OOP Refresher

1 From Design to Code

Building a set of CRC cards gives you a set of responsibilities — both storage and operational — for each piece of the program. Turning that into Java is mostly a direct translation:

- each **card** becomes a **class**
- each **storage responsibility** (something the class needs to *know*, like "rank and suit") becomes an **instance variable**
- each **operational responsibility** (something the class needs to *support on behalf of others*, like "flip face up") becomes a **method**

However, the CRC card design focuses on the big picture — what classes to create and roughly what each one is responsible for. What it *doesn't* tell you is exactly how specific methods will fit together when called in sequence in an actual program.

For example, when removing cards from a tableau column exposes a face-down card at the top of a column, that card gets turned face-up. But who is actually responsible for that behavior? Is it handled by `TableauCard`'s "pick up cards" method just before returning the cards picked up? Or is it one of the steps in carrying out the move, and thus the move-handling code's responsibility? What `TableauCard`'s method does must be documented so the caller knows what they need to do.

Write the documentation as you write the class and method headers, before writing the bodies of methods or code that calls them.

2 Naming Instance Variables

Guideline: **Instance variable names should end with `_`.** This distinguishes them from parameter and local variable names, helping to prevent initialization bugs in constructors.

3 Access Modifiers

Java gives you four levels of access control:

Modifier	Who can access it
`private`	only code inside this class
(no modifier / *package-private*)	only code in the same package
`protected`	subclasses, as well as code in the same package
`public`	any code, anywhere

A good rule of thumb: **instance variables are `private`, methods stemming from operational responsibilities are `public`.**

This guideline reflects the principles of encapsulation and information hiding. *Encapsulation* is the bundling of a class's data with the methods that operate on it, with outside access restricted to those methods. This matters because responsibility for a value's validity naturally belongs with responsibility for storing it — encapsulation lets the class that owns a piece of data be the same class that enforces what values are valid for it, rather than leaving that enforcement scattered across whatever code happens to touch the field. *Information hiding* is the practice of keeping a class's implementation details inaccessible to the rest of the program. Storage responsibilities only specify what a class is expected to know, not how it keeps track of that information; as a result, hiding instance variables from the rest of the program means a programmer is free to decide (and change) how those storage responsibilities are met without changes rippling outward and breaking code elsewhere.

4 Constructors

Constructors are responsible for putting a newly-created object into a valid initial state — usually this means initializing all of the instance variables.

The header for a constructor is a little different from regular methods: the constructor name must match the class name exactly (including capitalization) and there is no return value.

```
public Card ( Rank rank, Suit suit, boolean faceUp ) {
    rank_ = rank;
    suit_ = suit;
    faceUp_ = faceUp;
}
```

Adhering to the naming scheme of ending instance variable names with _ avoids having to use (and possibly forget to use, resulting in a bug) `this.` to distinguish the instance variables from the parameters with the same name.

4.1 Overloading: More Than One Constructor

A class can have more than one constructor as long as each has a different parameter list. This is called **overloading**. For example, most cards start face down so it might be convenient to have a shortcut for this case:

```
public Card ( Rank rank, Suit suit ) {
    rank_ = rank;
    suit_ = suit;
    faceUp_ = false;
}

public Card ( Rank rank, Suit suit, boolean faceUp ) {
    rank_ = rank;
    suit_ = suit;
    faceUp_ = faceUp;
}
```

Overloading isn't limited to constructors — any method can be overloaded. The key is that since the method names are the same, the parameter lists must be different so that Java can tell the calls apart.

5 `static`

Typically, a Java class has one of two purposes — as a blueprint for creating objects or as a holder of subroutines — and the guideline is: if the class is a blueprint, nothing is `static`; if the class is a holder of subroutines, everything is `static`.

`Card` is being used as a blueprint for creating objects, and object needs its *own* rank, suit, and face-up state. Sometimes, though, you want to create objects but you also need data or behavior that is shared by all of the instances of a class — it belongs to the *class as a whole*, not to any one object. `static` is used for the shared elements.

For example, suppose we want to track how many `Card` objects have been created so far, across the entire program:

```
public class Card {
    private static int cardsCreated_ = 0; // shared by ALL Card objects

    private Rank rank_; // each Card has its own
    private Suit suit_; // each Card has its own
    private boolean faceUp_; // each Card has its own

    public Card ( Rank rank, Suit suit ) {
        rank_ = rank;
        suit_ = suit;
        faceUp_ = false;
        cardsCreated_++;
    }

    public static int getCardsCreated () {
        return cardsCreated;
    }
}
```

There is only ever `cardsCreated_` value, shared by all of the `Card` objects, while every `Card` has its own separate `rank_`, `suit_`, and `faceUp_`.

A `static` method (like `getCardsCreated`) belongs to the class itself and can be called without any particular object: `Card.getCardsCreated()`. Instance methods, by contrast, are called an object: `someCard.getRank()`. `getCardsCreated` can be `static` because it only accesses values that belong to the class rather than individual objects.

Rule of thumb: if a piece of data or behavior is a property of one specific object, it's an instance member. If it's a property of the class as a whole, independent of any particular object, it's static.

6 `toString()` and `equals()`

Every class in Java implicitly extends the built-in `Object` class. `Object` provides a few default methods that every object inherits, including `toString()` and `equals(Object other)`.

The default behavior of `toString()` is to display the object's class and memory address. Often it is desirable to produce something more user-friendly:

```
@Override
public String toString() {
    return rank + " of " + suit;
}
```

`equals()` reflects what the notion of *equivalence* means for the class. The default behavior is "same object" — it returns true only if two references point to the same memory location. This is the desired behavior when object identity is important, such as with players in a game — two `Player` objects might happen to have the same name and score at one point, but they still represent two distinct people. However, with classes like `String` or `Point` (representing a coordinate), two instances with the same value are considered to be equivalent. `Card` arguably falls into this category — even in a game with multiple decks, what usually matters is just the suit and value of the card, not which deck it came from.

We can override `equals` to get this behavior for `Card`:

```
@Override
public boolean equals(Object other) {
    if (!(other instanceof Card)) {
        return false;
    }
    Card otherCard = (Card) other;
    return this.rank == otherCard.rank && this.suit == otherCard.suit;
}
```

The `@Override` annotation isn't strictly required, but it's good practice: it tells the compiler "I intend to replace an inherited method" and the compiler will warn you if the method signature doesn't actually match one being overridden (for example, a typo in the method name).

7 Documenting the Class: Javadoc

Java has a standard documentation-comment format called **Javadoc**. A Javadoc comment starts with `/**` (note the extra asterisk) and typically precedes a class or method declaration. Tools can automatically generate browsable documentation from these comments, and most IDEs will show them to you (and to anyone using your class) as a tooltip.

The body of a Javadoc comment has a strict format: descriptive text is followed by (optional) tags. The first sentence of the descriptive text should be a complete (but brief) summary of the class or method being documented, followed by further detail in subsequent sentences. The first sentence is pulled out for listings of methods or classes in generated documentation so it should be able to stand alone as a brief description.

Key tags:

- `@param` — describes a parameter
- `@return` — describes the return value
- `@throws` — describes an exception the method may throw

- `@author` — identifies the author (typically used for classes rather than methods)

8 Putting It All Together

```
/**
 * Represents a single playing card with a rank, a suit, and a
 * face-up/face-down state.
 */
public class Card {

    private static int cardsCreated_ = 0;

    private Rank rank_;
    private Suit suit_;
    private boolean faceUp_;

    /**
     * Creates a new face-down card of the given rank and suit.
     *
     * @param rank the card's rank (Ace through King)
     * @param suit the card's suit (Hearts, Diamonds, Clubs, or Spades)
     */
    public Card ( Rank rank, Suit suit ) {
        rank_ = rank;
        suit_ = suit;
        faceUp_ = false;
        cardsCreated_++;
    }

    /**
     * Creates a new card of the given rank and suit, with the
     * specified face-up state.
     *
     * @param rank the card's rank
     * @param suit the card's suit
     * @param faceUp whether the card starts face-up
     */
    public Card ( Rank rank, Suit suit, boolean faceUp ) {
        rank_ = rank;
        suit_ = suit;
        faceUp_ = faceUp;
        cardsCreated_++;
    }

    /**
     * Returns this card's rank.
     *
     * @return the rank
     */
    public Rank getRank () {
        return rank_;
    }
```

```
    /**
     * Returns this card's suit.
     *
     * @return the suit
     */
    public Suit getSuit () {
        return suit_;
    }

    /**
     * Returns whether this card is currently face-up.
     *
     * @return true if face-up, false if face-down
     */
    public boolean isFaceUp () {
        return faceUp_;
    }

    /**
     * Flips this card, turning it face-up if it was face-down,
     * or face-down if it was face-up.
     */
    public void flip () {
        faceUp_ = !faceUp_;
    }

    /**
     * Returns the number of Card objects created so far, across
     * the entire program.
     *
     * @return the total number of cards created
     */
    public static int getCardsCreated () {
        return cardsCreated_;
    }

    @Override
    public String toString () {
        return rank_ + " of " + suit_;
    }

    @Override
    public boolean equals ( Object other ) {
        if ( ! ( other instanceof Card ) ) {
            return false;
        }
        Card otherCard = (Card) other;
        return rank_ == otherCard.rank_ && suit_ == otherCard.suit_;
    }
}
```

9 Checklist: Before You Write Your Own Classes

- Are your fields `private`, with controlled access through methods?
- Does every field get properly initialized in every constructor?
- If you have more than one constructor, is each one's parameter list genuinely different (true overloading), and do they avoid duplicating logic unnecessarily?
- Is each piece of state or behavior correctly instance-level (belongs to one object) or static (belongs to the class as a whole)?
- Have you overridden `toString()` (and `equals()`, where object equality matters) to give your class sensible default behavior?
- Does every class, and every public method, have a Javadoc comment explaining its purpose?