# Module 08 — Object-Oriented Programming

**ITP 120 — Java Programming I**

Northern Virginia Community College | Randy Michak

### 🎯 Learning Objectives

- Explain the difference between procedural and object-oriented programming
- Describe what a class is and how objects are created from it
- Write a class with private fields, a constructor, and public methods
- Use the `this` keyword to resolve naming conflicts in constructors
- Create and use objects with the `new` keyword and dot notation
- Write getter and setter methods that enforce encapsulation
- Override `toString()` to give objects a meaningful string representation
- Explain the principle of encapsulation and why it matters
- Read and sketch a basic UML class diagram
- Build and test a complete `BankAccount` class from scratch

### 📚 Key Terms

**object** · **class** · **instance** · **field** · **constructor** · **encapsulation** · **access modifier** · **getter** · **setter** · **dot notation** · **this** · **toString()** · **UML** · **overriding**

## 1. What Is Object-Oriented Programming?

Up through Module 7 you have been writing **procedural** code: a list of instructions that runs top-to-bottom, broken up by methods. That works fine for small programs. Once a project grows,

though, procedural code gets messy fast. Variables get passed everywhere, methods have to know about all the data, and adding a new feature often breaks existing ones.

**Object-Oriented Programming (OOP)** takes a different angle. Instead of writing a sequence of steps, you model your program as a collection of *objects* that each know their own data and what they can do with it.

### 1.1 Procedural vs. Object-Oriented

| Procedural | Object-Oriented |
| --- | --- |
| Program = sequence of steps | Program = collection of objects |
| Data and logic are separate | Data and logic live together in a class |
| Methods receive all data as parameters | Methods operate on the object's own fields |
| Harder to scale, easier to start | More setup upfront, much easier to maintain at scale |

### 1.2 Objects Have State and Behavior

Think about a bank account. It has *data*: the owner's name, the balance. It also has *things it can do*: accept a deposit, process a withdrawal, report the balance. In OOP those two ideas have names:

- **State** — the data an object holds (also called *fields* or *instance variables*)
- **Behavior** — what the object can do (its *methods*)

A `BankAccount` object knows its own balance and knows how to deposit and withdraw. A `Student` object knows its own name and GPA. That pairing — state plus behavior — is the core idea of OOP.

**Why This Matters in Real Projects**

Almost every major Java codebase — Android apps, Spring web services, enterprise software — is built with OOP. Learning to think in objects now sets you up for the rest of your Java career. Every framework you will ever use assumes you understand classes and objects.

## 2. Classes and Objects

The words *class* and *object* get used a lot. Here are the precise definitions:

- A **class** is a blueprint or template. It describes what an object looks like and what it can do. No actual data yet — just the design.
- An **object** is an *instance* of a class. It is the real thing created from that blueprint, with its own actual data values.

> 🧠 **Think of It This Way**
>
> A class is like a cookie cutter. The cutter itself is not a cookie — it is just the shape. Every time you press it into dough you get a new *cookie* (an object). All cookies share the same shape (same fields and methods), but each one is its own separate thing with its own frosting, its own place on the tray. You can make as many cookies as you want from one cutter.

### 2.1 Your First Class

Here is a minimal `Dog` class with two fields and one method:

```
public class Dog {
    String name;   // field: the dog's name
    int age;       // field: the dog's age

    void bark() {
        System.out.println(name + " says: Woof!");
    }
}
```

And here is a separate `Main` class that creates two `Dog` objects from that blueprint:

```
public class Main {
    public static void main(String[] args) {
        Dog d1 = new Dog();   // create first Dog object
        d1.name = "Rex";
        d1.age  = 3;

        Dog d2 = new Dog();   // create second Dog object
        d2.name = "Bella";
        d2.age  = 5;

        d1.bark();
        d2.bark();
    }
}
```

```
Rex says: Woof! Bella says: Woof!
```

Both objects share the same `bark()` method defined in the class, but each has its own `name` and `age`. That is the blueprint-versus-instance relationship in action.

> ⚠️ **Hold On — Those Fields Shouldn't Be Public**
>
> The `Dog` above lets anyone write `d1.age = -99`. That is a real problem. We will fix it in the next section with **access modifiers** and **encapsulation**. For now, just notice the issue.

## 3. Defining a Class

A well-written Java class has three sections in this order:

1. **Fields** — the data the object holds
2. **Constructor(s)** — special methods that set up a new object
3. **Methods** — what the object can do

### 3.1 Access Modifiers: public and private

Java lets you control who can see each part of a class. The two you will use most:

| Modifier | Who can access it? | Typical use |
| --- | --- | --- |
| `public` | Any code anywhere | Methods you want others to call |
| `private` | Only code inside this class | Fields, internal helper methods |

### 3.2 Fields Should Be private

The rule you will follow from now on: **fields are always `private`**. This prevents outside code from directly reading or changing an object's data without going through methods you control. That is the whole point of encapsulation (more in Section 8).

```
public class Person {
    private String name;   // private -- only accessible inside this class
    private int age;

    // constructors and methods go here
}
```

> 💡 **Private Fields Are the Standard**
>
> In every Java textbook, course, and professional codebase you will encounter, fields are private. Start that habit now and it will be second nature by the time you build anything serious.

## 4. Constructors

A **constructor** is a special method that runs automatically when you create a new object with `new`. Its job is to *initialize* the object — give its fields starting values.

Constructor rules:

- Same name as the class (exact capitalization matters)
- No return type — not even `void`
- Called exactly once, at the moment the object is created

### 4.1 Default Constructor

If you write no constructor at all, Java quietly provides a **default constructor** with no parameters. It sets every field to its zero/null default: `0` for numbers, `null` for objects, `false` for booleans. That is what happened in the `Dog` example above.

### 4.2 Custom Constructor with Parameters

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

    // This constructor runs when "new Person(...)" is called
    public Person(String name, int age) {
        this.name = name;   // assign the parameter to the field
        this.age  = age;
    }
}
```

### 4.3 The this Keyword

Notice `this.name = name`. There is a naming conflict: the constructor parameter is called `name`, and the field is also called `name`. Java needs a way to tell them apart.

`this` refers to *the current object*. So `this.name` means “the `name` field that belongs to this object.” The plain `name` on the right side is the constructor parameter.

> 🧠 **Think of It This Way**
>
> Imagine filling out a form that asks “Name: ___”. The blank on the form is `this.name` — the field on the object. The piece of paper someone handed you to copy from is `name` — the parameter. You are copying from the paper into the form.

### 4.4 Overloaded Constructors

You can have more than one constructor as long as they have different parameter lists. This is **constructor overloading** — the same concept as method overloading from Module 7.

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

    // Constructor 1: name and age provided
    public Person(String name, int age) {
        this.name = name;
        this.age  = age;
    }

    // Constructor 2: only name provided, default age to 0
    public Person(String name) {
        this.name = name;
        this.age  = 0;
    }
}

// Usage:
Person p1 = new Person("Alice", 30);   // calls Constructor 1
Person p2 = new Person("Bob");         // calls Constructor 2
```

**⚠️ Watch Out: Writing a Constructor Removes the Default One**

The moment you write any constructor yourself, Java stops providing the free no-argument version. If you still need a no-arg constructor, write it explicitly: `public Person() { }`.

## 5. Creating and Using Objects

You create an object with the `new` keyword. This does three things:

1. Allocates memory for the new object
2. Calls the matching constructor to initialize it
3. Returns a *reference* to the object, which you store in a variable

### 5.1 The new Keyword

```
Person p1 = new Person("Alice", 30);
Person p2 = new Person("Bob",   22);

// p1 and p2 are completely independent objects
// Changing p1 does not affect p2
```

### 5.2 Dot Notation

Once you have an object, you call its methods using a dot:

```
objectVariable.methodName(arguments);
```

Full example with a `greet()` method:

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

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

    public void greet() {
        System.out.println("Hi, I'm " + name + " and I'm " + age + " years
old.");
    }
}

// In Main:
Person p1 = new Person("Alice", 30);
Person p2 = new Person("Bob",   22);
p1.greet();
p2.greet();
```

```
Hi, I'm Alice and I'm 30 years old. Hi, I'm Bob and I'm 22 years old.
```

### 5.3 Multiple Objects from One Class

```
Person alice = new Person("Alice", 30);
Person bob   = new Person("Bob",   22);
Person carol = new Person("Carol", 45);

alice.greet();
bob.greet();
carol.greet();
```

```
Hi, I'm Alice and I'm 30 years old. Hi, I'm Bob and I'm 22 years old. Hi, I'm Carol and I'm 45 years old.
```

Each object has its own independent copy of the fields. They do not share data — only the method definitions are shared across all instances.

> 💡 **Variable vs. Object**
>
> `alice` is a variable that *refers to* a `Person` object. The variable holds the memory address of where the object lives — it is not the object itself. This distinction matters when you start passing objects as method arguments and working with arrays of objects.

## 6. Getters and Setters

Fields are private. So how does outside code read or change them? Through methods you write specifically for that purpose:

- A **getter** (accessor method) returns the value of a field. Convention: name it `getFieldName()`.
- A **setter** (mutator method) changes the value of a field. Convention: name it `setFieldName()`.

### 6.1 Why Not Just Use Public Fields?

If a field is public, anyone can set it to anything — including nonsense values like a negative age or an empty name. A setter gives you a place to validate the incoming value before you accept it.

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

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

    // --- Getters ---
    public String getName() { return name; }
    public int    getAge()  { return age;  }

    // --- Setters with validation ---
    public void setName(String name) {
        if (name != null && !name.isEmpty()) {
            this.name = name;
        }
    }

    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
        // if negative, the old value is kept -- no error thrown yet
    }
}

// In Main:
Person p = new Person("Alice", 30);
System.out.println(p.getName());    // Alice
p.setAge(31);
System.out.println(p.getAge());     // 31
p.setAge(-5);                       // rejected by setter
System.out.println(p.getAge());     // still 31
```

```
Alice 31 31
```

🧠 **Think of It This Way**

Private fields with getters and setters are like a front desk at an office. You cannot walk into the back room and grab files yourself (no direct field access). You go through the front desk (getters/setters). The front desk checks your request, applies rules, and either hands you what you need or says no.

⚠️ **Naming Matters — Especially for Frameworks**

Java frameworks like JavaBeans, Spring, and Hibernate *expect* getters named `getFieldName()` and setters named `setFieldName()`. Deviate and those tools break. Stick with the convention.

## 7. The toString() Method

Every Java class automatically inherits a method called `toString()`. By default it returns something like `Person@3764951d` — the class name plus a memory address. Not useful for debugging.

You can **override** `toString()` to return something meaningful instead.

### 7.1 Overriding toString()

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

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

    public String getName() { return name; }
    public int    getAge()  { return age;  }

    @Override
    public String toString() {
        return "Person[name=" + name + ", age=" + age + "]";
    }
}

// In Main:
Person p = new Person("Alice", 30);
System.out.println(p);       // Java calls toString() automatically
```

```
Person[name=Alice, age=30]
```

The `@Override` annotation tells Java (and other programmers) that you are intentionally replacing the inherited version. The compiler uses it to catch typos in the method signature. Always include it.

### 7.2 How println Uses toString()

Whenever you pass an object to `System.out.println()`, Java automatically calls that object's `toString()` and prints the result. The same thing happens when you concatenate an object into a String:

```
Person p = new Person("Alice", 30);
String msg = "Employee: " + p;   // toString() called automatically here
System.out.println(msg);
```

```
Employee: Person[name=Alice, age=30]
```

**Try It**

Add a `toString()` to your `Dog` class from Section 2. Return something like `Dog[name=Rex, age=3]`. Then print a `Dog` object with `System.out.println()` and confirm it uses your version instead of the cryptic default.

## 8. Encapsulation

**Encapsulation** is the principle of bundling data and behavior together in one class, and restricting direct access to the data from outside. The recipe:

- Fields: `private`
- Methods intended for outside use: `public`

That combination hides the internals and forces all interaction through a controlled interface you design.

### 8.1 Benefits of Encapsulation

| Benefit | What it means in practice |
|---|---|
| Data protection | Outside code cannot set a field to an invalid value |
| Validation | Setters can enforce business rules before storing a value |
| Flexibility | You can change internal implementation without breaking callers |
| Easier debugging | Only one place where a field can change — inside the class |
| Readable API | The public methods tell you exactly what an object can do |

## 8.2 A Broken Example (No Encapsulation)

```
// BAD -- public fields, no protection
public class BankAccount {
    public double balance;   // anyone can set this directly
}

// In Main:
BankAccount acct = new BankAccount();
acct.balance = -50000;   // no one stopped us -- this is a bug waiting to
happen
```

## 8.3 Fixed with Encapsulation

```
// GOOD -- private field, controlled access
public class BankAccount {
    private double balance = 0;

    public double getBalance() {
        return balance;
    }

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

// In Main:
BankAccount acct = new BankAccount();
acct.deposit(500);
System.out.println(acct.getBalance());   // 500.0
// acct.balance = -50000;  -- COMPILER ERROR: balance is private
```

```
500.0
```

**Make Encapsulation the Default**

Get into the habit of making every field `private` and every useful method `public`. If you are ever tempted to make a field public, stop and ask whether a getter/setter would serve the purpose instead. Nine times out of ten it will.

## 9. UML Class Diagrams

**UML** (Unified Modeling Language) is a standard way to draw pictures of classes before or after you code them. A class diagram is the most common type — you will see it in textbooks, job interviews, and design meetings.

A class diagram has three sections stacked vertically:

1. **Class name** — at the top, usually bold
2. **Fields** — in the middle, with their types
3. **Methods** — at the bottom, with return types and parameters

Visibility symbols:

- **+** means `public`
- **–** means `private`

### 9.1 UML for the Person Class

| Person |
|---|
| − name : String<br>− age : int |
| + Person(name : String, age : int)<br>+ getName() : String<br>+ getAge() : int<br>+ setName(name : String) : void<br>+ setAge(age : int) : void<br>+ toString() : String |

Reading left to right in the methods section: visibility — method name — parameters — return type. Simple once you know the notation.

**Sketch Before You Type**

In real projects, developers draw a UML class diagram *before* writing any code. It forces you to think through what fields and methods you need, spots design problems early, and makes it much easier to communicate with teammates. Get in the habit.

### 9.2 Reading a UML Diagram

Given this diagram for a `Car` class:

```
Car
− make : String
− model : String
− year : int
− speed : double

+ Car(make : String, model : String, year : int)
+ getMake() : String
+ getSpeed() : double
+ accelerate(amount : double) : void
+ brake(amount : double) : void
+ toString() : String
```

You can immediately tell: four private fields, a constructor that takes three arguments, six public methods. You could write the Java skeleton for this class without reading another word of documentation.

> 🔧 **Try It**
>
> Draw a UML class diagram for a `Student` class with fields `name`, `id`, and `gpa`, a constructor, getters, a setter for `gpa` (reject values outside 0.0–4.0), and a `toString()`. Then write the Java code from your diagram.

## 10. Complete Example — The BankAccount Class

Let's build a real, working `BankAccount` class from scratch, applying everything from Sections 1 through 9. We will follow this UML diagram:

| BankAccount |
| --- |
| − owner : String<br>− balance : double |
| + BankAccount(owner : String, initialBalance : double)<br>+ getOwner() : String<br>+ getBalance() : double<br>+ deposit(amount : double) : void<br>+ withdraw(amount : double) : boolean<br>+ toString() : String |

## 10.1 The BankAccount Class

```
public class BankAccount {

    // Fields -- always private
    private String owner;
    private double balance;

    // Constructor
    public BankAccount(String owner, double initialBalance) {
        this.owner = owner;
        if (initialBalance >= 0) {
            this.balance = initialBalance;
        } else {
            this.balance = 0;   // reject negative starting balance
        }
    }

    // Getters
    public String getOwner()   { return owner;   }
    public double getBalance() { return balance; }

    // deposit -- add money to the account
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited $" + amount
                               + " | New balance: $" + balance);
        } else {
            System.out.println("Deposit rejected: amount must be positive.");
        }
    }

    // withdraw -- remove money if funds are available
    // returns true if successful, false if not enough funds
    public boolean withdraw(double amount) {
        if (amount <= 0) {
            System.out.println("Withdrawal rejected: amount must be
positive.");
            return false;
        }
        if (amount > balance) {
            System.out.println("Withdrawal rejected: insufficient funds.");
            return false;
```

```java
        }
        balance -= amount;
        System.out.println("Withdrew $" + amount
                           + " | New balance: $" + balance);
        return true;
    }

    // toString -- readable summary of the account
    @Override
    public String toString() {
        return "BankAccount[owner=" + owner + ", balance=$" + balance + "]";
    }
}
```

### 10.2 Using BankAccount from Main

```
public class Main {
    public static void main(String[] args) {

        // Create two accounts
        BankAccount alice = new BankAccount("Alice", 1000.00);
        BankAccount bob   = new BankAccount("Bob",    250.00);

        // Print using toString()
        System.out.println(alice);
        System.out.println(bob);
        System.out.println();

        // Alice deposits and withdraws
        alice.deposit(500.00);
        alice.withdraw(200.00);
        alice.withdraw(2000.00);   // should be rejected
        System.out.println();

        // Bob tries a bad deposit
        bob.deposit(-100.00);      // should be rejected
        bob.deposit(75.00);
        System.out.println();

        // Final state
        System.out.println("Final balances:");
        System.out.println(alice.getOwner() + ": $" + alice.getBalance());
        System.out.println(bob.getOwner()   + ": $" + bob.getBalance());
    }
}
```

```
BankAccount[owner=Alice, balance=$1000.0] BankAccount[owner=Bob, balance=$250.0] Deposited $500.0 | New balance: $1500.0 Withdrew $200.0 | New balance: $1300.0 Withdrawal rejected: insufficient funds. Deposit rejected: amount must be positive. Deposited $75.0 | New balance: $325.0 Final balances: Alice: $1300.0 Bob: $325.0
```

### 10.3 What This Demonstrates

- **Private fields** — `owner` and `balance` cannot be changed directly from `Main`

- **Constructor with validation** — negative starting balances are silently corrected to 0
- **Getter methods** — `getOwner()` and `getBalance()` give read access
- **Validated behavior** — `deposit()` rejects negative amounts; `withdraw()` checks for sufficient funds
- **Return values from methods** — `withdraw()` returns `boolean` so callers know if it succeeded
- **toString()** — `System.out.println(alice)` gives useful output automatically
- **Two independent objects** — Alice's account and Bob's account do not affect each other

> 🔧 **Try It**
>
> Extend the `BankAccount` class with a `transfer(BankAccount target, double amount)` method. It should withdraw from this account and deposit to the target account. Both operations should use the existing validated methods — do not manipulate `balance` directly. Test it from `Main`.

## 📋 Summary

**Module 08 — Key Takeaways**

- **OOP** models programs as collections of objects, each with state (fields) and behavior (methods).
- A **class** is a blueprint; an **object** is an instance created from that blueprint with `new`.
- **Fields should always be `private`**. This is the foundation of encapsulation.
- A **constructor** initializes a new object. It has the same name as the class and no return type.
- The `this` keyword refers to the current object and resolves naming conflicts between fields and parameters.
- **Getters** return field values; **setters** change them with validation. Name them `getFieldName()` / `setFieldName()`.
- Override `toString()` to give objects a readable string representation. `println()` calls it automatically.
- **Encapsulation** = private fields + public methods. It protects data, enables validation, and makes code easier to maintain.
- **UML class diagrams** have three sections: class name, fields (– for private), methods (+ for public).
- A well-designed class like `BankAccount` uses all of these ideas together: private fields, a validating constructor, getters, validated behavior methods, and `toString()`.

## Vocabulary

| Term | Definition |
|---|---|
| **object** | An instance of a class, with its own copy of the class's fields |
| **class** | A blueprint that defines the fields and methods of its objects |
| **instance** | |

| Term | Definition |
|---|---|
| | A single object created from a class; often used interchangeably with “object” |
| **field** | A variable declared inside a class that stores part of an object’s state |
| **instance variable** | Another name for a field; each object gets its own copy |
| **constructor** | A special method (same name as the class, no return type) that initializes a new object |
| **access modifier** | A keyword ( `public` , `private` ) that controls who can access a class member |
| **encapsulation** | Bundling data and behavior together and hiding the data behind a controlled interface |
| **getter** | A public method that returns the value of a private field; named `getFieldName()` |
| **setter** | A public method that validates and sets a private field; named `setFieldName()` |
| **dot notation** | The syntax `object.method()` used to call a method on an object |
| **this** | A reference to the current object; used to distinguish fields from parameters with the same name |
| **toString()** | An inherited method that returns a string representation of an object; override it for useful output |
| **@Override** | An annotation that tells the compiler you are intentionally replacing an inherited method |
| **overriding** | Providing a new implementation of a method that was inherited from a parent class |
| **overloading** | Writing multiple constructors (or methods) with the same name but different parameter lists |

| Term | Definition |
|---|---|
| **UML** | Unified Modeling Language; a standard notation for diagramming classes and their relationships |
| **state** | The data an object holds at a given moment, represented by its field values |
| **behavior** | The actions an object can perform, represented by its methods |
| **new** | The keyword that allocates memory, calls the constructor, and creates a new object |

# 📋 Module 08 Quiz

Choose the best answer for each question. Each question has exactly one correct answer.

**1.** Which term describes the data an object holds — for example, the `balance` in a `BankAccount`?

A. State

B. Constructor

C. Behavior

D. Access modifier

**2.** What does the keyword `new` do in the statement `Person p = new Person("Alice", 30);`?

A. Declares a variable named `p`

B. Allocates memory, calls the constructor, and returns a reference to the new object

C. Copies an existing object

D. Deletes the old object and replaces it

**3.** A developer writes this class. What is printed when `System.out.println(c);` is called?

```
public class Counter {
    private int count = 5;

    @Override
    public String toString() {
        return "Count: " + count;
    }
}

Counter c = new Counter();
```

A. `Counter@` followed by a memory address

B. `count = 5`

C. A compiler error — you cannot print an object directly

D. `Count: 5`

**4.** Which access modifier should be used on a class field to enforce encapsulation?

A. `private`

B. `protected`

C. `public`

D. No modifier (package-private)

**5.** What is the purpose of the `this` keyword in the constructor below?

```
public Product(String name, double price) {
    this.name  = name;
    this.price = price;
}
```

A. It calls the parent class constructor

B. It creates a second copy of the object

C. It distinguishes the object's fields from the constructor parameters that have the same names

D. It marks the fields as static

**6.** A class has a private field `speed` . Which method signature follows the standard Java getter convention for that field?

A. `public double speed()`

B. `public double getSpeed()`

C. `public void returnSpeed()`

D. `private double getSpeed()`

**7.** Look at this code. What is the output?

```
public class Box {
    private int value;

    public Box(int value) { this.value = value; }

    public void setValue(int v) {
        if (v > 0) this.value = v;
    }

    public int getValue() { return value; }
}

Box b = new Box(10);
b.setValue(-5);
b.setValue(20);
System.out.println(b.getValue());
```

A. `-5`

B. `10`

C. `0`

D. `20`

**8.** In a UML class diagram, what does the – (minus) symbol before a member name indicate?

A. The member is `private`

B. The member has been removed from the design

C. The member is `public`

D. The member is a constructor

**9.** You define a custom constructor `public Dog(String name)` in your `Dog` class. What happens if you then try to create a dog with `Dog d = new Dog();` ?

A. Java uses the default no-arg constructor automatically

B. A runtime error occurs

C. A compile-time error occurs because the no-arg constructor no longer exists

D. The custom constructor is called with `null` as the argument

**10.** Which statement best describes the relationship between a class and an object?

A. An object is a blueprint; a class is an instance created from that blueprint

B. A class and an object are the same thing — just different terms

C. A class is the blueprint; an object is a specific instance created from that blueprint

D. An object can only be created once from a given class

## ✅ Answer Key

| Question | Answer | Explanation |
|---|---|---|
| 1 | **A** | State is the data (fields) an object holds; behavior is what it can do (methods). |
| 2 | **B** | A constructor initializes a new object and has the same name as the class. |
| 3 | **D** | `toString()` returns a String representation of the object. |
| 4 | **A** | Encapsulation hides internal data behind `private` fields and `public` getters/setters. |
| 5 | **C** | The `this` keyword refers to the current object instance. |
| 6 | **B** | Getters follow the naming convention `getFieldName()` . |
| 7 | **D** | Setters can include validation logic before assigning a value. |
| 8 | **A** | UML class diagrams use `-` for private and `+` for public members. |
| 9 | **C** | A class can have multiple constructors with different parameter lists (overloading). |
| 10 | **C** | A class is the blueprint; an object is a specific instance created from that blueprint. |

📚 Open Educational Resource (OER) — CC BY 4.0

**ITP 120 — Java Programming I** — Northern Virginia Community College
Author: Randy Michak | Contribute on GitHub