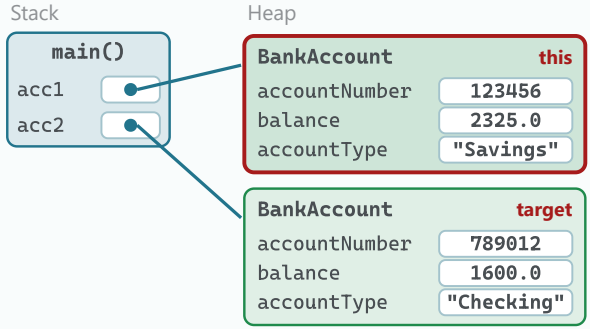
- **acc2.deposit(300.0)**
- acc1.transfer(acc2, 300.0)
- main()

VARIABLES

amount = **300.0**

Two Accounts, One Transfer: Step by Step

```
1  // in main:
2  BankAccount acc1 =
3      new BankAccount(123456, 2000.0, "Savings");
4  BankAccount acc2 =
5      new BankAccount(789012, 1500.0,
   "Checking");
6  acc1.deposit(500.0);
7  acc2.withdraw(200.0);
8  acc1.addInterest(5.0);
▶  acc1.transfer(acc2, 300.0);
10 System.out.println(acc1.getBalance());
11 System.out.println(acc2.getBalance());
```



`deposit` returns; `transfer` resumes where it left off and prints its own message.

Step 8 / 9

OUTPUT

```
500.0 TL deposited. New balance: 2500.0
200.0 TL withdrawn. New balance: 1300.0
Adding interest: 125.0 TL. New balance: 2625.0
300.0 TL deposited. New balance: 1600.0
300.0 TL transferred. New balance: 2325.0
```

CALL STACK

`acc1.transfer(acc2, 300.0)`

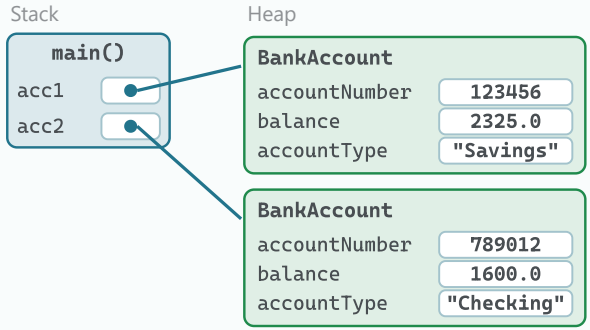
`main()`

VARIABLES

`amount = 300.0`

Two Accounts, One Transfer: Step by Step

```
1 // in main:
2 BankAccount acc1 =
3     new BankAccount(123456, 2000.0, "Savings");
4 BankAccount acc2 =
5     new BankAccount(789012, 1500.0,
  "Checking");
6 acc1.deposit(500.0);
7 acc2.withdraw(200.0);
8 acc1.addInterest(5.0);
9 acc1.transfer(acc2, 300.0);
▶ System.out.println(acc1.getBalance());
▶ System.out.println(acc2.getBalance());
```



The getters read the final balances. Total money is conserved: 2325 + 1600 = 3925 = 2625 + 1300.

Step 9 / 9

OUTPUT

```
500.0 TL deposited. New balance: 2500.0
200.0 TL withdrawn. New balance: 1300.0
Adding interest: 125.0 TL. New balance: 2625.0
300.0 TL deposited. New balance: 1600.0
300.0 TL transferred. New balance: 2325.0
2325.0
1600.0
```

CALL STACK

`main()`

VARIABLES

`acc1 = →BankAccount` `acc2 = →BankAccount`

EXAMPLE: BANK ACCOUNT

The Constructor Must Validate Too

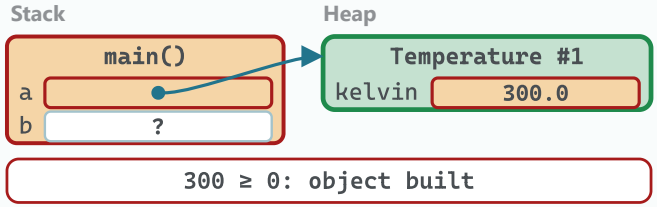
- Setters and transaction methods protect the balance, but the constructor in the example assigns its input **without checking**.
- `new BankAccount(1, -500.0, "Checking")` creates an account with a negative balance; write `"savings"` (lowercase) and the account never earns interest, with no one noticing.
- The class invariant must hold from **the moment the object is born**: the constructor is a precondition checker too.
- If the constructor throws, the `new` expression yields no reference: the invalid object never reaches the user.

```
public BankAccount(int accountNumber,
        double balance, String accountType) {
    if (balance < 0)
        throw new IllegalArgumentException(
            "Initial balance cannot be negative");
    if (!accountType.equals("Savings")
            && !accountType.equals("Checking"))
        throw new IllegalArgumentException(
            "Unknown account type: " + accountType);
    this.accountNumber = accountNumber;
    this.balance = balance;
    this.accountType = accountType;
}
```

The constructor is the only door into the object: if the rules are enforced there once, the class's other methods can trust that every object is valid.

Validation in the Constructor: Code Trace

```
 1 class Temperature {
 2     private final double kelvin;
 3     public Temperature(double kelvin) {
 4         if (kelvin < 0)
 5             throw new IllegalArgumentException(
 6                 "Kelvin cannot be negative");
 7         this.kelvin = kelvin;
 8     }
 9     public double getKelvin() { return kelvin; }
10 }
11 // in main:
▶ Temperature a = new Temperature(300);
13 Temperature b = null;
14 try {
15     b = new Temperature(-5);
16 } catch (IllegalArgumentException e) {
17     System.out.println("Rejected: "
18         + e.getMessage());
19 }
20 System.out.println(a.getKelvin() + " " + b);
```



`300 ≥ 0: object built`

300 is valid: the condition is false; `kelvin` is assigned and `a` points to the new object.

Step 1 / 6

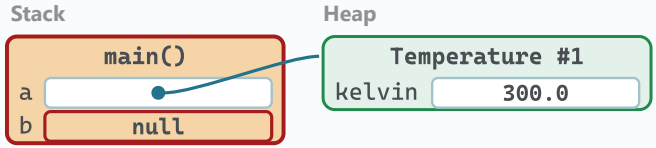
OUTPUT

VARIABLES

`a = →#1`

Validation in the Constructor: Code Trace

```
 1 class Temperature {
 2     private final double kelvin;
 3     public Temperature(double kelvin) {
 4         if (kelvin < 0)
 5             throw new IllegalArgumentException(
 6                 "Kelvin cannot be negative");
 7         this.kelvin = kelvin;
 8     }
 9     public double getKelvin() { return kelvin; }
10 }
11 // in main:
12 Temperature a = new Temperature(300);
▶ Temperature b = null;
14 try {
15     b = new Temperature(-5);
16 } catch (IllegalArgumentException e) {
17     System.out.println("Rejected: "
18         + e.getMessage());
19 }
20 System.out.println(a.getKelvin() + " " + b);
```



b points to no object yet

`b` is explicitly set to `null` so it can still be read after the `catch` block.

Step 2 / 6

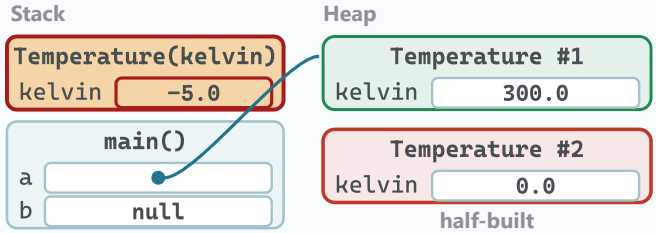
OUTPUT

VARIABLES

`a = →#1` `b = null`

Validation in the Constructor: Code Trace

```
 1 class Temperature {
 2     private final double kelvin;
 ▶     public Temperature(double kelvin) {
 4         if (kelvin < 0)
 5             throw new IllegalArgumentException(
 6                 "Kelvin cannot be negative");
 7         this.kelvin = kelvin;
 8     }
 9     public double getKelvin() { return kelvin; }
10 }
11 // in main:
12 Temperature a = new Temperature(300);
13 Temperature b = null;
14 try {
 ▶     b = new Temperature(-5);
16 } catch (IllegalArgumentException e) {
17     System.out.println("Rejected: "
18         + e.getMessage());
19 }
20 System.out.println(a.getKelvin() + " " + b);
```



`new allocated; constructor runs`

`new` allocates the second object (`kelvin` defaults to 0.0) and calls the constructor. The parameter is −5.0.

Step 3 / 6

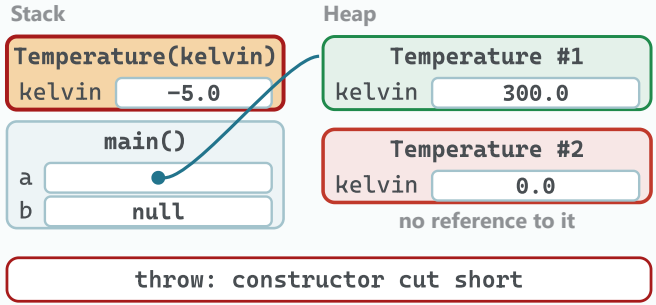
OUTPUT

VARIABLES

`a = →#1` `b = null` `kelvin = -5.0`

Validation in the Constructor: Code Trace

```
 1 class Temperature {
 2     private final double kelvin;
 3     public Temperature(double kelvin) {
 4         if (kelvin < 0)
 5             throw new IllegalArgumentException(
 6                 "Kelvin cannot be negative");
 7         this.kelvin = kelvin;
 8     }
 9     public double getKelvin() { return kelvin; }
10 }
11 // in main:
12 Temperature a = new Temperature(300);
13 Temperature b = null;
14 try {
15     b = new Temperature(-5);
16 } catch (IllegalArgumentException e) {
17     System.out.println("Rejected: "
18         + e.getMessage());
19 }
20 System.out.println(a.getKelvin() + " " + b);
```



Precondition broken: `throw` stops the constructor here. The assignment line never runs.

Step 4 / 6

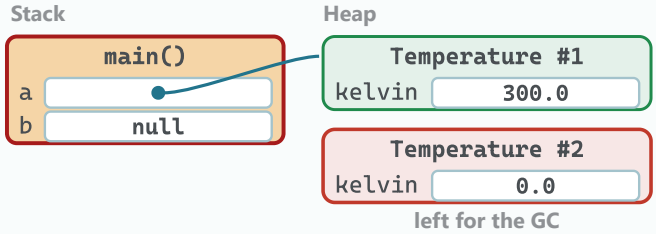
OUTPUT

VARIABLES

`a = →#1` `b = null` `kelvin = -5.0`

Validation in the Constructor: Code Trace

```
 1 class Temperature {
 2     private final double kelvin;
 3     public Temperature(double kelvin) {
 4         if (kelvin < 0)
 5             throw new IllegalArgumentException(
 6                 "Kelvin cannot be negative");
 7         this.kelvin = kelvin;
 8     }
 9     public double getKelvin() { return kelvin; }
10 }
11 // in main:
12 Temperature a = new Temperature(300);
13 Temperature b = null;
14 try {
15     b = new Temperature(-5);
▶ } catch (IllegalArgumentException e) {
▶     System.out.println("Rejected: "
▶         + e.getMessage());
19 }
20 System.out.println(a.getKelvin() + " " + b);
```



no assignment: b is still null

Control passes to the `catch` block. `b = ...` was cut short, so `b` did not change; no reference reaches the half-built object.

Step 5 / 6

OUTPUT

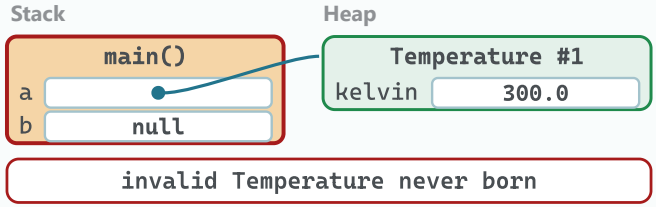
```
Rejected: Kelvin cannot be negative
```

VARIABLES

`a = →#1` `b = null`

Validation in the Constructor: Code Trace

```
 1 class Temperature {
 2     private final double kelvin;
 3     public Temperature(double kelvin) {
 4         if (kelvin < 0)
 5             throw new IllegalArgumentException(
 6                 "Kelvin cannot be negative");
 7         this.kelvin = kelvin;
 8     }
 9     public double getKelvin() { return kelvin; }
10 }
11 // in main:
12 Temperature a = new Temperature(300);
13 Temperature b = null;
14 try {
15     b = new Temperature(-5);
16 } catch (IllegalArgumentException e) {
17     System.out.println("Rejected: "
18         + e.getMessage());
19 }
▶ System.out.println(a.getKelvin() + " " + b);
```



The program can only use valid `Temperature` objects: the invariant (`kelvin ≥ 0`) holds for every object.

Step 6 / 6

OUTPUT

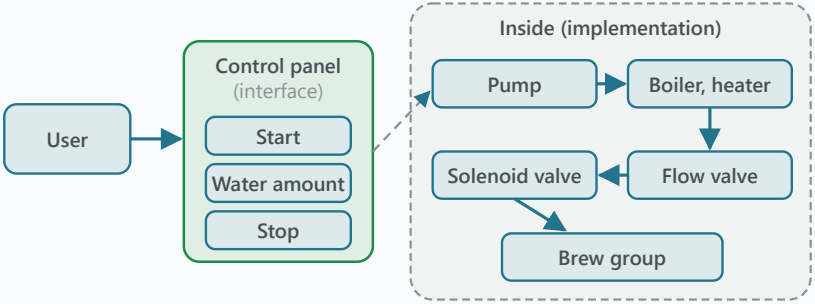
```
Rejected: Kelvin cannot be negative
300.0 null
```

VARIABLES

a = →#1 b = null

Class Design

- A class definition has two parts: **interface** and **implementation**.
- The interface states **what the class does** and **how to use it**: the names, parameters, return types and comments of the public methods.
- The implementation holds the class's internal mechanics and private fields, which matter only inside the class.
- **The implementation may change while the interface stays the same**; private fields are hidden from outside the class.
- Three parts: **specification** (constructor and method signatures), **representation** (the `private` fields that hold the data), and **implementation** (constructor and method bodies). Users know only the specification; their code survives changes to the representation and implementation.

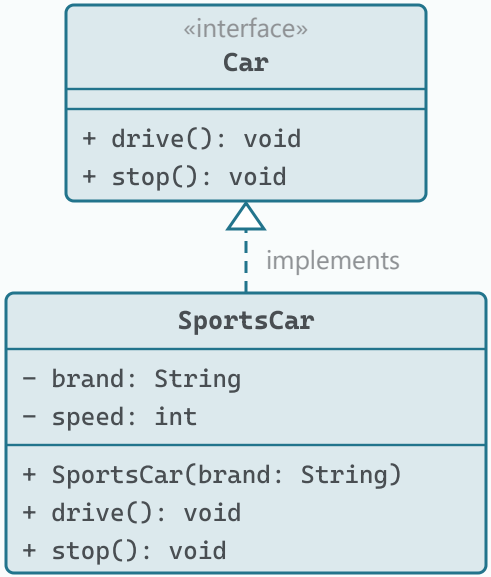


Interface

```java
public interface Car {
    // Interface: WHAT is done?
    public void drive();

    public void stop();
}
```

- In Java, an `interface` is a contract that says **what** a class will do: the methods have only signatures, no bodies.
- Whoever uses the car knows only the `drive()` and `stop()` methods.



Class Implementation

```java
public class SportsCar implements Car {
  // Implementation: HOW is it done?
  private String brand;
  private int speed;

  public SportsCar(String brand) {
    this.brand = brand;
  }

  public void drive() {
    speed = 100;
    System.out.println(brand + " is moving: "
        + speed + " km/h");
  }

  public void stop() {
    speed = 0;
    System.out.println(brand + " stopped.");
  }
}
```

```java
// in main:
Car car = new SportsCar("Rüzgar");
car.drive();
car.stop();
```

OUTPUT

```
Rüzgar is moving: 100 km/h
Rüzgar stopped.
```

- `implements Car` : the class must write **every** method of the interface.
- An interface and a class in the same package cannot share **a name** (`duplicate class: Car`); that is why the class is named `SportsCar` .
- `Main` declares the variable as type `Car` : it depends only on the interface.

Changing the Implementation

Old: two separate fields

```
public class DigitalClock {
  private int minute = 0;
  private int second = 0;
  public int getMinute() {
    return minute;
  }
  public int getSecond() {
    return second;
  }
  public void setMinute(int minute) {
    if (minute >= 0 && minute < 60) {
      this.minute = minute;
    }
  }
  public void setSecond(int second) {
    if (second >= 0 && second < 60) {
      this.second = second;
    }
  }
}
```

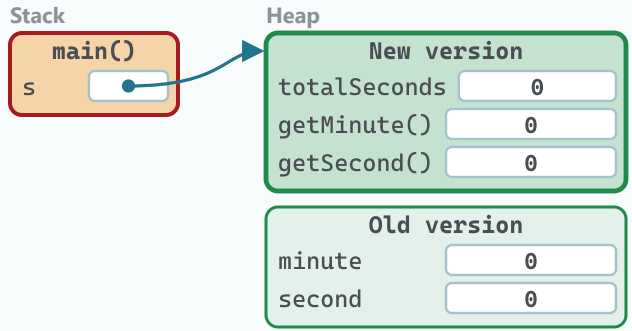
New: a single field

```
public class DigitalClock {
  // seconds since midnight
  private int totalSeconds = 0;
  public int getMinute() {
    return totalSeconds / 60 % 60;
  }
  public int getSecond() {
    return totalSeconds % 60;
  }
  public void setMinute(int minute) {
    if (minute >= 0 && minute < 60) {
      int diff = minute - getMinute();
      totalSeconds += diff * 60;
    }
  }
  public void setSecond(int second) {
    if (second >= 0 && second < 60) {
      totalSeconds += second - getSecond();
    }
  }
}
```

Client code is unchanged: after `setMinute(12); setSecond(45); setSecond(75);`, both versions give **12:45**.

Same Interface, New Implementation: Code Trace

```
 1 private int totalSeconds = 0;  // one field
 2 public int getSecond() {
 3     return totalSeconds % 60;
 4 }
 5 public void setSecond(int second) {
 6     if (second >= 0 && second < 60)
 7         totalSeconds += second - getSecond();
 8 }
 9 // getMinute(): totalSeconds / 60 % 60
▶DigitalClock s = new DigitalClock(); // main
11 s.setMinute(12);
12 s.setSecond(45);
13 s.setSecond(75);
14 s.setSecond(9);
15 System.out.println(s.getMinute() + ":"
16     + s.getSecond());
```



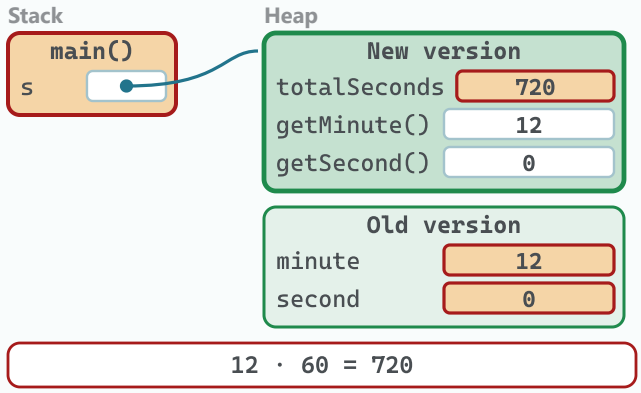
The new version keeps one field; the old one (for comparison) keeps two. From outside, both read 00:00.

Step 1 / 7

OUTPUT

Same Interface, New Implementation: Code Trace

```
1  private int totalSeconds = 0;  // one field
2  public int getSecond() {
3      return totalSeconds % 60;
4  }
5  public void setSecond(int second) {
6      if (second >= 0 && second < 60)
7          totalSeconds += second - getSecond();
8  }
9  // getMinute(): totalSeconds / 60 % 60
10 DigitalClock s = new DigitalClock(); // main
▶  s.setMinute(12);
12 s.setSecond(45);
13 s.setSecond(75);
14 s.setSecond(9);
15 System.out.println(s.getMinute() + ":"
16     + s.getSecond());
```



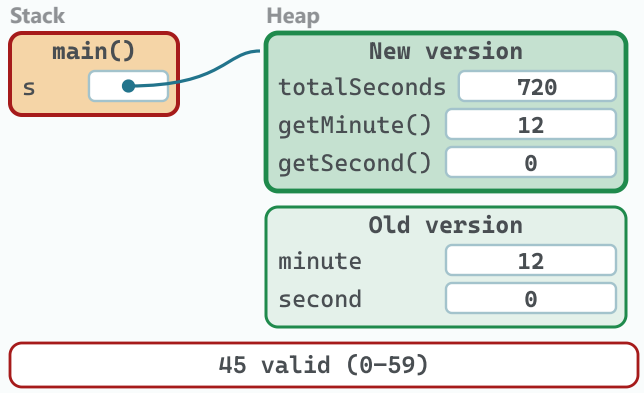
`setMinute(12)` : a 12-minute difference, so 12 · 60 = 720 seconds are added. The old version simply writes `minute = 12` .

Step 2 / 7

OUTPUT

Same Interface, New Implementation: Code Trace

```
1  private int totalSeconds = 0;  // one field
2  public int getSecond() {
3      return totalSeconds % 60;
4  }
▶  public void setSecond(int second) {
▶      if (second >= 0 && second < 60)
7          totalSeconds += second - getSecond();
8  }
9  // getMinute(): totalSeconds / 60 % 60
10 DigitalClock s = new DigitalClock(); // main
11 s.setMinute(12);
▶  s.setSecond(45);
13 s.setSecond(75);
14 s.setSecond(9);
15 System.out.println(s.getMinute() + ":"
16     + s.getSecond());
```



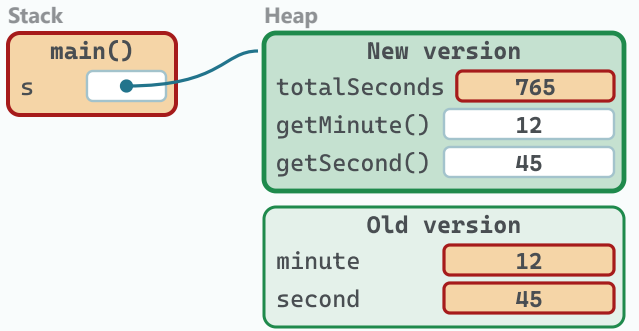
`setSecond(45)` : the value is in range, the condition is true.

Step 3 / 7

OUTPUT

Same Interface, New Implementation: Code Trace

```
1  private int totalSeconds = 0;  // one field
▶  public int getSecond() {
▶      return totalSeconds % 60;
4  }
5  public void setSecond(int second) {
6      if (second >= 0 && second < 60)
▶          totalSeconds += second - getSecond();
8  }
9  // getMinute(): totalSeconds / 60 % 60
10 DigitalClock s = new DigitalClock(); // main
11 s.setMinute(12);
12 s.setSecond(45);
13 s.setSecond(75);
14 s.setSecond(9);
15 System.out.println(s.getMinute() + ":"
16     + s.getSecond());
```



`720 + (45 − 720 % 60) = 720 + 45 = 765`

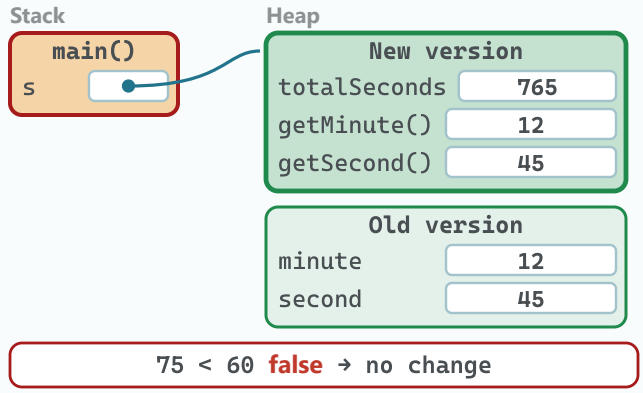
First, `getSecond()` = 720 % 60 = 0; the difference 45 is added: **765**. From outside it reads 12:45.

Step 4 / 7

OUTPUT

Same Interface, New Implementation: Code Trace

```
1  private int totalSeconds = 0;  // one field
2  public int getSecond() {
3      return totalSeconds % 60;
4  }
5  public void setSecond(int second) {
▶      if (second >= 0 && second < 60)
7          totalSeconds += second - getSecond();
8  }
9  // getMinute(): totalSeconds / 60 % 60
10 DigitalClock s = new DigitalClock(); // main
11 s.setMinute(12);
12 s.setSecond(45);
▶  s.setSecond(75);
14 s.setSecond(9);
15 System.out.println(s.getMinute() + ":"
16     + s.getSecond());
```



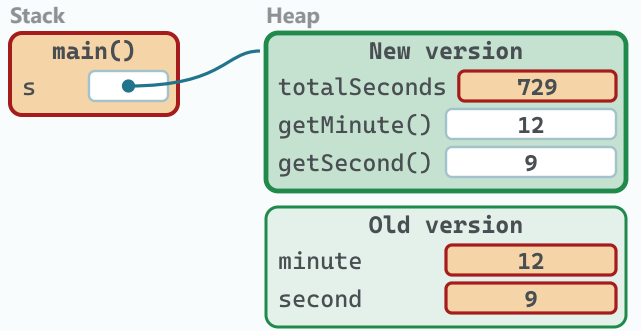
75 is invalid: both versions reject it. The rule (0–59) stayed inside the class.

Step 5 / 7

OUTPUT

Same Interface, New Implementation: Code Trace

```
 1 private int totalSeconds = 0;  // one field
 2 public int getSecond() {
 3     return totalSeconds % 60;
 4 }
 5 public void setSecond(int second) {
 6     if (second >= 0 && second < 60)
 ▶         totalSeconds += second - getSecond();
 8 }
 9 // getMinute(): totalSeconds / 60 % 60
10 DigitalClock s = new DigitalClock(); // main
11 s.setMinute(12);
12 s.setSecond(45);
13 s.setSecond(75);
 ▶ s.setSecond(9);
15 System.out.println(s.getMinute() + ":"
16     + s.getSecond());
```



`765 + (9 − 45) = 729`

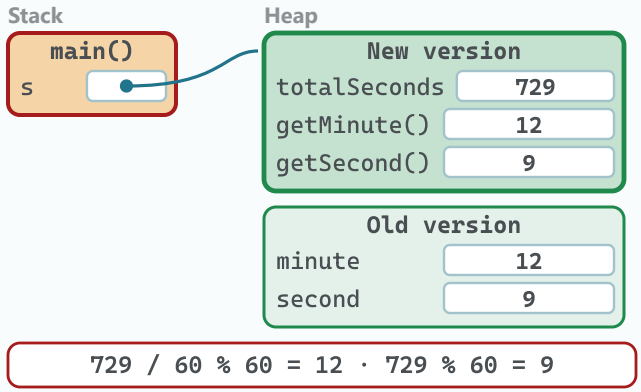
The difference can be negative: 9 − 45 = −36, total **729**. The minute is still 12.

Step 6 / 7

OUTPUT

Same Interface, New Implementation: Code Trace

```
1  private int totalSeconds = 0;  // one field
2  public int getSecond() {
3      return totalSeconds % 60;
4  }
5  public void setSecond(int second) {
6      if (second >= 0 && second < 60)
7          totalSeconds += second - getSecond();
8  }
9  // getMinute(): totalSeconds / 60 % 60
10 DigitalClock s = new DigitalClock(); // main
11 s.setMinute(12);
12 s.setSecond(45);
13 s.setSecond(75);
14 s.setSecond(9);
▶  System.out.println(s.getMinute() + ":"
▶      + s.getSecond());
```



`729 / 60 % 60 = 12 · 729 % 60 = 9`

Output **12:9**; the old version would give the same. The internal representation changed, yet not one character of `main` changed.

Step 7 / 7

OUTPUT

```
12:9
```

Guidelines

1. All variables (fields) in a class should be `private`.
2. Provide **only as many** `public` accessor methods as needed (you don't need an automatic setter for every field).
3. Add comments saying how to use each method: what it does and its pre- and postconditions.
4. Internal-only **helper methods should be** `private`.

```
public class Student {
    private int grade;
    private boolean isValid(int value) {
        return value >= 0 && value <= 100;
    }
    public void setGrade(int grade) {
        if (isValid(grade)) this.grade = grade;
    }
}
```

If another class calls `o.isValid(50)`:

```
error: isValid(int) has private access in
Student
```

CLASS DESIGN

Tell, Don't Ask

Writing a getter and a setter for every field **is not** encapsulation: if the rules are still enforced outside the class, the data has only moved one step away.

Asking: the rule lives in client code

```
// same check at every withdrawal
if (h.getBalance() >= 300) {
    h.setBalance(h.getBalance() - 300);
}
// ...and wherever points are added
o.setGrade(Math.min(100, o.getGrade() + 5));
```

- The rule repeats at every call site; one slip and the object is broken.
- `setBalance`: once it exists, anyone can set the balance to anything.

Telling: the rule lives in the object

```
h.withdraw(300);   // rule in BankAccount
o.addBonus(5);     // 100 cap in Student
```

- Instead of asking the object for its data and deciding for it, you **tell it** what you want; the object holding the data makes the decision.
- Write a setter only when the field can truly change freely. `BankAccount` therefore has no `setBalance`.

Test question: “Who has the data needed to make this decision?” If the answer is the object itself, the decision belongs in its method.

Benefits of Encapsulation

Abstraction

Simplifies the relation between an object and its user: the user knows only the methods.

Protection

Protects the object from unwanted, faulty access by users.

Modifiability

Without changing the interface, the class's internals can change; client code is unaffected.

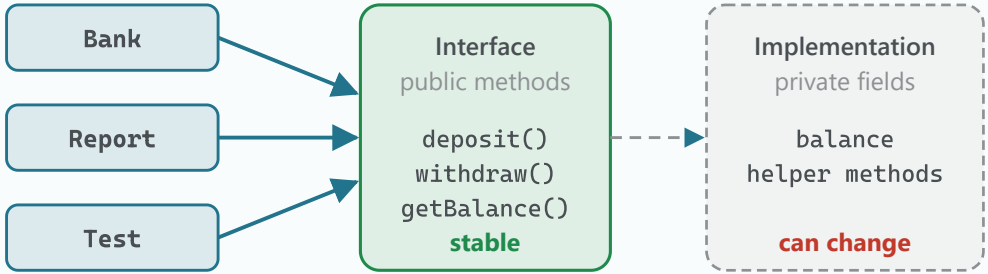
Valid state (invariant)

The object's state always obeys the rules (class invariant):

- only points with positive coordinates,
- seconds and minutes between 0 and 59,
- hours never negative.

In this chapter's examples: `setSecond(75)` is rejected, `withdraw` never takes the balance below zero, `BankAccount` 's internals may change, yet code calling `deposit` stays the same.

Observations



- Interfaces change much **less often** than implementations: this keeps the code **stable**.
- When the implementation changes, code that depends on it **directly** must change too.
- Hiding the implementation reduces other classes' dependence on it: changes stay inside the class (loose coupling).

A Leaking Reference

```
public class ReportCard {
    private final int[] grades;
    public ReportCard(int[] grades) {
        this.grades = grades; // same array
    }
    public int[] getGrades() {
        return grades;        // inner array
    }
    public double average() {
        int t = 0;
        for (int n : grades) t += n;
        return (double) t / grades.length;
    }
}
```

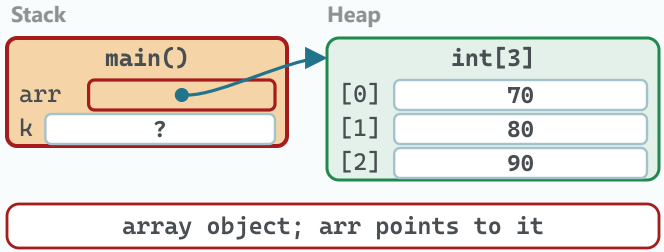
```
int[] arr = {70, 80, 90};
ReportCard k = new ReportCard(arr);
arr[0] = 0;                 // outside array
k.getGrades()[1] = -100;    // from the getter
System.out.println(k.average());
```

```
-3.3333333333333335
```

- The field is `private final`, with no setter; yet the grades changed from outside, and even an invalid grade like −100 got in.
- The reason: an array is an **object**. The constructor and getter shared only the **reference**; the object is reachable from three places.
- `final` only stops the reference from pointing to another array; it does not protect the array's **contents**.

A Leaking Reference: Code Trace

```
1  public ReportCard(int[] grades) {
2      this.grades = grades;
3  }
4  public int[] getGrades() {
5      return grades;
6  }
7  // in main:
▶ int[] arr = {70, 80, 90};
9  ReportCard k = new ReportCard(arr);
10 arr[0] = 0;
11 k.getGrades()[1] = -100;
12 System.out.println(k.average());
```



An array is an object on the heap; the `arr` variable holds only its reference.

Step 1 / 5

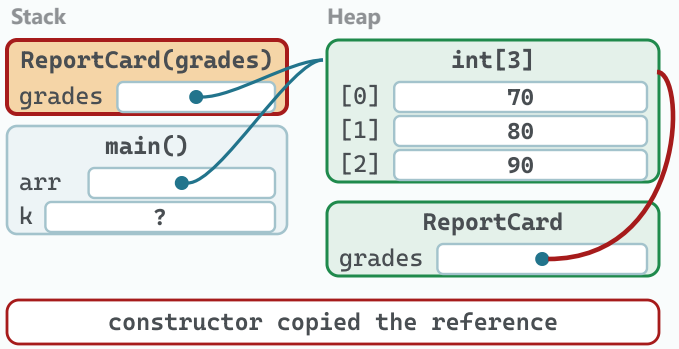
OUTPUT

VARIABLES

`arr = →[70, 80, 90]`

A Leaking Reference: Code Trace

```
1  public ReportCard(int[] grades) {
▶      this.grades = grades;
3  }
4  public int[] getGrades() {
5      return grades;
6  }
7  // in main:
8  int[] arr = {70, 80, 90};
▶  ReportCard k = new ReportCard(arr);
10 arr[0] = 0;
11 k.getGrades()[1] = -100;
12 System.out.println(k.average());
```



The constructor writes the **reference** from its parameter into the field. Now `ReportCard`'s `private` field and `main`'s `arr` variable point to the same array.

Step 2 / 5

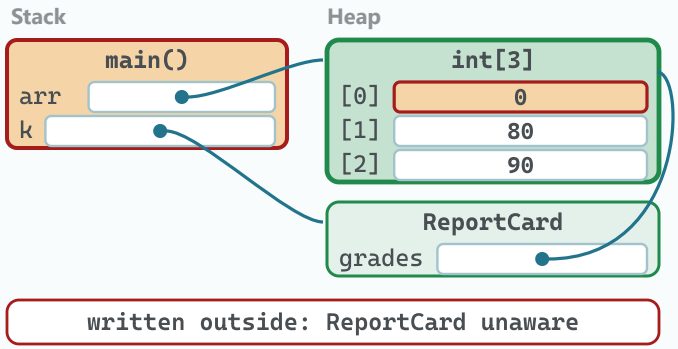
OUTPUT

VARIABLES

`arr = →[70, 80, 90]` `k = →ReportCard`

A Leaking Reference: Code Trace

```
1  public ReportCard(int[] grades) {
2      this.grades = grades;
3  }
4  public int[] getGrades() {
5      return grades;
6  }
7  // in main:
8  int[] arr = {70, 80, 90};
9  ReportCard k = new ReportCard(arr);
▶  arr[0] = 0;
11 k.getGrades()[1] = -100;
12 System.out.println(k.average());
```



`arr[0] = 0` goes through none of ReportCard's methods, yet the grade ReportCard sees has changed.

Step 3 / 5

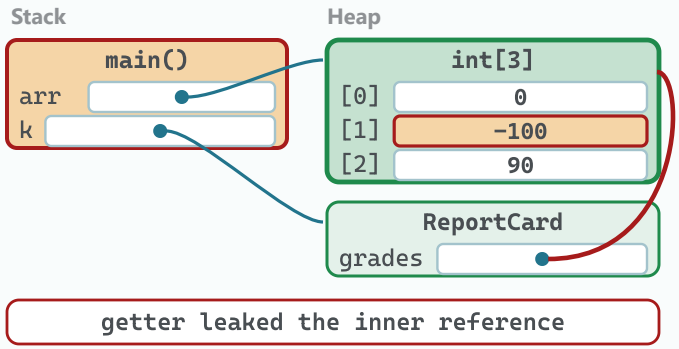
OUTPUT

VARIABLES

`arr = →[0, 80, 90]` `k = →ReportCard`

A Leaking Reference: Code Trace

```
1  public ReportCard(int[] grades) {
2      this.grades = grades;
3  }
4  public int[] getGrades() {
▶      return grades;
6  }
7  // in main:
8  int[] arr = {70, 80, 90};
9  ReportCard k = new ReportCard(arr);
10 arr[0] = 0;
▶ k.getGrades()[1] = -100;
12 System.out.println(k.average());
```



`getGrades()` returns the inner array itself; −100 is written through the returned reference. An invalid grade got into the object without passing any check.

Step 4 / 5

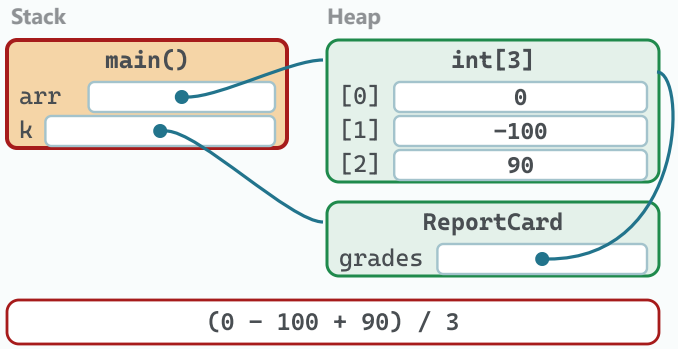
OUTPUT

VARIABLES

`arr = →[0, -100, 90]` `k = →ReportCard`

A Leaking Reference: Code Trace

```
1  public ReportCard(int[] grades) {
2      this.grades = grades;
3  }
4  public int[] getGrades() {
5      return grades;
6  }
7  // in main:
8  int[] arr = {70, 80, 90};
9  ReportCard k = new ReportCard(arr);
10 arr[0] = 0;
11 k.getGrades()[1] = -100;
▶  System.out.println(k.average());
```



Average (0 − 100 + 90) / 3 ≈ −3.33. `private` alone was not enough: the reference to the inner object **leaked out**.

Step 5 / 5

OUTPUT

```
-3.3333333333333335
```

VARIABLES

`arr = →[0, -100, 90]` `k = →ReportCard`

Defensive Copying

```
public class ReportCard {
    private final int[] grades;
    public ReportCard(int[] grades) {
        this.grades = grades.clone(); // on entry
    }
    public int[] getGrades() {
        return grades.clone();        // on exit
    }
    // average() unchanged
}
```

With the same `main`

```
80.0
```

- **Defensive copy**: a mutable object is copied when it **enters** and **leaves** the class. The only reference to the inner array stays the class's own field.
- The constructor's copy guards against the caller later changing its array; the getter's copy guards against changes to the returned array.
- For lists, use `new ArrayList<>(list)` or the read-only view `Collections.unmodifiableList(list)`.
- `String`, `Integer` and other immutable objects are not copied: sharing them is harmless.

Copying has a cost; for large data, a read-only view or an immutable type from the start is cheaper.

Designing an Immutable Class

```
public final class Money {         // no subclasses
    private final long cents;      // assigned once
    private final String unit;
    public Money(long cents, String unit) {
        this.cents = cents;
        this.unit = unit;
    }
    public Money plus(Money other) {   // instead of a setter
        if (!unit.equals(other.unit))
            throw new IllegalArgumentException("Units differ");
        return new Money(cents + other.cents, unit);
    }
    @Override public String toString() {
        return cents / 100 + "."
            + String.format("%02d", cents % 100) + " " + unit;
    }
}
```

Rules

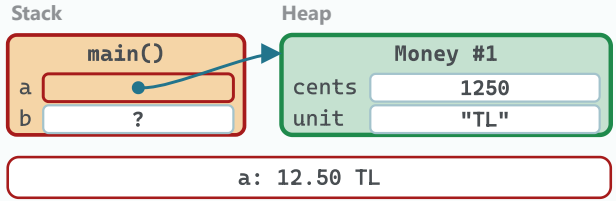
1. No setters; the state is set only in the constructor.
2. Fields are `private final`.
3. The class is `final`: no mutable subclass can be written.
4. "Modifying" methods return a new object.
5. Mutable fields get defensive copies.

Payoff

- The invariant is checked only in the constructor; after that it can never break.
- The object can be shared without copying, can be a `HashMap` key, and is thread-safe.
- `String` and `record` are ready-made examples of these rules.

Immutable `Money`: Code Trace

```
 1 public Money plus(Money other) {
 2     if (!unit.equals(other.unit))
 3         throw new IllegalArgumentException(
 4             "Units differ");
 5     return new Money(cents + other.cents, unit);
 6 }
 7 // in main:
▶Money a = new Money(1250, "TL");
 9 a.plus(new Money(100, "TL"));  // result lost
10 Money b = a.plus(new Money(375, "TL"));
11 System.out.println(a + " | " + b);
```



The first object is built. Its fields are `final`: from now on no method can change them.

Step 1 / 5

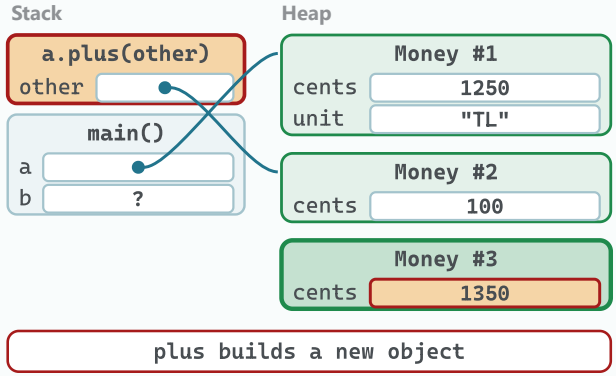
OUTPUT

VARIABLES

`a = 12.50 TL`

Immutable `Money`: Code Trace

```
 1 public Money plus(Money other) {
 2     if (!unit.equals(other.unit))
 3         throw new IllegalArgumentException(
 4             "Units differ");
 ▶     return new Money(cents + other.cents, unit);
 6 }
 7 // in main:
 8 Money a = new Money(1250, "TL");
 ▶ a.plus(new Money(100, "TL"));  // result lost
10 Money b = a.plus(new Money(375, "TL"));
11 System.out.println(a + " | " + b);
```



`plus` leaves `a` unchanged; it builds and returns a **new** object (#3) holding the sum.

Step 2 / 5

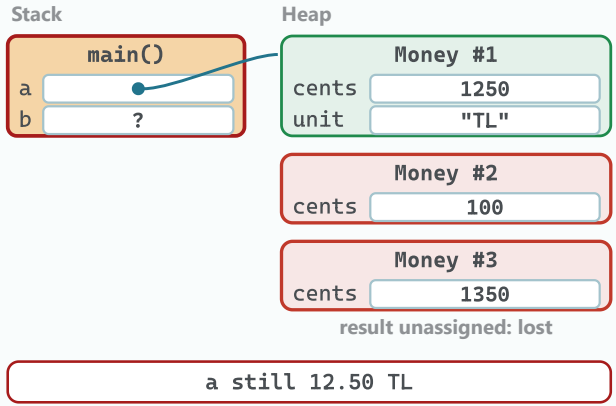
OUTPUT

VARIABLES

`a = 12.50 TL`

Immutable `Money`: Code Trace

```
1  public Money plus(Money other) {
2      if (!unit.equals(other.unit))
3          throw new IllegalArgumentException(
4              "Units differ");
5      return new Money(cents + other.cents, unit);
6  }
7  // in main:
8  Money a = new Money(1250, "TL");
▶  a.plus(new Money(100, "TL"));  // result lost
10 Money b = a.plus(new Money(375, "TL"));
11 System.out.println(a + " | " + b);
```



`a still 12.50 TL`

The returned value was not assigned to a variable: #3 is lost. This is the most common mistake with immutable objects; `a` is unchanged.

Step 3 / 5

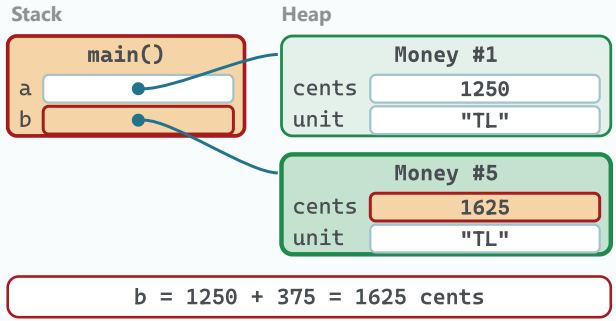
OUTPUT

VARIABLES

`a = 12.50 TL`

Immutable `Money`: Code Trace

```
 1 public Money plus(Money other) {
 2     if (!unit.equals(other.unit))
 3         throw new IllegalArgumentException(
 4             "Units differ");
 ▶     return new Money(cents + other.cents, unit);
 6 }
 7 // in main:
 8 Money a = new Money(1250, "TL");
 9 a.plus(new Money(100, "TL"));  // result lost
 ▶ Money b = a.plus(new Money(375, "TL"));
11 System.out.println(a + " | " + b);
```



This time the result is assigned to `b`: 1250 + 375 = 1625 cents. #4, built as the argument, is no longer used either.

Step 4 / 5

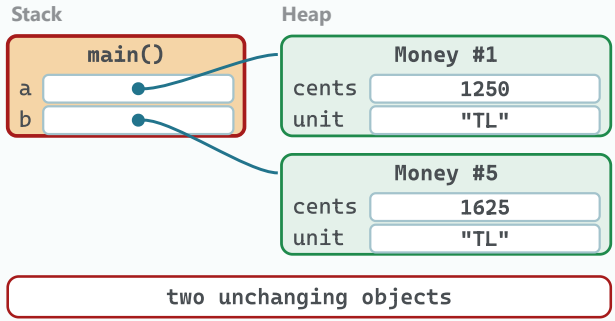
OUTPUT

VARIABLES

`a = 12.50 TL` `b = 16.25 TL`

Immutable `Money`: Code Trace

```
 1 public Money plus(Money other) {
 2     if (!unit.equals(other.unit))
 3         throw new IllegalArgumentException(
 4             "Units differ");
 5     return new Money(cents + other.cents, unit);
 6 }
 7 // in main:
 8 Money a = new Money(1250, "TL");
 9 a.plus(new Money(100, "TL"));  // result lost
10 Money b = a.plus(new Money(375, "TL"));
▶ System.out.println(a + " | " + b);
```



`a` kept its initial value. `String`'s `toUpperCase()` and `concat()` methods work the same way.

Step 5 / 5

OUTPUT

```
12.50 TL | 16.25 TL
```

VARIABLES

`a = 12.50 TL` `b = 16.25 TL`

Packages

- **Package**: a group of classes collected in one folder. The folder's name is the package's name.
- Each `public` class is defined in its own file, named after the class.
- The file's first line declares its package: `package packagename;`
- The compiler and JVM look for packages starting from the **class path** roots; the package name gives the folder path from there. The default class path is the current directory (`.`).
- For nested folders, use a dot instead of `/` or `\` : `school/admin` → `school.admin` .
- Package names are lowercase; Java's own packages: `java.lang` , `java.util` , `java.io` ...

```
project/
├── src/
│   ├── bank/
│   │   ├── Account.java      // package bank;
│   │   └── Helper.java   // package bank;
│   └── app/
│       └── Main.java        // package app;
└── out/                    // compiler output
    ├── bank/
    │   ├── Account.class
    │   └── Helper.class
    └── app/
        └── Main.class
```

A class's full name includes its package: `bank.Account` , `app.Main` .

Package Example

```java
package bank;

public class Account {
    private double balance;

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

```java
package app;

import bank.Account;

public class Main {
    public static void main(String[] args) {
        Account acc = new Account();
        acc.deposit(250.0);
        System.out.println("Balance: " + acc.getBalance());
    }
}
```

COMMAND LINE (INSIDE PROJECT/)

```
> javac -d out src/bank/Account.java src/app/Main.java
> java -cp out app.Main
Balance: 250.0
```

- `-d out` : .class files are written under `out/` with their package folders.
- `-cp out` : the class path; when running, give the class's **full name**.

Importing Classes

- To use a class from another package by its short name: `import packagename.ClassName;` The whole package: `import java.util.*;` (not subpackages).
- Without `import`, you can also use the **full name**: `java.util.Random`.
- `java.lang` (`String`, `Math`, `System`) and classes in the same package are visible automatically.
- Order: first `package`, then `import`s, then the class. `import static` lets you use a static member without the class name.

```
import java.util.Scanner;
import static java.lang.Math.sqrt;

public class ImportDemo {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    java.util.Random rnd =
        new java.util.Random(42);
    System.out.println(sqrt(n) + " "
        + Math.max(n, 10) + " "
        + rnd.nextInt(100));
  }
}
```

OUTPUT (INPUT: 49)

```
7.0 49 30
```

- `Scanner`: short name via import.
- `java.util.Random`: full name, no import.
- `sqrt`: `import static`; `Math.max`: `java.lang`.

Package Access

```
package bank;

class Helper {      // not public
    static double vat(double amount) {
        return amount * 0.20;
    }
}
```

```
package app;

import bank.Helper;

public class Main2 {
}
```

```
Main2.java:3: error: Helper is not public in
bank;
cannot be accessed from outside package
import bank.Helper;
           ^
```

- Classes and members without a modifier have **package access**: they are visible only within their own package.
- `Helper` is the `bank` package's **internal business**: `Account` can use it; the `app` package cannot.
- A package is also a level of encapsulation: its classes are closed to the outside, and only the `public` ones form the package's interface.

Package Naming and Subpackages

- To avoid name clashes, organizations start package names with their **reversed domain name**: `giresun.edu.tr` → `tr.edu.giresun.bank`.
- Dotted names map to folders: `tr/edu/giresun/bank/`.
- **A subpackage is a separate package.** `tr.edu.giresun.bank.report` cannot see the package-private members of `bank`; similar names grant no access rights.
- `import tr.edu.giresun.bank.*;` does not import subpackages: `import java.util.*;` likewise skips the classes in `java.util.function`.

```java
// tr/edu/giresun/bank/Helper.java
package tr.edu.giresun.bank;
class Helper {                    // package-private
    static double vat(double t) { return t * 0.20; }
}
```

```java
// tr/edu/giresun/bank/report/Report.java
package tr.edu.giresun.bank.report;
import tr.edu.giresun.bank.Helper;
public class Report {
    double compute() { return Helper.vat(100); }
}
```

```
error: Helper is not public in tr.edu.giresun.bank;
cannot be accessed from outside package
```

If the package name and folder path don't match exactly, the compiler cannot find the class; that is why IDEs create folders from the package name.

Class Name Clashes

```
import java.util.*;
import java.sql.*;

public class Ambiguous {
    public static void main(String[] args) {
        Date date = null;
    }
}
```

```
Ambiguous.java:6: error: reference to Date
is ambiguous
  both class java.sql.Date in java.sql and
  class java.util.Date in java.util match
```

```
import java.util.Date;

public class DateDemo {
  public static void main(String[] args) {
    Date start = new Date(0);  // java.util
    java.sql.Date sqlDate =          // full name
        java.sql.Date.valueOf("2024-09-16");
    System.out.println(start.getTime());
    System.out.println(sqlDate);
  }
}
```

OUTPUT

```
0
2024-09-16
```

- Classes in two different packages may share a name: `java.util.Date` and `java.sql.Date`.
- To resolve the ambiguity, use the **package name**: bring one in with `import`, and write the other **with its full name**.

Summary

Concept	Meaning	In Java
Encapsulation	Bundle data + methods into one unit, hide internal details	`private` fields, `public` methods
Information hiding	Separate “what” from “how”	Users know only the method signature
Pre-/postcondition	State assumed before / guaranteed after a call	Javadoc comment + `if` check
Access modifier	Who can access	`public`, `protected`, (package), `private`
Getter / setter	Method that reads / validates and changes a field	`getSecond()`, `setSecond(int)`, `this.second = second`
Interface / implementation	The class's outside / inner mechanism	`interface Car`, `class SportsCar implements Car`
Package, import	Group of classes in a folder; short names from other packages	`package bank;`, `import java.util.Scanner;`
Class name clash	Same class name in different packages	Full name: `java.sql.Date`

SUMMARY

Common Mistakes (1)

Accessing a private field from outside

```
error: second has private access in DigitalClock
```

`s.second = 75;` is wrong; write `s.setSecond(75);` instead.

Forgetting this in a setter

```
width = width;   // compiles, field unchanged
```

The parameter shadows the field; `getWidth()` returns 0. Correct: `this.width = width;`

Declaring a getter void

```
error: incompatible types: unexpected return value
```

`public void getWidth()` cannot return a value; the return type must be `int`.

Setter without validation

```
public void setSecond(int s) { second = s; }
```

This is no better than a public field; the setter must enforce the validity rule.

SUMMARY

Common Mistakes (2)

Forgetting import

```
error: cannot find symbol
  symbol:   class Scanner
```

Add `import java.util.Scanner;` at the top of the file.

public class ≠ file name

```
error: class Account is public, should be declared in a file named Account.java
```

The file name must exactly match the public class's name.

Using a package-private class from outside

```
error: Helper is not public in bank; cannot be accessed from outside package
```

Either make the class `public`, or use it only from the same package.

Not implementing an interface method

```
error: SportsCar is not abstract and does not override abstract method stop() in Car
```

`implements`: a class must write every method of its interface.

SUMMARY

Common Mistakes (3)

Exposing an inner object via a getter

`return grades;` exposes a mutable array or list. Return a copy or a read-only view.

Thinking a final field is immutable

`private final int[] grades` : the reference is fixed, not the array elements.

Forgetting to assign the new object

`a.plus(p);` or `s.toUpperCase();` alone changes nothing: `a = a.plus(p);`

Not validating in the constructor

Even if setters check, `new BankAccount(1, -500.0, ...)` builds an invalid object. The constructor must enforce the rule too.

Checking arguments with assert

`-ea` off means the check never runs. `public` method arguments call for `IllegalArgumentException` .

Assuming subpackages are imported

`import java.util.*;` does not cover `Function` : `cannot find symbol` . `java.util.function` must be imported separately.

A setter for every field

The rule drifts into client code. Design behavior first (`withdraw` , `addBonus`); you may not need a setter.

Silently swallowing errors

Ignoring an invalid value without a trace hides the bug; either document it in the contract or throw an exception.

SUMMARY

Worked Problems (1)

```
DigitalClock t = new DigitalClock();
t.setHour(25);
t.setMinute(30);
t.setSecond(-1);
t.setSecond(59);
t.show();
```

```
DigitalClock a = new DigitalClock();
DigitalClock b = a;
a.setMinute(15);
b.setMinute(45);
b = new DigitalClock();
b.setMinute(10);
System.out.println(a.getMinute() + " " + b.getMinute());
```

From the “Accessor Methods” slide: the `DigitalClock` class (hour 0–23, minute/second 0–59).

1. What does the code on the left print?
2. What does the code on the right print? How many `DigitalClock` objects are created?

▶ **Show answers**

SUMMARY

Worked Problems (2)

```
public class Test {
    public static void main(String[] args) {
        DigitalClock s = new DigitalClock();
        s.setSecond(30);                    // A
        int x = s.second;                   // B
        System.out.println(s.getSecond());  // C
        s.getSecond() = 10;                 // D
        s.setSecond("40");                  // E
    }
}
```

Which of lines A–E cause a **compile error**? Why?

▶ **Show answers**

SUMMARY

Worked Problems (3)

```
BankAccount h = new BankAccount(111, 1000.0, "Checking");
h.withdraw(5000.0);
h.deposit(-50.0);
h.addInterest(3.0);
h.withdraw(250.0);
BankAccount v = new BankAccount(222, 100.0, "Savings");
h.transfer(v, 1000.0);
h.transfer(v, 750.0);
System.out.println(h.getBalance() + " " + v.getBalance());
```

`BankAccount`: what does this code print? Which balances appear on the last line?

▶ **Show answer**

SUMMARY

Worked Problems (4)

A) Non-copying `ReportCard`

```
int[] d = {50, 60};
ReportCard k = new ReportCard(d);
int[] g = k.getGrades();
g[0] = 100;
d[1] = 100;
System.out.println(k.average());
```

B) `Money` class

```
Money p = new Money(100, "TL");
p.plus(p);
p = p.plus(p);
p.plus(new Money(1, "TL"));
System.out.println(p);
```

1. What does A print? And with the defensive-copying `ReportCard`?
2. What does B print?
3. A program calls `increment()` ten times on a `Counter`. Run it without `-ea` and with `-ea`: what happens? Which run reveals the bug?
4. `Ratio`: does `new Ratio(1, 2).num` compile in another class?

▶ **Show answers**

SUMMARY

Exercises (1)

1. `Student`: `private` `name` and `grade`; accept grades only in 0–100; `isPassing()` returns `true` if grade ≥ 50.
2. `Point`: accept only non-negative coordinates; `distance()` returns the distance to the origin.
3. `Thermometer`: store a Celsius value (reject anything below −273.15); `getFahrenheit()` computes the value without **a separate field**.

▶ **Answers / hints**

SUMMARY

Exercises (2)

4. `Counter`: give it only `increment()`, `reset()`, `getValue()` methods. Why no setter?
5. `Student`: move it into the `school` package, use it from `app.Main`, and write the compile and run commands.
6. `BankAccount`: add a getter-only `txCount` field that grows by 1 on each successful deposit or withdrawal. What are the two accounts' values at the end of the bank example (trace slide)?

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: "Controlling Access to Members of a Class", "Using the this Keyword", "Packages", "Interfaces".
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition* (6.6 Access Control, 7 Packages and Modules, 7.5 Import Declarations, 15.12.2 method selection).
- P. Deitel and H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2017 (Chapters 3 and 8: Classes and Objects).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022 (Chapter 4: encapsulation).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., 2022 (Chapter 4: Objects and Classes).
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley, 2018 (Items 15–16: minimize accessibility; use accessor methods, not public fields).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 11: Designing Classes).

The End

Chapter 4.2: Encapsulation

Previous: Chapter 4.1: Objects and Classes · Next: Chapter 4.3: Inheritance