

Contents

Chapter 1	4
1.1.....	4
1.2.....	4
1.3.....	4
1.4.....	5
Chapter 2	6
2.1.....	6
2.2.....	6
2.3.....	7
2.4.....	7
2.5.....	10
2.6.....	11
2.7.....	12
2.8.....	14
2.9.....	15
Chapter 3	18
3.2.....	19
3.3.....	21
3.4.....	22
3.5.....	23
3.6.....	25
3.7.....	25
3.8.....	26
3.9.....	27
Chapter 4	28
4.2.....	28
4.3.....	29
4.4.....	30

4.5.....	31
4.6.....	33
4.7.....	34
4.8.....	36
4.9.....	38
4.10.....	40
Chapter 5	43
5.1.....	43
5.2.....	44
5.3.....	46
5.4.....	47
5.5.....	49
5.6.....	50
5.7.....	51
5.8.....	52
5.9.....	54
Chapter 6	56
6.1.....	56
6.2.....	57
6.3.....	59
6.4.....	61
6.5.....	63
6.6.....	64
6.7.....	67
6.8.....	68
6.9.....	71
Chapter 7	73
7.1.....	73
7.2.....	74

7.3.....	75
7.4.....	77
7.5.....	78
7.6.....	79
7.7.....	81
7.8.....	82
7.9.....	84
7.10.....	85
7.11.....	87
Chapter 8	90
8.1.....	90
8.2.....	91
8.3.....	92
8.4.....	95
8.5.....	99
8.6.....	101
8.7.....	103
8.8.....	105
Chapter 9	107
9.1.....	107
9.2.....	108
9.3.....	110
9.4.....	112
9.5.....	114
9.6.....	115

Chapter 1

1.1

Software design is often a vague and misunderstood process, with no clear, tangible "design" to reference. It is defined as the creation of abstractions of data and computation and organizing them into a functioning software system, with design decisions being central to the process. These decisions occur within a "design space," a conceptual space where each choice affects different design quality attributes such as understandability and reusability, leading to trade-offs between various goals. Although software design might seem like a systematic process, it is inherently uncertain and requires iterative problem-solving, guided by experience and principles, making it a creative activity. Ultimately, the design process aims to reduce complexity, with context-specific goals such as maximizing reusability or robustness, and good design should prioritize attributes like understandability and sustainability to avoid messy code and errors.

1.2

Software design is just one aspect of the broader software development process, which involves various models for organizing tasks. Early on, the waterfall model, emphasizing heavy planning, was popular, but in the 1990s, agile development emerged as a more flexible approach. Despite evolving methods, the key is to have a development process tailored to the specific system and organization. A related concept is the idea of software development practices, such as version control and pair programming, which support good design. Iteration and refactoring are central to software design, allowing periodic improvements to the system's design without changing its functionality, addressing issues like technical debt and ensuring ongoing evolution of the design.

1.3

A design is a collection of decisions made during the process of solving a design problem, each decision based on reasoning. These decisions can be stored in a person's mind for small projects, but for larger ones, it is beneficial to capture them externally. This can be done through source code, design documents, email and discussion platforms, or specialized models like those used in model-driven development (MDD). For complex design problems, a specialized modeling language like the Unified Modeling Language (UML) can be used to represent different aspects of a system, such as class relationships or object states. UML is a standardized tool that helps capture design information in a concise way, focusing on key ideas and leaving out irrelevant details, and is used in various

ways depending on the development process, from generating code to sketching design ideas.

1.4

Capturing general design knowledge is challenging due to the heuristic nature of software design, which relies on the skills and experience of the designer. In the 1980s, comprehensive design methods aimed to disseminate design know-how, but these were replaced by object-oriented design methods in the 1990s. The concept of design patterns emerged from the book *Design Patterns: Elements of Reusable Object-Oriented Software*, providing reusable solutions to common design problems, which made it easier to share design knowledge without adopting entire methodologies. A design pattern consists of a name, problem context, solution, and consequences, and is intended as a template rather than a concrete design. Additionally, design antipatterns, or "code smells," represent common flaws in designs that should be avoided, and recognizing them helps guide refactoring and improve code quality.

Chapter 2

2.1

Encapsulation in software design refers to enclosing data and computation to limit interactions between different parts of the code, like how a nut is protected by its shell. This approach offers several benefits, such as making code easier to understand in isolation, reducing errors in interactions, and allowing easier changes without affecting other parts of the system. Encapsulation is closely tied to the principle of information hiding, which suggests revealing only the minimum necessary information and keeping other details hidden, as seen in a stack ADT with its push and pop operations. The term "client code" refers to code that interacts with other components without being part of their definition, which varies depending on the context. While encapsulation and information hiding are broad principles, specific techniques exist to ensure they are properly applied in software design.

2.2

The first design task involves defining abstractions to represent a deck of cards in software. An abstraction is a conceptual building block, and in this case, a deck of cards can be viewed as an ordered collection of playing cards. Several ways to represent a card in code include using integers, arrays, or Boolean values, but each method has significant drawbacks. For example, using integers to represent cards can lead to confusion and errors, as it does not clearly map to the concept of a playing card. The challenge lies in finding a representation that accurately reflects the domain concept, reduces the potential for programming mistakes, and allows for easy maintenance and understanding. The issue of using basic types like integers or arrays to represent domain concepts, such as a playing card, can lead to confusion and errors. A better approach is to avoid using primitive types for domain-specific abstractions, a problem known as the "Primitive Obsession" antipattern. Instead, we can hide the internal representation of a concept behind a custom type or class, such as defining a Card class in Java. This allows us to properly represent a card while hiding the details of its internal structure. Although defining a Card class solves some problems, further steps are needed to ensure the internal representation is fully abstracted and protected from direct manipulation. To improve the encoding of a playing card, we decompose it into two sub-concepts: its rank and its suit. Instead of using primitive types for these, we define dedicated types for rank and suit, as these concepts have a fixed, limited set of values. The most suitable way to represent these values is through enumerated types, which allow us to define a set of named constants for suits (Clubs, Diamonds, Spades, Hearts) and ranks (Ace, 2, 3, etc.). In Java, enumerated types

are a special kind of class, where the constants are objects that can only take valid values, ensuring the code is robust and avoids primitive obsession. By using enumerated types for rank and suit, we ensure the design is clearer, less error-prone, and more aligned with the domain concepts. To represent a deck of cards, we could initially use a simple list of Card objects, such as `List<Card> deck = new ArrayList<>()`. However, this approach has several issues: it doesn't strongly tie the list to the concept of a deck, making it potentially ambiguous; it couples the deck's representation with its implementation (e.g., using an `ArrayList`); and it allows the structure to be corrupted by holding more than 52 cards or duplicates. A better solution is to define a dedicated class, `Deck`, which encapsulates the list of cards and directly ties the deck to its domain concept. This class allows us to hide the implementation details (such as how the cards are stored) and avoids the problems associated with using a basic list for this purpose.

2.3

Encoding abstractions as types is just the first step in achieving encapsulation. Once we have defined the necessary types (e.g., `Deck`, `Card`, `Rank`, and `Suit`), the next task is to ensure these types effectively hide information from client code. This requires controlling access to the variables and methods associated with these types.

In Java and other object-oriented languages, objects group variables together, and their values are accessed through dereferencing. Without proper encapsulation, these variables can be accessed indiscriminately, leading to potential misuse and bugs. For example, in the code where fields such as `aCards` and `aRank` are made public, it's easy to manipulate them in ways that can lead to errors, such as a `NullPointerException`.

To prevent such issues, encapsulation hides the internal workings of an object behind a well-defined interface. One simple but powerful way to do this is by using access modifiers in Java. These modifiers control what parts of the code can access certain fields or methods. By restricting access to internal variables, we ensure that the implementation details are protected and that the abstraction is only used in intended ways. This approach helps maintain the integrity of the object and reduces the likelihood of errors, making code more reliable and maintainable.

2.4

In Java, controlling the visibility of classes and their members is essential for encapsulation, and this is accomplished using access modifiers. The primary access modifiers are `public` and `private`. When a field or method is marked as `public`, it is accessible from anywhere in the code, which can lead to unintended use or modification. For instance, in the case of the `aCards` field in the `Deck` class, if it were `public`, it could be

accessed and modified directly from any code that has a reference to the Deck object. In contrast, when a field or method is marked as private, it is only accessible within the class itself, ensuring that the internal state is protected.

A good design principle is to always use the narrowest possible scope for class members. This means that instance variables should almost always be marked as private to limit their access, while public methods should reveal as little as possible about the internal workings of the class. For example, in the revised Card class, the aRank and aSuit fields are private, and the only access to them is through public getter methods. This ensures that client code cannot directly interact with the internal representation of a card, preserving the integrity of the design.

With this approach, the internal implementation of a class can change without affecting client code. For instance, the representation of a card could be changed to use a single int or an enumerated type, and client code would not need to be modified. This flexibility is one of the core benefits of good encapsulation.

Access modifiers also serve a dual purpose in software design. They express the developer's intent about where certain structures should be used and automatically enforce this intent during compilation. The public methods of a class form the interface to that class, representing what the client code can interact with. The design and implementation of all other fields and methods remain hidden, ensuring that client code only has access to the functionality intended for use. This structure promotes cleaner, more maintainable code.

An object diagram is a type of UML diagram that visually represents objects and how they are interrelated through references. It is especially useful when dealing with complex structures or when relationships between objects are key to understanding the system. When a statement is executed in a program, an object of a class is created, and a reference to this object is returned. This reference can be passed around the program, creating a network of objects that may interact with one another.

In an object diagram, each object is represented by a rectangle. The rectangle typically contains the name and type of the object, which are written as name:type. The inclusion of the name and type is optional, but it is generally helpful to have at least one of them to provide clarity. In UML object diagrams, the name of objects (as opposed to classes) is underlined.

Objects in the diagram may contain fields, similar to the instance variables in a Java program. These fields can store values of either primitive types or reference types. If a field holds a reference type, this is represented by a directed arrow pointing to the object that is

being referenced. This allows the diagram to visually show the relationships between different objects and how they refer to one another, which is particularly helpful for understanding the organization and interdependencies within a program.

By using object diagrams, developers and designers can better visualize the relationships and structures in a system, aiding in both understanding and debugging complex object-oriented designs.

The second example diagram (Figure 2.2) demonstrates some modeling simplifications that are sometimes useful in UML object diagrams. These simplifications help make the diagram more understandable and focused on the core relationships between objects, without delving into unnecessary details.

Key Simplifications:

1. Untyped "main" Object:

- An untyped object called main is introduced to represent the method body of the main function. Since object diagrams don't have a specific notation for method bodies, this untyped object serves as a trick to represent local variables within a method, which are similar to instance variables in objects.

2. Anonymous Deck Object:

- The Deck object is anonymized in the diagram. Instead of representing a specific instance of a Deck, the variable deck holds a reference to some object. This abstracts away the actual object and focuses on the variable holding the reference to it.

3. String Representation:

- The main method contains a name variable, which stores a string. In Java, a string is technically an instance of the String class. However, to avoid unnecessary complexity, the diagram simply shows the string literal, rather than representing it as a reference to a String object with an internal array of characters. This keeps the diagram cleaner without losing essential meaning.

4. Abstracting Internal Data Structures:

- The internal data structure used to store the cards (like an ArrayList) is abstracted away. The diagram shows that the Deck object contains cards, but it does not reveal how the cards are stored (whether in an array, list, etc.). In many cases, such internal details are irrelevant to the diagram's purpose and are omitted to maintain focus on higher-level concepts.

5. Evocative Names for Card Instances:

- The Card instances are represented by evocative names, rather than showing the values of their fields (aRank and aSuit). This simplification does not imply that the Card objects don't have these fields; rather, it means that this specific detail is omitted in the diagram for clarity.

These simplifications make the diagram more abstract and easier to read, without getting bogged down in low-level details that may not be crucial for the understanding of the system's structure at this stage of design. They focus on relationships and essential elements, keeping the diagram useful and clear.

2.5

Main Points:

1. Encapsulation via the private Keyword:

- The private keyword provides basic encapsulation by restricting direct access to fields within a class. However, this is not foolproof, and encapsulation can still be compromised if internal structures are improperly accessed.

2. Problem with Getter Methods:

- Returning a reference to an internal structure (e.g., through a getter method) can break encapsulation. For example, returning a reference to a List<Card> in the Deck class allows external code to modify the deck's internal list, which may lead to errors (e.g., adding invalid cards).

3. Inappropriate Use of Getters and Setters:

- Automatically providing getters and setters for every field weakens the encapsulation of an object. This is referred to as the "Inappropriate Intimacy" antipattern, where objects expose too many internal details, which should ideally be hidden.

4. Storing External References Internally:

- Allowing external references to modify the internal state of an object can compromise encapsulation. For instance, using a setter method or passing external references through the constructor to initialize internal fields can lead to unintended changes to the object's state.

5. Leaking References Through Shared Structures:

- References can also escape a class through shared structures, such as lists or collections. For example, adding an internal list to an external list (via a method like `collect`) can allow external code to modify the internal structure of the object, violating encapsulation.

6. **Manual Detection and Prevention:**

- Detecting when references escape the class scope is a complex problem and currently cannot be automatically detected by most tools. Preventing this requires careful programming and code inspection to ensure references are not inadvertently leaked.

7. **Best Practices:**

- To maintain good encapsulation, internal structures should not be exposed to client code directly. Instead, objects should interact with each other through well-defined methods that encapsulate the internal details, ensuring that the state of the object is protected from external modifications.

2.6

Main Points:

1. **Encapsulation and Modifying Internal State:**

- Good encapsulation ensures that the internal state of an object can only be modified through its methods, preventing direct access or modification from external code.

2. **Escaping References:**

- Section 2.5 discussed how leaking references can threaten encapsulation. However, if the object being referenced is **immutable**, leaking a reference is harmless since the object's state cannot be changed.

3. **Immutability:**

- Immutability refers to objects whose internal state cannot be modified after creation. Immutable objects can be safely shared because their data cannot be altered.
- In Java, **String** is an example of an immutable object, as its value cannot be changed once created.

4. **Ensuring Immutability:**

- To create immutable objects, a class must be carefully designed to prevent any modification of its internal state (e.g., by not providing setter methods or allowing references to escape).
- A class that produces immutable objects is called an **immutable class**. In Java, there is no built-in mechanism to enforce immutability, so it must be ensured through careful design.

5. **Advantages of Immutability:**

- Immutability supports safe sharing of objects without violating encapsulation. This makes it a desirable property in many design situations.

6. **Immutable Card Class Example:**

- The **Card** class is made immutable by making its fields private and only setting their values via the constructor (which can only be executed once per object).
- The class has no setter methods, and its fields are of immutable types (Rank and Suit), ensuring that the object's state cannot be changed after initialization.

In summary, immutability is a key design property that helps maintain encapsulation by preventing the internal state of objects from being modified externally. It provides a safer way to share data without breaking encapsulation.

2.7

Main Points:

1. **Exposing Information Without Breaking Encapsulation:**

- There are several ways to expose part of an object's internal state (like the cards in a Deck class) while maintaining encapsulation, which prevents external modification of the object's internal state.

2. **Extended Interface:**

- One approach is to extend the interface of the class to include methods that only return references to immutable objects. For example, methods like `size()` and `getCard()` can be used to expose the number of cards or access individual cards, but if Card is immutable, this maintains encapsulation. However, this approach can lead to cumbersome client code, especially if clients need to access all cards in the deck.

3. Returning Copies:

- Another approach is to return a **copy** of the internal data (e.g., a copy of the list of cards). By using `new ArrayList<>(aCards)`, you return a new list containing the same cards, thus protecting the internal state of the deck.
- If the Card objects are immutable, this shallow copy is sufficient to maintain encapsulation. However, if Card objects are mutable, this shallow copy could still allow external modification of the card objects, which violates encapsulation.

4. Deep Copying:

- To ensure proper encapsulation with **mutable** Card objects, a **deep copy** must be performed. This means copying not only the list but also each individual Card object within the list. This can be done by using a copy constructor for Card to create new instances of Card when copying the list.

5. Copy Constructors:

- A **copy constructor** can be used to copy the fields of an object when creating a new instance. For example, copying a Card object would involve creating a new Card with the same rank and suit. This ensures that even with mutable objects, encapsulation is maintained.

6. Other Strategies:

- In addition to copying, Java provides alternatives like **unmodifiable view collections**. For instance, using `Collections.unmodifiableList(aCards)` returns an unmodifiable wrapper around the list, allowing the external code to access the cards without modifying the underlying list.

Conclusion:

There are several techniques for exposing internal data while maintaining encapsulation, including extending interfaces with access methods for immutable objects, returning copies of data, and using unmodifiable view collections. For mutable objects, deep copying is necessary to preserve encapsulation and prevent external modifications.

2.8

Main Points:

1. Benefit of Encapsulation:

- Encapsulation prevents client code from corrupting the internal state of an object. However, issues can arise, such as the creation of invalid instances, as seen in the Card class where a null rank or suit could be passed.

2. Input Validation:

- To prevent creating invalid instances (e.g., a card with null values), input validation can be added to constructors or methods. In the Card class, a null check is added to the constructor, and if either Rank or Suit is null, an exception (IllegalArgumentException) is thrown.
- This ensures that invalid Card objects cannot be created, as exceptions prevent the constructor from completing.

3. Exception Handling:

- Exception handling is crucial for ensuring proper behavior when invalid input is detected. For the Card constructor, the exception is documented using Javadoc's @throws tag.
- It's important to design the exception handling clearly so users know the conditions under which exceptions will be raised.

4. Handling Invalid Operations in Other Methods:

- In the Deck class, the draw() method may throw an exception (IndexOutOfBoundsException) when attempting to draw from an empty deck. This happens because the remove method is misused without checking if the deck is empty.
- A better solution is to check if the deck is empty beforehand and throw a more specific exception, such as IllegalStateException, which aligns with good design principles.

5. Design Principles for Exception Handling:

- Exceptions should be used for unpredictable situations, not for predictable ones like an empty deck. Therefore, the Deck class should validate the

deck's state (`isEmpty()`) before attempting to draw a card, throwing an `IllegalStateException` if the deck is empty.

6. Input Validation Burden:

- Input validation ensures object validity but introduces the responsibility of error handling to the client code. The client must be prepared to catch exceptions raised by invalid operations.
- The additional validation code adds complexity to the class and requires maintenance. It may be excessive in some cases, such as when the code guarantees valid inputs.

7. Trade-offs:

- While input validation makes classes more robust, it shifts the responsibility of handling errors to the client. Developers must balance the benefits of robustness with the additional code complexity and documentation required for error handling.

Conclusion:

Input validation is a powerful tool to ensure object integrity and prevent invalid operations, but it adds complexity in terms of exception handling and maintenance. Careful design and clear documentation of error cases are essential for maintaining encapsulation while avoiding unnecessary overhead.

2.9

Main Points:

1. Input Validation vs. Client Responsibility:

- Input validation ensures the correct usage of an object, but sometimes the client code can be written to preclude errors, making validation unnecessary. For example, if all cards in a deck are created using predefined values, no invalid inputs (like null) will be passed to the Card constructor.
- However, this reliance on client code can lead to ambiguity about whether a method should handle invalid inputs or assume the client has done so.

2. Ambiguity in Method Behavior:

- Without clear documentation, a method's behavior can be ambiguous. For example, it's unclear whether the Card constructor should validate inputs or if it relies on the client to pass valid arguments.
- Different interpretations of constructor behavior can lead to confusion, such as whether invalid inputs result in exceptions, default values, or undefined behavior.

3. Design by Contract:

- To eliminate ambiguity, **design by contract** provides a formal framework for defining method interfaces. It involves specifying **preconditions** (conditions that must be true before a method is called) and **postconditions** (conditions that must be true after a method executes).
- The method contract ensures the client meets the preconditions, and the method guarantees the postconditions. If a precondition is violated, the client is responsible; if a postcondition fails, the method is at fault.

4. Using Javadoc for Preconditions and Postconditions:

- Preconditions and postconditions can be specified using Javadoc tags like @pre and @post, helping clarify the expected behavior of methods.
- Java's assert statement can be used to enforce these conditions during development, raising an AssertionError if a condition is violated. This helps in debugging by clearly identifying where the problem lies.

5. Avoiding Defensive Programming:

- Design by contract reduces the need for defensive programming (e.g., checking for null values everywhere), as the method's contract explicitly defines valid inputs.
- The technique also helps with **blame assignment**: if an assertion fails, it's clear whether the caller or the method itself is responsible for the error.

6. Preconditions vs. Input Validation:

- A key distinction is that preconditions specify what is valid input, while input validation (such as checking for nulls) is part of the method's behavior. These are two different design decisions.
- It's important to decide whether the method itself handles invalid inputs (like throwing an exception) or simply specifies them as invalid in its contract.

7. Role of Assertions:

- Assertions are not meant for general input validation but for detecting design flaws. They should not be used to handle invalid user inputs during runtime, but rather to verify that the program's logic adheres to the expected contract.

Conclusion:

Design by contract enhances code clarity and robustness by formally specifying preconditions and postconditions. This approach minimizes ambiguity in method behavior, improves debugging, and avoids unnecessary input validation in the code. Assertions help enforce these contracts during development, but they should not replace regular input validation mechanisms.

Chapter 3

3.1

Main Points:

1. Interface Definition:

- An interface to a class consists of the **public methods** of that class that are accessible to other classes.
- The interface defines the methods through which other classes can interact with an object of the class.

2. Coupling of Interface and Implementation:

- In a basic setup, like the Deck class, the interface is tightly coupled to the class definition (i.e., the available methods are specific to the class and its implementation).

3. Decoupling Interface from Implementation:

- There are situations where it's beneficial to **decouple** the interface from its implementation, allowing flexibility. For example, in card games, you may need to draw cards from various sources, not just from a standard deck.

4. Using Interfaces for Flexibility:

- By defining an interface like CardSource that specifies methods (draw() and isEmpty()), we can design methods that work with any object that implements this interface, rather than just one specific class like Deck.

5. Java Interface Types:

- In Java, **interfaces** define abstract methods and provide a specification without implementation details.
- A class can implement an interface using the implements keyword, ensuring that the class provides concrete implementations of the interface methods.

6. Polymorphism:

- Polymorphism allows different classes to implement the same interface, and objects of these classes can be treated as instances of the interface type.
- This enables more flexible code, as methods can operate on any class that implements the interface, regardless of the concrete class.

7. Example of Polymorphism:

- In the drawCards method, by using CardSource as a parameter type, the method can now work with any object that implements CardSource, not just a specific class like Deck.

8. Loose Coupling and Extensibility:

- **Loose coupling:** Code using an interface is not dependent on specific implementations, making the system more flexible.
- **Extensibility:** New implementations of an interface can be added without affecting the existing codebase, as long as the new class conforms to the interface.

9. Java Collections Framework:

- The concept of interfaces and polymorphism is exemplified in the **Java Collections Framework** (e.g., List interface and its implementations like ArrayList and LinkedList).
- Both ArrayList and LinkedList provide the same methods (specified by the List interface), so they can be swapped easily in code.

In summary, defining interfaces and using polymorphism allows for more reusable, flexible, and extensible code, as it decouples the interface from specific implementations and facilitates easier swapping of components without changing the overall code logic.

3.2

Main Points:

1. Design Questions about Interfaces:

- Determining whether a separate interface is needed depends on the specific design problem. Interfaces help solve problems or realize features but are not always necessary.

2. Code Reuse with Library Methods:

- Reusing code is easy when library methods like Collections.shuffle() work generically with any collection, without needing to know the specific objects in the collection.

3. Sorting and the Need for Interfaces:

- Sorting objects with `Collections.sort()` requires a way to compare them. The standard Java sorting method needs to know how to compare custom objects like `Card`, but different sorting orders might be needed (e.g., by rank or suit).

4. **Comparable Interface:**

- The **Comparable** interface defines the `compareTo(T)` method, allowing objects to be compared and sorted. It returns:
 - 0 for equality,
 - a negative value if the object should come before the other,
 - a positive value if it should come after.
- By implementing this interface, classes can be sorted in a standardized way.

5. **Polymorphism and Loose Coupling:**

- The `Comparable` interface allows code like `Collections.sort()` to work with any object that implements it, without knowing the specifics of the object. This is an example of **loose coupling**, where the sorting code only requires minimal functionality from the objects.

6. **Minimal and Cohesive Interface Design:**

- Interfaces should capture the smallest cohesive slice of behavior that client code will need, as demonstrated by `Comparable`. Many Java interfaces are named with an "able" suffix (e.g., `Iterable`, `Serializable`, `Cloneable`), indicating that the objects are "fit to be" used for specific behaviors.

7. **Card Class and Comparable Implementation:**

- The `Card` class can implement `Comparable<Card>` to define how cards should be compared. This allows cards to be sorted. An initial implementation compares ranks but leaves the suit order undefined, which could lead to unpredictability. A better implementation ensures a total ordering.

8. **Natural Ordering and Java Enumerations:**

- Java's `Comparable` interface is also implemented by types like `String` (lexicographically) and enumerated types (using ordinal values). The `Card`

class's compareTo method can be simplified by using the natural ordering of the rank if it is an enumerated type.

9. Separation of Concerns:

- Using small, well-defined interfaces like Comparable encourages **separation of concerns**. The compareTo method in Card is focused solely on comparison, making the code easier to understand and maintain. This principle avoids tangled or scattered concerns in the design.

In summary, the **Comparable interface** in Java provides a way to define a natural ordering for objects, facilitating sorting and promoting code reuse. Proper use of interfaces supports **loose coupling, separation of concerns, and minimalistic, cohesive designs**.

3.3

Main Points:

1. Class Diagrams:

- Class diagrams represent the **static (compile-time)** view of a software system, focusing on **types** and their **relationships**.
- They are useful for showing how types are defined and related but are not ideal for capturing **run-time properties** of the code.

2. Purpose of UML Diagrams:

- UML diagrams are **models** that capture the essence of design decisions, not an exact translation of code.
- They are used to simplify and visualize key design concepts without including every detail.

3. Key Concepts in UML Class Diagrams:

- **Aggregation, Association, Dependency:** These represent different types of relationships between classes.
- **Navigability:** Shows how objects of one type can access objects of another, which can be **unidirectional, bidirectional, or unspecified**.

4. Class Attributes and Methods:

- **Attributes:** The Card class doesn't list attributes for aRank and Suit since they are represented as **aggregations** to Rank and Suit types.

- **Methods:** The Card class's methods (constructors and accessors) are omitted to avoid clutter. It's not wrong to include them, but they don't add significant insight.

5. UML Limitations:

- UML doesn't have a straightforward way to indicate the **absence** of members (fields or methods). Notes are used to clarify such details.
- Representing **generic types** can be tricky, as it might make more sense to show the **type parameter** (e.g., Comparable<T>) or the **type instance** (e.g., Comparable<Card>).

6. Static vs. Non-static Members:

- UML tools often lack a way to distinguish between static and non-static members. In some tools (like JetUML), the keyword static is prefixed to static methods to indicate their status.

7. Cardinality in UML:

- **Cardinality** indicates the number of instances involved in a relationship, e.g., a deck might aggregate **0 to 52** instances of Card.
- Cardinalities include specific numbers (e.g., 1), wildcards (e.g., * for zero or more), and ranges (e.g., M..N).

These points summarize how UML class diagrams help represent design concepts, their limitations, and how to convey key relationships and structure in software design.

3.4

Here's a summary of the main points:

1. **Interface Subsets:** An interface type often defines only a subset of the operations for the classes implementing it. For example, the Comparable interface defines comparison behavior, but Card may have additional methods like getSuit() and getRank().
2. **Challenges with Switching Comparison Strategies:** Using global variables to switch comparison strategies within the compareTo method is a bad design (an anti-pattern). It degrades code clarity and violates separation of concerns.

3. **Comparator Interface:** The solution is to use the `Comparator<T>` interface, which has a `compare(T pObject1, T pObject2)` method for comparing two objects. It allows flexible sorting strategies without modifying the original class.
4. **Comparison Using Comparator:** The Comparator interface allows defining multiple sorting strategies, like a "RankFirstComparator" or "SuitFirstComparator". These comparators can be passed to methods like `Collections.sort()`.
5. **Function Objects:** Comparators are often referred to as function objects because they implement a single method (`compare`), making them a straightforward and reusable piece of logic.
6. **Accessing Private Members:** If a comparator needs access to private members of the class it compares, it can be defined as a nested class within the class being compared, or as an anonymous class, or using lambda expressions for conciseness.
7. **Lambda Expressions:** Lambda expressions can be used to define comparators in a more concise way, avoiding the need for named classes and methods, making the code cleaner.
8. **Factory Methods:** A static factory method can be used to create comparators. This approach helps encapsulate comparator logic and ensures good design.
9. **Storing Data in Comparators:** There is a question of whether comparators should store data. For example, a `UniversalComparator` could store an enumerated value that specifies the sorting criterion (e.g., rank or suit), but this could complicate the design.

In summary, using the Comparator interface improves flexibility and maintainability when sorting objects by different criteria, and it can be implemented in various ways, including using lambda expressions or factory methods for better design and encapsulation.

3.5

Here's a summary of the main points:

1. **Accessing Encapsulated Objects:** When designing data structures, it's important to provide access to internal objects (e.g., cards in a deck) without violating information hiding principles.
2. **Initial Solution:** One approach is to return copies of the internal data (e.g., a copy of the list of cards), but this can subtly expose implementation details like how data is stored.

3. **Iterator Concept:** A better solution is to use an iterator, which allows client code to access and iterate over the internal objects without exposing the internal structure. Java provides the Iterator interface with methods hasNext() and next().
4. **Iterator in Deck Class:** By implementing the Iterator interface in the Deck class, we can return an iterator to iterate over the cards in a deck, hiding the internal structure of the deck.
5. **Iterable Interface:** To generalize iteration across different object types, the Iterable<T> interface can be used. This interface provides a single method Iterator<T> iterator(), which allows any object to be iterated over if it implements this interface.
6. **Implementing Iterable in Deck:** The Deck class can implement the Iterable<Card> interface and provide an iterator() method, allowing it to be used in any context that expects an Iterable<Card>, like the enhanced for loop.
7. **Enhanced For Loop:** The enhanced for loop (or foreach loop) works with any Iterable object. For example, iterating over a deck of cards can be done using for (Card card : deck).
8. **Iterator Implementation:** Instead of manually creating an iterator, the Deck class can delegate the iteration to an existing iterator from a List<Card>. This keeps the design simple and reuses the list's iterator.
9. **Encapsulation and Iterator:** Using List<Card>'s iterator can break encapsulation if the iterator's remove() method is used. To avoid this, one option is to return an iterator from an unmodifiable list.
10. **UML Representation:** Representing the Iterator interface in UML is tricky since the iterator's concrete class is often unknown or anonymous. The UML stereotype <<anonymous>> can be used to indicate this.

In summary, the iterator design pattern provides a clean way to access encapsulated objects in a data structure without exposing internal details, and using interfaces like Iterable and Iterator in Java enables flexible and decoupled iteration across different object types.

3.6

- **Iterators** are tools that let you access objects inside another object without showing how they're stored.
- This is part of a design idea called the **ITERATOR pattern**.
- The **ITERATOR pattern** lets us go through a collection of objects one by one without revealing how they're kept inside the container.
- In Java, **Iterable** and **Iterator** are the main tools used to follow the ITERATOR pattern:
 - **Iterable** is for the object that holds the collection (like a list).
 - **Iterator** is for the tool that lets us go through the objects in the collection.
- You can use **Iterable** for the object that holds the items, and **Iterator** for the tool that helps us go through them, making it easy to use Java's **for-each loop**.
- To create an **Iterator**, the easiest way is to use the one already provided by Java tools (like **ArrayList.iterator()**).
- If you need to mix items from different collections or change the order, it's simplest to create a new collection with the items in the correct order and then use that collection's iterator.

3.7

Here's a summary of the main points:

- **Interfaces and polymorphism** promote flexible designs, allowing parts of the code to be easily swapped or changed.
- A key example of this flexibility is the use of **Comparator** in the **Collections.sort(...)** method, which is part of the **STRATEGY design pattern**.
- The **STRATEGY pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows algorithms to vary independently from their clients.
- In object-oriented code, algorithms in the same family implement the same interface, making them easy to switch between.
- **STRATEGY pattern** is useful for cases where different algorithms or behaviors need to be selected or switched dynamically.

- Common design questions when applying STRATEGY:
 - Does the strategy need one or multiple methods?
 - Should the strategy method return a value or modify its argument?
 - Does the strategy need to store data?
 - What types should the method parameters and return values have to minimize coupling?
- The **Comparator interface** is a simple example of STRATEGY, allowing different card comparison strategies.
- Using a **UniversalComparator** that changes the comparison strategy based on internal state is not a true STRATEGY, as it changes the strategy by modifying state, not by switching the strategy object itself.

3.8

Here's a summary of the main points:

- **Interfaces** and **polymorphism** help decouple classes from specific implementations, making code more flexible.
- In the **Deck of cards** example, different sorting strategies can be applied using the **STRATEGY pattern**, where the sorting behavior is defined by a **Comparator**.
- A simple approach to setting the **Comparator** in the **Deck** class would be to initialize it directly in the class, but this makes it hard to change the sorting strategy and tightly couples the **Deck** class to a specific implementation of the comparator.
- Another option is to use an **anonymous class** for the comparator, but it still has the same issues of tight coupling and lack of flexibility.
- A better solution is **dependency injection**, where the comparator is passed into the **Deck** class from outside, decoupling the class from any specific comparator.
 - This allows the comparator to be switched easily without changing the **Deck** class code.
 - Example: `Deck deck = new Deck(new ByRankComparator());`
- **Dependency injection** is a common technique that can be applied in various ways, such as through constructors, factory methods, or setter methods.

- **Constructor-based injection** (as shown in the example) is preferred over setter methods to avoid state management issues.

3.9

Here are the main points of the text:

- **Interface Segregation Principle (ISP):** This principle suggests that client code should not depend on interfaces it doesn't need. Interfaces should be specialized and focused on a small, coherent slice of behavior.
- **Example of ISP:** In the case of a Deck of cards, a CardSource interface with just draw() and isEmpty() methods allows for flexibility. If we define a broader interface (like IDeck with methods like shuffle()), client code might be forced to depend on methods it doesn't need.
- **Splitting behavior:** The idea is to separate behaviors into multiple interfaces (e.g., Shufflable for shuffle behavior), allowing for maximum flexibility. This reduces tight coupling between different features.
- **Problem with combining behaviors:** When a client needs multiple behaviors (e.g., Iterable<Card> and CardSource), Java's type system only allows one interface per variable, leading to awkward and potentially unsafe solutions like casting.
- **Using subtyping:** The solution to combining behaviors in a clean way is to use interface inheritance (subtyping). For example, CardSource can extend Iterable<Card>, allowing a single type to provide both behaviors.
- **Real-world limitations:** In practice, Java's Iterable<T> is a library type and cannot be modified. This makes combining interfaces like CardSource and Iterable<Card> difficult, though it's still theoretically possible.
- **Flexibility in design:** It's important to balance between strict adherence to ISP and practical usage. If two interfaces are often used together in the client code, it might make sense to combine them into one interface.

Chapter 4

4.1

Here are the main points in enumerated form:

1. **Static Perspective:** Refers to the software system in terms of source code and relationships between elements (e.g., a Deck class with a aCards field). This is a compile-time view and is best represented by source code or class diagrams.
2. **Dynamic Perspective:** Refers to the state of objects during runtime (e.g., the number of cards in a Deck instance at different points). This view represents values and references held by variables during program execution and is typically observed in debuggers. It cannot easily be captured in one diagram.
3. **Complementary Views:** The static and dynamic perspectives are complementary. Sometimes it's better to focus on one perspective, but often both are needed for a full understanding of the system.
4. **Wave-Particle Duality Analogy:** Similar to the duality in physics, where light can be viewed as both a particle and a wave, software design requires sometimes using the static view and sometimes the dynamic view. Neither fully explains software on its own, but together, they provide a complete picture.
5. **Chapter Focus:** This chapter emphasizes understanding important dynamic properties of software.

4.2

Here is the summary in enumerated points:

1. **Object State:** Refers to the information an object represents at any given moment. It is useful to distinguish between concrete state and abstract state.
2. **Concrete State:** The actual values stored in an object's fields. For example, a Player object may store a score, and its concrete state represents all possible values for the score.
3. **State Space:** The set of all possible concrete states for an object. For objects with reference types, the state space can grow exponentially. For example, a Deck object has a state space of 2.2×10^{68} due to the permutations of cards.

4. **Abstract State:** A subset of the concrete state space, focusing on specific characteristics of the object. Abstract states are more practical to consider when designing software, as they simplify the state space.
5. **Meaningful Abstract States:** These are abstract states that reflect characteristics important for the design of a software system. For example, "Non-zero Score" for a Player or "Empty" for a Deck are meaningful abstract states.
6. **Non-Meaningful Abstract States:** Abstract states like "Three Kings" for a Deck are not meaningful because they don't impact the object's usage or functionality in the system.
7. **Stateless vs. Stateful Objects:** Some objects, such as function objects, may not have fields and are considered stateless. Objects that have fields and hold values are stateful.
8. **Mutability and Statefulness:** This chapter focuses on objects that are mutable and stateful, though for immutable objects, the distinction between stateful and stateless is less clear because they only have one state.

4.3

Here is a summary of the text in enumerated points:

1. **Purpose of UML State Diagrams:** They represent how objects transition between abstract states in response to external events, providing a dynamic view of a software system.
2. **State Diagram Notation:** The diagram shows abstract states, transitions between them, and annotations describing methods that trigger these transitions.
3. **Deck Class Example:** The diagram models the states of a Deck class (Empty, Complete, Incomplete) and the transitions between these states.
 - **Empty State:** Initial state where the deck has no cards. A transition to the Complete state occurs through the shuffle method.
 - **Complete State:** Represents a shuffled deck. A self-transition through shuffle keeps it in this state. The deck moves to the Incomplete state upon drawing a card.
4. **Actions on Transitions:** Actions can be attached to transitions, describing what happens as a result of the transition (e.g., "remove card from the deck" for the draw event).

5. **Guards on Transitions:** Guards (conditions) on transitions specify when an event occurs. For example, a draw event can either keep the deck in the Incomplete state or transition to Empty, depending on the deck's size.
6. **Final State:** The final state is used to specify if an object must be in a certain state at the end of its lifetime. However, many objects can end in any state, making the final state element unnecessary.
7. **Impact on Design Decisions:** State diagrams help evaluate design decisions by showing how they affect the complexity of the abstract state space. Combining deck initialization and shuffling into one state simplifies the state space.
8. **Systematic Exploration:** State diagrams support systematic exploration of object behavior by considering all possible states and transitions, helping to identify overlooked paths (e.g., shuffling an incomplete deck).
9. **Incorrect Use of State Diagrams:** State diagrams should not be used to model data flow (with states representing processing and arrows showing data flow). States should be named as abstract states, not actions or verbs.
10. **Similarity to DFAs:** State diagrams resemble deterministic finite automata (DFA) in modeling stateful phenomena, but state diagrams are more abstract and used for software design, testing, and documentation. DFAs are strict theoretical models for computation.

4.4

Here is a summary of the text in enumerated points:

1. **Object Life Cycle:** An object's life cycle describes its transition through different abstract states, from initialization to eventual destruction by garbage collection. This life cycle is influenced by the object's design and can be complex.
2. **Complexity in Life Cycle:** As the design of a class grows more complex (with more abstract states), the life cycle of objects from that class becomes harder to understand, and state diagrams may not suffice to fully represent it.
3. **Challenges of Complex Life Cycles:** Objects with complex life cycles are difficult to use, test, and maintain. A good design principle is to minimize the state space to only what is necessary for the object to perform its intended tasks.
4. **Invalid and Useless States:** Some states in the state space may be irrelevant to the software system. For example, an unshuffled deck in a game of Solitaire is not useful. Introducing unnecessary states increases complexity without adding value.

5. **Avoiding Speculative Generality:** Some developers may argue for keeping unnecessary states for potential future use, but the cost in terms of developer time, debugging, and understanding the code often outweighs the speculative benefit.
6. **Unnecessary Stateful Information:** Storing redundant stateful information in instance variables, often for performance reasons or convenience, is a common design mistake. This can unnecessarily complicate the design.
7. **Example of Unnecessary Stateful Information:** In a Deck class, caching the size of the deck (e.g., storing it in a field) may seem efficient, but it introduces unnecessary complexity. The performance gain from this change is often insignificant compared to the loss in code simplicity.
8. **Trade-off Between Time and Space:** The decision to store redundant information for performance optimization (e.g., caching the size of a deck) is a trade-off between time (faster access) and space (more memory). In most cases, the time savings are not significant enough to justify the extra memory and complexity.
9. **Convenience and Redundant Information:** Storing redundant information for convenience (e.g., when a value is difficult to compute) is another bad practice. Information should only be stored if it contributes meaningfully to the object's intrinsic value.
10. **Temporary Field Antipattern:** Storing redundant information that is not crucial to the object's primary role can lead to the **Temporary Field** antipattern, which introduces unnecessary complexity and reduces code clarity.

4.5

Here is a summary of the text in enumerated points:

1. **Problem with null in Programming Languages:** In languages like Java and C++, the null value indicates the absence of a value for reference type variables, which can lead to problems such as NullPointerException when attempting to dereference a null reference.
2. **Inherent Ambiguity of null:** null references are ambiguous and can be interpreted in several ways:
 - As a temporarily uninitialized variable.
 - As an incorrectly initialized variable due to an overlooked code path.
 - As a flag indicating the absence of a value in the object's life cycle.

- As a flag requiring special interpretation.
- 3. **Avoiding null References:** To prevent expanding and complicating an object's state space, it's recommended to design classes without using null references. This reduces ambiguity and the risk of bugs associated with null dereferencing.
- 4. **Design Without Null References:** If a class can be designed so that a variable always has a value (e.g., non-null Rank and Suit for a Card), it is best to enforce this and avoid null references. This can be achieved through input validation or design by contract.
 - **Input validation:** Check for null values and throw an exception if found.
 - **Design by contract:** Use preconditions to stipulate that variables cannot be null.
- 5. **Modeling Absent Values:** Sometimes, the domain model requires representing the absence of a value, such as in the case of a "joker" card that has no rank or suit. Several strategies can be used to handle this:
 - **Null references:** Risk of NullPointerException.
 - **Bogus values:** Assigning meaningless values (e.g., Ace of Clubs), which can cause confusion and bugs.
 - **Special values in enumerated types:** Adding a value like INAPPLICABLE to an enum (e.g., Rank and Suit), which can lead to inconsistencies and off-by-one errors.
- 6. **Better Solutions for Absent Values:** Two solutions can avoid the problematic use of null references:
 - **Optional Types:** Use `Optional<T>` to represent a variable that may or may not have a value, making it explicit whether the value is present. The `Optional.empty()` method represents the absence of a value.
 - Use `Optional.of(value)` for non-null values.
 - Use `Optional.ofNullable(value)` for values that may be null.
 - **Unwrapping Optional:** Either change the class interface to return `Optional<T>`, or unwrap `Optional` in getter methods to preserve the interface while ensuring safety.

7. **NULL OBJECT Design Pattern:** The **NULL OBJECT** pattern uses a special object to represent the absence of a value. This pattern is based on polymorphism and is applicable when a type hierarchy is available. For example, a NullCardSource object could represent an unavailable CardSource:
 - **Polymorphism:** A NullCardSource behaves like any other CardSource object, simplifying client code.
 - **isNull() method:** This method can be used to check for the null object, but in practice, it might be unnecessary if all code already checks for emptiness.
8. **Efficient NULL OBJECT Implementation:** In Java 8, the NULL OBJECT can be implemented with a default method in an interface to avoid modifying all subclasses. Using an anonymous class, a constant NULL object can be created, which avoids the need for a separate NullCardSource class. This approach leverages polymorphism without creating extra classes.
9. **Practical Considerations:** By using the NULL OBJECT pattern, client code can handle absent values without needing special checks, making the design cleaner and avoiding edge cases like null dereferencing.

4.6

Here is a summary of the text in enumerated points:

1. **Principle of Minimizing Abstract State Space:**
 - When designing a class, it's important to minimize the abstract state space of objects, which means limiting the number of possible values that the fields of an object can take.
 - For example, a well-designed Deck class should have only a few meaningful abstract states (e.g., three states, not ten).
2. **Limiting Field Updates:**
 - To minimize the abstract state space, it's essential to limit how fields can be updated throughout the object's life cycle.
 - One way to do this is to prevent changing the value of a field after initialization, ensuring that the field remains constant.
3. **Using the final Keyword:**

- The final keyword in Java is used to mark a field as constant after it is initialized, ensuring that it can be assigned a value only once.
- Example: In the Card class, declaring aRank and aSuit as final makes the fields immutable after they are initialized.

4. Effect of final on Field Reassignment:

- Once a field is marked final, it cannot be reassigned, and any attempt to do so will result in a compilation error.
- This effectively makes objects immutable by ensuring that the field values do not change.

5. Impact of final on Reference Types:

- For reference types, final prevents reassignment of the reference, but does not prevent modification of the object being referenced if it is mutable.
- Example: In the Deck class, the aCards field is final, meaning it always refers to the same ArrayList, but the contents of that list can still be modified (e.g., adding cards).

6. Final Local Variables:

- Local variables (including method parameters) can also be declared final, but this is less common and generally less necessary.
- The primary use case for marking a local variable final is to make it clear that it should not be reassigned, especially in long or complex methods where the intent might not be clear.

7. Final Local Variables in Complex Methods:

- Declaring a local variable final can improve clarity in complex methods, making it explicit that the variable should not be reassigned.
- However, this should be used sparingly, as well-designed methods tend to be short and simple, and overly long methods are considered an antipattern.

4.7

Here is a summary of the text in enumerated points:

1. Identity of Objects:

- Identity refers to the unique reference or memory location that represents a particular object, even if it's not stored in a variable.
- In modern programming environments, object identity is abstracted, and tools like IDEs provide handles (e.g., object IDs in Eclipse) to represent object identity.

2. Object Identity in Programming:

- Object identity is tied to references or pointers to the object. For example, two Card objects can have different identities even if they represent the same card (e.g., Ace of Clubs), because they are separate objects in memory.

3. Equality of Objects:

- Equality refers to the concept of two objects being considered equal despite possibly being distinct objects. This needs to be defined by the programmer, as the default equality is based on identity.
- For simple objects, equality might be based on whether all fields have the same values, but for complex objects, equality might depend on other factors (e.g., the same elements in a set, despite internal order differences).

4. Overriding equals() Method:

- Java allows developers to define what it means for two objects of a class to be equal by overriding the equals(Object) method.
- The default equals() method compares objects by identity, but this can be customized to compare field values or other criteria.
- Example implementation of equals() for Card class: it compares the rank and suit fields to define equality.

5. Hash Code Consistency:

- When overriding equals(), it is crucial to also override the hashCode() method.
- The hashCode() method must return the same result for two objects that are considered equal by equals(). This is important for correct behavior in data structures like collections.

6. Uniqueness of Objects:

- Uniqueness ensures that no two distinct objects are considered equal, meaning there is only one instance of each object in a given context.
- If objects are guaranteed to be unique, equality becomes equivalent to identity, and objects can be compared using the == operator.
- Achieving strict uniqueness is difficult in Java due to features like metaprogramming and serialization, but design patterns and careful avoidance of these features can help ensure practical uniqueness in many cases.

4.8

Here is a summary of the FLYWEIGHT pattern in enumerated points:

1. Purpose of the FLYWEIGHT Pattern:

- The FLYWEIGHT pattern is used to manage collections of low-level, immutable objects.
- It helps ensure object uniqueness, especially when instances of a class are heavily shared in a system (e.g., Card objects in Solitaire).

2. Core Idea of FLYWEIGHT:

- The key idea is to control the creation of objects of a class (called the flyweight class) to ensure that duplicate objects do not exist.
- Flyweight objects are managed through an access method that either returns an existing object or creates a new one if it doesn't exist.

3. Components of FLYWEIGHT:

- **Private constructor:** Prevents client code from creating instances of the flyweight class.
- **Static flyweight store:** A collection of flyweight objects, often implemented as a static field.
- **Static access method:** Ensures only one unique instance of each flyweight object is created or retrieved by checking the store.

4. Example with Card Class:

- A non-flyweight version of Card allows distinct but equal instances.

- To implement flyweight, the constructor is made private, preventing arbitrary creation of Card instances.
- A static 2D array (CARDS) is used to store the flyweight objects, where each card is indexed by its rank and suit.

5. Flyweight Store Implementation:

- The store is initialized in a static block, pre-populating the collection of cards.
- A simple access method get() retrieves the unique card instance based on rank and suit.
- The method is static since the store is static, ensuring it is accessed without needing an instance of Card.

6. Flyweight Factory:

- The combination of the flyweight store and the access method is referred to as the flyweight factory.
- The access method ensures that flyweight objects are not duplicated, and only one instance of each unique object is created.

7. Identification Key for Flyweight Objects:

- The identification key for accessing a flyweight object (e.g., rank and suit for Card) should be unique and cannot be the object itself.
- A flawed example would involve using the object instance as the key, leading to an infinite cycle.

8. Pre-initialization vs. Lazy Initialization:

- Flyweight objects can either be pre-initialized (as in the example with 52 playing cards) or lazily created when requested.
- In the lazy initialization approach, the access method checks if an object exists; if not, it creates it based on the key.

9. Immutable vs. Mutable Flyweights:

- The FLYWEIGHT pattern is especially useful for managing immutable objects.
- While it can be applied to mutable objects, this can be error-prone, as the mutable state might inadvertently affect object identity.

- It is crucial that the immutable part of the flyweight objects' state (defining identity) cannot be changed.

4.9

Here's a summary of the SINGLETON design pattern in enumerated points:

1. Purpose of the SINGLETON Pattern:

- Ensures that only one instance of a class exists throughout the execution of the program.
- Useful when managing a single object that holds state or information needed by various parts of the code (e.g., game state in a card game).


2. Solution Template for SINGLETON:

- **Private constructor:** Prevents clients from creating multiple instances of the class.
- **Global variable:** Holds a reference to the single instance of the singleton object.
- **Accessor method:** Typically called `instance()`, returns the singleton instance (this method is optional if the instance is a public constant).

3. Example Implementation:

- A singleton `GameModel` class in a card game might look like:

java

 Copy code

```
public class GameModel {  
    private static final GameModel INSTANCE = new GameModel();  
    private GameModel() { ... }  
    public static GameModel instance() { return INSTANCE; }  
}
```

4. Difference from FLYWEIGHT:

- The SINGLETON pattern ensures a **single instance** of a class, while FLYWEIGHT ensures **unique instances**.
- Singleton objects are typically **stateful and mutable**, whereas flyweight objects are usually **immutable**.

5. Common Mistake in Implementing SINGLETON:

- Storing a reference in a static field (INSTANCE) without preventing independent object creation.
- This leads to misleading usage, where users might assume there is only one instance when there could be more.

6. Private Constructor or Enum for Singleton:

- The **private constructor** is the classic approach to enforce the singleton constraint.
- **Using an enum** for singletons is an alternative (as suggested in *Effective Java*), but can be confusing because enums are meant for finite sets of values, not singleton instances.

```
java

public enum GameModel {
    INSTANCE;
    public void initializeGame() {}
}
```

7. Challenges with Singleton Pattern:

- **Global instance:** Singletons can be accessed from anywhere, making them prone to misuse and leading to high coupling in the code.
- **Testing difficulties:** Since singletons control their lifecycle and persist throughout the application's lifetime, they are difficult to replace or mock in tests.

8. Alternatives to SINGLETON:

- **Dependency injection** is a common alternative to Singleton for managing unique objects, though it doesn't prevent multiple instances by itself.
- **Methodical programming and documentation** can help enforce the singleton constraint in dependency injection scenarios.

9. Evaluating the Need for SINGLETON:

- It is important to recognize when only one instance of a class should exist and evaluate whether a Singleton or another strategy (e.g., dependency injection) is more appropriate.
- In some cases, the Singleton might be the best solution, but other strategies should be considered depending on the context.

4.10

Here's a summary of the concepts related to nested classes, inner classes, and anonymous classes in Java:

1. Types of Nested Classes:

- **Static Nested Classes:** Not linked to an outer instance, mainly used for encapsulation and code organization.
- **Inner Classes:** Declared within another class and provide behavior involving an instance of the enclosing class.
 - **Anonymous Classes:** A subclass with no explicit name, typically used for function objects or callbacks.
 - **Local Classes:** Similar to anonymous classes but have a name and can be defined within a method.

2. Inner Classes and State:

- Inner classes have an implicit reference to the outer class instance, known as the **outer instance**.
- Example: In the Deck and Shuffler classes, the inner class Shuffler maintains a reference to the outer class Deck via Deck.this.
- **Accessing Outer Class:** The inner class can access the methods and fields of the outer class using Deck.this.
- Example:

java

```
public class Deck {
    public void shuffle() { ... }
    public class Shuffler {
        private int aNumberOfShuffles = 0;
        public void shuffle() {
            aNumberOfShuffles++;
            Deck.this.shuffle(); // Accessing outer Deck instance
        }
    }
}
```

3. Design Implications of Inner Classes:

- The inner class contributes to the state space of the outer class.
- Proper design ensures that the abstract state of inner classes represents the state of the outer class without introducing unnecessary complexity.
- **State management:** Inner classes can maintain additional state (like `aNumberOfShuffles` in the `Shuffler` example).

4. Static Nested Classes:

- Unlike inner classes, **static nested classes** do not have a reference to the outer instance.
- They are typically used for **encapsulation** or **organization** without linking to an outer class's instance.

5. Anonymous Classes:

- Anonymous classes are often used for implementing **function objects** or **callbacks**.
- They can refer to local variables in the method from which they are created, even though the lifecycle of the anonymous class is independent of the method.

6. Closure in Anonymous Classes:

- In Java, when an anonymous class accesses local variables of the enclosing method, Java **captures** those variables and stores them in fields within the anonymous class.
- This mechanism allows the anonymous class to "close over" local variables, preserving their values after the method exits.
- Example of a factory method with an anonymous class:

```
java

public class Deck {
    public static Comparator<Deck> createRankComparator(Rank pRank) {
        return new Comparator<Deck>() {
            public int compare(Deck pDeck1, Deck pDeck2) {
                return countCards(pDeck1) - countCards(pDeck2);
            }
            private int countCards(Deck pDeck) { ... }
        };
    }
}
```

7. Handling Local Variables:

- When the anonymous class uses local variables from the enclosing method, those variables are copied into fields within the anonymous class.
- **Closure:** The technique where a method's local variables are captured and referenced in the anonymous class.

8. Restrictions on Variable Reassignment:

- Java prevents **reassigning** variables from the enclosing scope within anonymous classes to avoid unexpected behavior, ensuring stable references to local variables.

This summary highlights key aspects of how nested, inner, and anonymous classes work in Java, including their state management and lifecycle implications.

Chapter 5

5.1

Here's a summarized list of the key points regarding software quality, unit testing, and related concepts:

1. Software Quality Problems:

- Bugs often arise when code doesn't behave as expected, and programmers remain unaware of the mismatch between expectations and reality.
- Example: A bug in JetUML caused the directory structure to disappear because of a flawed conditional statement in the code.
 - Issue: The code was incorrectly filtering directories from the file chooser, while directories should have been included.
 - Fix: Changing logical operators to allow directories to appear in the chooser.

2. Unit Testing:

- **Definition:** Unit testing involves testing small parts of the code (unit under test, UUT) in isolation.
- **Purpose:** Detects issues by comparing the test results against an expected result (oracle).
- **Structure:** A unit test involves executing the UUT with input data and checking if the output matches the oracle (expected result).

3. Unit Under Test (UUT):

- A UUT can be a method, class, initialization statement, or specific path in the code.
- Example: Testing the `Math.abs(5)` method, where the UUT is `Math.abs(int)`, the input data is 5, and the oracle is 5.

4. Implicit Arguments in Unit Testing:

- When testing instance methods, remember that the instance itself (implicit argument) also affects the behavior.

- Example: The `sameColorAs(Suit)` method in the `Suit` enum compares the color of two suits based on their order.
 - Test Case: `CLUBS.sameColorAs(HEARTS)` verifies if the method works as expected.

5. Exhaustive Testing:

- Exhaustive testing involves testing all possible input combinations. For example, testing all pairs of suits for the `sameColorAs` method in the `Suit` enum.
- Exhaustive testing is often impractical, especially in large or complex systems.

6. Regression Testing:

- Unit tests not only help detect bugs in new code but also verify that previously tested behavior still works after changes (e.g., when the order of the suits changes).
- A failed test after code changes indicates that the assumptions of the code have been violated, highlighting the risk of introducing bugs.

7. Limitations of Unit Testing:

- Unit testing cannot guarantee code correctness; it only confirms that the specific test case passes as expected.
- Unit testing is not a complete verification technique and cannot cover all possible scenarios.
- Additional techniques and practices are required to ensure comprehensive code correctness.

This summary outlines the primary concepts of unit testing, its limitations, and its importance in detecting bugs and verifying code behavior.

5.2

Here's a summarized list of the key points regarding unit testing frameworks, specifically JUnit:

1. Automation of Unit Testing:

- Unit testing is typically automated, and to automate testing, developers write code that tests other code.
- Unit testing frameworks support this automation by handling mundane tasks like collecting tests, running them, and reporting results.

2. Key Constructs in Unit Testing Frameworks:

- **Test Cases:** Individual tests that are executed to verify specific functionality.
- **Test Suites:** Groups of related test cases that are run together.
- **Test Fixtures:** Setups needed before running the tests (e.g., initializing test data).
- **Assertions:** Methods used to verify that the code under test produces the expected result.

3. JUnit Framework:

- JUnit is the dominant unit testing framework for Java.
- It provides tools for writing structured tests and automating the execution of those tests.

4. JUnit Test Method Structure:

- In JUnit, unit tests are mapped to methods annotated with `@Test`.
- Example code:

```
java

@Test
public void testAbs_Positive() {
    assertEquals(5, Math.abs(5));
}
```

- The `@Test` annotation marks methods as unit tests to be run by the framework.

5. Assertions in JUnit:

- Assertions, like `assertEquals()`, are used to verify that the code behaves as expected.

- If an assertion fails (i.e., the predicate is false), the test fails.

6. **JUnit Test Runner:**

- JUnit includes a test runner, which scans input code, identifies the tests, executes them, and reports the results.
- The test runner provides feedback on whether tests passed or failed.

7. **Handling Test Failures:**

- Example of a failed test in JUnit:
 - A test verifying `Math.abs(Integer.MIN_VALUE)` failed, because the absolute value of `Integer.MIN_VALUE` does not equal `Integer.MAX_VALUE` due to physical constraints.
 - This behavior is documented in the Javadoc for `Math.abs()`.

8. **Bug Detection and Corner Cases:**

- Unit tests are useful not only for detecting bugs but also for uncovering edge cases or corner cases that may be overlooked.

This summary highlights the key concepts of using JUnit for unit testing, including how tests are structured, executed, and reported.

5.3

1. **Test Suite Definition:** A test suite is a collection of tests for a project. By default, it includes all unit tests for the production code.
2. **Subset of Tests:** There may be times when only a subset of tests should be run, such as focusing on a specific feature or saving time.
3. **Mechanisms for Subset Execution:** Unit testing frameworks like JUnit allow developers to define arbitrary test suites or run subsets of tests. For example, JUnit provides a `@Suite` construct to list test classes for execution together.
4. **JUnit Integration with IDEs:** The JUnit plugin for the Eclipse IDE enables running tests for a specific package or even a single test class.
5. **Test Suite Design Focus:** The chapter primarily focuses on writing tests rather than on executing them, as test execution is somewhat independent from the test design process.

6. **Organizing Test Suites:** A common approach for organizing tests in Java is to have one test class per project class, with each test class collecting tests related to the class or its methods.
7. **Test Code Organization:** It is a standard practice to place test code in a separate source folder, with a package structure mirroring that of the production code.
8. **Package Scope and Access:** In Java, classes within the same package can access non-public (but non-private) members of each other, even if they are located in different file system directories. This allows tests to access the production code's non-public members while maintaining separation.

5.4

Here is the summary in enumerated points:

1. Metaprogramming in Unit Testing:

- Unit testing frameworks like JUnit use metaprogramming to detect and run test methods automatically.
- This is done through annotations (e.g., `@Test`) which mark methods as tests.
- Metaprogramming refers to writing code that manipulates or operates on other code, which is essential for test automation.

2. Reflection in Java:

- Java's metaprogramming feature is called **reflection**. It is used to inspect and manipulate code at runtime.
- The **Class** class in Java provides access to metadata about classes, methods, and fields.
- Reflection can be used to obtain class references, invoke methods, and modify field values dynamically.

3. Introspection:

- Introspection is the process of obtaining information about a class or method in a program using reflection.
- **Class.forName()**, **Card.class**, and **getClass()** are examples of methods used to obtain class references.

- Class literals (e.g., **Card.class**) are less error-prone and can be used when the class type is known at compile time.

4. Program Manipulation:

- Reflection allows manipulation of program behavior, such as changing field values, invoking methods, and creating new instances.
- **getDeclaredConstructor()**, **setAccessible(true)**, and **newInstance()** allow access to private members and the creation of new objects despite access restrictions.

5. Metaprogramming for Tests:

- Metaprogramming techniques can be used to facilitate writing and executing tests by manipulating code elements (e.g., fields, methods).
- Example: Changing the value of a private field (`rankField.set(card, Rank.ACE)`) using reflection to modify a class instance during a test.

6. Program Metadata with Annotations:

- Java supports adding metadata to code elements using **annotations**. For example, **@Test** is used to mark test methods in JUnit.
- Annotations provide structured, type-checked metadata that can be accessed by the compiler, IDEs, and testing frameworks.
- Annotations can be accessed using reflection and are a safer, more reliable alternative to using comments for metadata.

7. Advantages of Annotations:

- Annotations are type-safe and checked by the compiler, ensuring consistent application of metadata (e.g., marking test methods).
- They provide a structured way to attach metadata to code, unlike unstructured comments, reducing human error.

8. Accessing Annotations through Reflection:

- Annotations can be read and manipulated via reflection, allowing tools to inspect and act on the metadata attached to code.

This summary highlights key concepts around metaprogramming, reflection, and annotations as they relate to unit testing in Java.

5.5

Here is a summary in enumerated points of the given text:

1. **Unit Testing Challenges:** Writing unit tests for non-trivial classes is a creative process and there is no standard formula for writing them. Different open-source communities follow different styles and techniques for testing.
2. **Basic Principles of Unit Tests:**
 - **Fast:** Unit tests should execute in a few seconds to be useful. Tests should avoid long-running operations like device I/O or network access.
 - **Independent:** Each test should be able to run in isolation, without depending on other tests. Test independence is essential as test suites evolve.
 - **Repeatable:** Unit tests should produce the same results regardless of the environment, without relying on environment-specific properties.
 - **Focused:** Tests should focus on small parts of code, ideally checking a single input on a single method call to make debugging easier.
 - **Readable:** Test code should be easy to read and understand, with clear identification of the unit under test, input data, and expected results.
3. **Example Unit Test:**
 - A method `canMoveTo` of the class `FoundationPile` is tested for the case when the pile is empty. The test ensures that an Ace can be moved to an empty pile, but a Three cannot.
 - The test is fast, independent, repeatable, focused, and readable.
4. **Test Coverage and Duplication:**
 - The initial test only covers a single case (empty pile). To improve coverage, another test is added for when the pile is not empty and the card is of the same suit.

- The test code starts to duplicate the creation of the FoundationPile and Card objects, which is inefficient.

5. **Test Fixture:**

- To avoid code duplication, commonly used objects can be declared as fields in the test class, known as a test fixture. This practice avoids repeated initialization code in each test method.
- JUnit 5 automatically ensures that a fresh version of the test class is instantiated before each test method, preserving test independence.

6. **Improved Version Using Test Fixture:**

- The test class is refactored to use a test fixture, with common Card objects stored as static fields and the FoundationPile instance as a regular field.
- This approach avoids code duplication and improves test readability while maintaining test independence. However, using a fixed variable name for the pile (e.g., aPile) may reduce flexibility in naming but still maintains clarity.

7. **Benefit of Test Fixture:**

- Using a test fixture improves code organization and readability, though it may slightly reduce flexibility in naming variables. The trade-off between clarity and flexibility is minimal in this case.

5.6

Here is a summary in enumerated points of the given text:

1. **Purpose of Unit Testing:** Unit tests aim to verify that the unit under test behaves as expected. It is important to test only when the input meets the method's preconditions, as testing with invalid input leads to unspecified behavior.
2. **Design by Contract:** When using design by contract, testing code with input that violates the method's preconditions does not make sense. For example, if a method's precondition specifies that the pile should not be empty, testing the method on an empty pile is meaningless.
3. **Example: FoundationPile's peek Method:**
 - The method peek in FoundationPile has a precondition that the pile must not be empty.

- Testing peek with an empty pile is unnecessary because the behavior is unspecified according to the precondition.
- 4. **Exception Handling in Methods:** If raising exceptions is part of the method's interface, such as throwing an `EmptyStackException` when calling `peek` on an empty pile, this behavior should be explicitly tested.
- 5. **Test for Expected Exceptions:**
 - When an exception is part of the expected behavior, a test should be written to check that the exception is thrown.
 - JUnit 5 provides the `assertThrows` method to check for exceptions. This method takes the expected exception type and an executable action.
- 6. **Example of Testing for Exceptions:**
 - The `testPeek_Empty` test verifies that calling `peek` on an empty pile correctly raises an `EmptyStackException`.
 - This requires an anonymous class to implement the `Executable` interface, which allows execution of the code expected to cause the exception.
- 7. **Using Lambda Expressions with `assertThrows`:**
 - Lambda expressions can simplify the setup for testing exceptions.
 - The `testPeek_Empty_LambdaExpression` test demonstrates the use of a lambda expression to trigger the exception, making the code more compact and readable.

5.7

Here is a summary of the given text in enumerated points:

1. **Design Issues in Testing:** Sometimes, to write a unit test, we need functionality not part of the class's interface (e.g., a `size()` method). While such functionality may not be necessary for production code, it can be very useful for testing.
2. **Encapsulation and Testing:** It's recommended to maintain the best possible encapsulation in production code. Testing code can work around this constraint, rather than adding unnecessary methods to the class's interface.
3. **Workaround for Missing Functionality:** A common solution to test functionality not exposed in the interface is to create helper methods within the test class. For

example, a `size()` method can be created in the test class to check the number of cards in a `FoundationPile` without modifying the class interface.

4. **Testing Private Methods:** The question of how to test private methods is debated. There are two common perspectives:
 - **Indirect Testing:** Private methods should not be tested directly, as they are internal parts of other accessible methods. Their behavior is tested by calling public methods that use them.
 - **Direct Testing:** Private methods can sometimes be tested separately, especially when they perform valuable and isolated tasks. This is more acceptable when the method's logic is clear and doesn't depend too heavily on the internal structure of the class.
5. **Bypassing Private Method Access Restrictions:** If testing a private method directly is necessary, it can be done using metaprogramming techniques like Java's reflection API to bypass access restrictions.
6. **Example of Testing a Private Method:**
 - If `FoundationPile` has a private method `getPreviousCard()`, it can be tested by creating a helper method in the test class using reflection. The helper method uses `getDeclaredMethod()` and `setAccessible(true)` to invoke the private method.
 - The test class then uses this helper method in the test case to check the behavior of the private method (e.g., `testGetPreviousCard_empty()`).
7. **Reflection in Test Code:** The test code uses reflection to invoke the private method, making the test look similar to normal test code but with an added step to bypass the private access modifier using metaprogramming techniques.

5.8

Here is a summary of the text in enumerated points:

1. **Unit Testing Challenges:** Unit testing aims to test small parts of code in isolation. However, certain factors can complicate this:
 - When a piece of code triggers a large amount of other code.
 - When behavior depends on the environment (e.g., system fonts).
 - When the code involves non-deterministic behavior (e.g., randomness).

2. **Example Scenario (Solitaire Application):** The GameModel class in a simplified Solitaire application has a method tryToAutoPlay() that delegates the task of computing the next move to a strategy object, creating several testing challenges:
 - The strategy might involve complex behavior, making it difficult to test the small part of code in isolation.
 - The strategy may introduce randomness.
 - The specific strategy used by the game is unknown, making it hard to predict results.
 - It's unclear how testing the tryToAutoPlay method differs from testing individual strategies.
3. **Solution - Focus on Delegation:** The core responsibility of GameModel#tryToAutoPlay() is not to compute the next move, but to delegate this task to the strategy. Therefore, the unit test should focus on verifying that the method properly delegates the task, rather than testing the strategy itself.
4. **Using Stubs:** A stub is a simplified version of an object that mimics its behavior enough to support testing. In this case, a stub strategy can be created to simulate the behavior of a real strategy, allowing the unit test to verify that tryToAutoPlay() delegates correctly:
 - The stub strategy implements the PlayingStrategy interface and tracks whether its computeNextMove method is called, while returning a dummy NullMove object.
5. **Test Implementation:**
 - A StubStrategy is defined inside the test class to mimic the behavior of the real strategy.
 - The StubStrategy has a flag aExecuted to track whether the computeNextMove method was called.
 - In the test, a Field object is used with metaprogramming to inject the stub strategy into the GameModel instance.
6. **Verifying the Delegation:**
 - After injecting the stub, the test calls tryToAutoPlay() on the GameModel instance and checks that the computeNextMove method was called by asserting strategy.hasExecuted().

7. **Sophisticated Use of Stubs:** The use of stubs in unit testing can be quite sophisticated, and frameworks exist to support this approach when needed.

5.9

Here's a summarized version of the text in enumerated points:

1. **Test Case Selection Challenge:**

- Unit testing requires selecting test inputs, but exhaustive testing is often impractical due to the vast input spaces, such as the astronomical number of card arrangements in a deck.
- The goal is to select a minimal set of test cases that provide maximal coverage of the code.

2. **Test Case Selection Problem:**

- The main challenge is efficiently selecting test cases that maximize testing coverage with minimal input sets.
- There's no universally agreed definition of what constitutes sufficient testing, but there is a large body of research and practical experience on the subject.

3. **Approaches to Test Case Selection:**

- **Functional (Black-box) Testing:**
 - Focuses on testing the UUT based on external specifications, without accessing the source code.
 - Good for revealing specification issues and missing logic.
- **Structural (White-box) Testing:**
 - Focuses on testing based on the source code's internal structure.
 - Reveals issues related to implementation details that may not be visible from the specification.

4. **Test Coverage:**

- Coverage metrics help determine how much of the code is tested.
- These metrics are used to evaluate the effectiveness of a test suite and can guide where to focus future testing efforts.

5. **Common Coverage Metrics:**

- **Statement Coverage:**

- Measures the proportion of executable statements that are run during tests.
- While simple, it has limitations because it doesn't account for compound expressions and can miss important code paths.
- For example, with 67% statement coverage, only some branches of a conditional statement might be tested.

- **Branch Coverage:**

- Measures the proportion of decision points (branches) in the code that are executed during tests.
- Provides a stronger metric than statement coverage because it tests both outcomes of decisions (true and false branches).
- With the same test cases, 37.5% branch coverage may be achieved, and additional tests could increase this to 87.5%.

- **Path Coverage:**

- Measures the proportion of execution paths through the code that are tested.
- It subsumes most other coverage metrics and provides a close approximation of complete behavior.
- While conceptually useful, path coverage is often impractical for complex code due to the unbounded number of paths, especially with loops. In the case of `canMoveTo`, there are five paths, and the two-test suite achieved 80% path coverage.

6. **Benefits of Coverage Metrics:**

- Coverage metrics help ensure that important parts of the code are tested.
- Branch coverage is particularly valuable and widely supported by testing tools, as it subsumes statement coverage and is relatively easy to interpret.

7. **Path Coverage in Practice:**

- Path coverage can provide a more comprehensive understanding of test effectiveness, but its practical use is limited in complex scenarios with loops

Chapter 6

6.1

Here's a summarized version of the text in enumerated points:

1. **Managing Complexity in Software Design:**

- A common strategy for managing software complexity is to define larger abstractions in terms of smaller ones, following the divide and conquer approach.

2. **Object Composition:**

- Object composition involves one object storing references to other objects, assembling separate parts (classes, methods, objects) to create a functional application.

3. **Two Main Uses of Composition:**

- **Representation:** When an abstraction is a collection of other abstractions (e.g., a deck of cards composed of card instances).
- **Decomposition (Delegation):** To break down complex classes into smaller, manageable parts by delegating specific tasks to specialized objects, preventing the creation of "God classes" that are overly complex and violate good design principles.

4. **Delegation and Aggregation:**

- **Delegation:** Involves an object delegating tasks to another object that handles a specific function, promoting loose coupling.
- **Aggregation:** A looser form of composition where one object holds references to other objects that provide services to it.

5. **Transitive Property of Composition:**

- Composition is transitive: an object composed of other objects can itself be a component or delegate of a parent object, forming complex aggregates from simpler components.

6. **Representation vs. Delegation:**

- It's important to distinguish between using composition for representation (essential parts of the aggregate object) and delegation (service providers for

the aggregate object). However, these purposes are not mutually exclusive and often overlap.

7. Example of GameModel Class:

- In the Solitaire example, the GameModel class aggregates instances like Deck, CardStack, Foundations, and Tableau.
- Instead of having a Deck class directly handle all card operations, CardStack provides a narrow interface for managing stacks of cards.

8. Using Composition to Avoid Complexity:

- Instead of managing all aspects within a single class (which could become overly complex), the design delegates specific tasks to other classes (e.g., Tableau handles determining card visibility in the tableau).

9. UML Representation:

- Composition in UML is typically represented with a diamond on the class that holds the reference to other objects. The distinction between aggregation (white diamond) and composition (black diamond) is often debated, and the author opts to use only the white diamond for both.

10. Challenges of Composition:

- While object composition can improve design, unprincipled use of composition can lead to overly complex code. It's important to use design patterns and structured design plans to maintain clarity and organization.

11. Goal of Using Composition:

- The goal is to develop the skill of using composition strategically, according to a clear and rational design plan, to manage complexity effectively.

6.2

Here's a summarized version of the text in enumerated points:

1. Goal of Composition:

- In situations where groups of objects need to behave like a single object, composition provides a solution. For example, in a card game, multiple sources of cards can be treated as a single unified source.

2. CardSource Interface:

- The CardSource interface defines methods for drawing a card and checking if the source is empty, enabling polymorphic behavior for different types of card sources.

3. **Static Class Definitions:**

- Initially, different card source configurations can be defined using multiple classes (e.g., Deck, MultiDeck, FourAces). However, this leads to problems:
 - **Combinatorial Explosion:** Too many class definitions for each possible configuration.
 - **Clutter:** Code becomes unnecessarily cluttered with classes that may be rarely used.
 - **Rigidity:** It's hard to handle new configurations that weren't anticipated at compile-time.

4. **Dynamic Configuration via Composition:**

- To overcome these limitations, the COMPOSITE design pattern is applied. This allows dynamically combining card sources without requiring static class definitions for every configuration.

5. **COMPOSITE Design Pattern:**

- **Component:** Represents the common interface (e.g., CardSource).
- **Leaf:** Implements the CardSource interface (e.g., Deck, FourAces).
- **Composite:** Aggregates multiple CardSource objects and implements the CardSource interface, allowing it to be treated like a leaf by client code.

6. **Advantages of COMPOSITE:**

- A composite object can aggregate multiple other objects, including other composites.
- Client code can interact with composites and leaves in the same way by depending on the common CardSource interface, not concrete types.

7. **Method Implementations for Composite:**

- In a composite, methods like isEmpty() involve iterating through aggregated elements. For draw(), the method iterates through sources until it finds a card to draw.

- The composite class ensures the preconditions of methods are met, e.g., `draw()` assumes `isEmpty()` returns false.

8. Adding Components to a Composite:

- Components can be added to a composite in two ways:
 - **Using an add method:** Allows modifying the composite at runtime, but adds complexity.
 - **Using a constructor:** Initializes the composite with a list of `CardSource` objects, avoiding runtime modifications.

9. Design Considerations:

- If the composite does not need to change dynamically, initializing it once via the constructor is often preferred. This leads to a simpler design.
- The choice between using an add method or constructor depends on whether runtime modifications are necessary.

10. Practical Aspects:

- The location of composite creation in client code and maintaining object graph integrity are critical to the correct application of the COMPOSITE pattern.
- The structure of the pattern alone is not enough; the object graph must be properly managed to avoid issues like shared or self-referencing objects that can lead to unmanageable behavior.

6.3

Here is a summarized version of the text in enumerated points:

1. Composition in Software Design:

- Composition refers to how objects collaborate with each other in software design. It impacts the interactions and method calls between objects.
- It contrasts with static design decisions, which define how classes depend on each other.

2. Use of Sequence Diagrams:

- Sequence diagrams model the dynamic behavior of a software system, showing how objects interact through method calls during execution.

- They represent a specific execution, similar to what one might observe while stepping through code in a debugger.

3. **Example: isEmpty() Call on CompositeCardSource:**

- A sequence diagram is used to model a call to isEmpty() on a CompositeCardSource instance, showing how the method recursively checks its elements.
- This dynamic behavior cannot be captured in a static class diagram, making sequence diagrams valuable for understanding interactions.

4. **Life Lines and Objects:**

- Objects in sequence diagrams are represented with life lines, which show the duration of an object's existence (from creation to garbage collection).
- The diagram shows interactions between a client and a CompositeCardSource with its component objects, created before the interaction begins.

5. **Message Representation:**

- Method calls between objects are represented as messages with directed arrows, labeled with the method name and optional arguments.
- Constructor calls are marked as <<create>>.
- Activation boxes (thicker white boxes) show when an object's method is active on the execution stack.

6. **Return Control:**

- Return control from a method is represented with a dashed directed arrow.
- Return edges are optional but can aid understanding, especially in complex sequences of messages.

7. **Iterator Pattern Example:**

- Sequence diagrams can also model patterns like the Iterator pattern.
- In the case of the Deck class, the iterator() call is delegated to the Stack object, which creates the iterator. The iterator is then returned to the client.
- Return edges show the flow of the Iterator object and the next() method result.

8. Representation of Types:

- Objects in sequence diagrams can be represented using either their concrete type or one of their supertypes (e.g., using `Iterable<Card>` for a `Deck`).
- For anonymous objects, such as iterators, the interface type is often used.

9. Modeling Specific Executions:

- Sequence diagrams represent specific executions, not all possible executions.
- They focus on the interactions relevant to the scenario being modeled, omitting irrelevant details like library method calls or variations in execution paths.

10. Exclusion of Details:

- UML sketches generally omit detailed elements like loops, conditionals, asynchronous calls, and insignificant method calls to maintain clarity and focus on the core interactions.

6.4

Here is a summarized version of the text in enumerated points:

1. Need for Extra Features:

- Sometimes objects need additional features, such as logging card draws or memorizing drawn cards, without altering the core behavior.
- Two potential design solutions are considered: specialized classes and multi-mode classes.

2. Specialized Class Solution:

- A separate class is created for each feature, e.g., `LoggingDeck` for logging card draws, and `MemorizingDeck` for memorizing drawn cards.
- This approach lacks flexibility because it doesn't allow toggling features at runtime, and objects cannot change type dynamically.

3. Multi-Mode Class Solution:

- A single class implements all features and uses a flag to control the mode (e.g., `Logging`, `Memorizing`, or both).

- This solution can toggle features but leads to complex state management and violates principles like separation of concerns, often resulting in a "God Class" with poor extensibility.

4. **Decorator Design Pattern:**

- The **Decorator** pattern offers a flexible solution, allowing additional features to be added to objects without modifying their core behavior.
- The pattern involves creating decorator classes that wrap a component, enabling dynamic feature addition without changing the original object.

5. **Decorator Solution Template:**

- Decorators aggregate objects of the component type (e.g., CardSource) and implement the same interface.
- They delegate method calls to the wrapped object and add new behavior (e.g., logging or memorizing card draws).
- Example: MemorizingDecorator delegates the draw action but also stores the drawn card.

6. **Combining Decorations:**

- Multiple decorators can be stacked together, each adding different features. For example, a deck can be decorated first with MemorizingDecorator, then with LoggingDecorator for combined behavior.
- The sequence diagram illustrates this delegation process.

7. **Constraints on Decorations:**

- Decorations must be independent and additive. The Decorator pattern should not remove features but only add new ones.
- The combination of features should be flexible without complex rules.

8. **Implementation in Java:**

- In Java, the decorated object should be stored in a final field, ensuring it remains the same throughout the decorator's lifetime.
- The decorator should be initialized via the constructor, maintaining consistency.

9. **Identity Change with Decorators:**

- When an object is decorated, it gains a new identity. A decorated object is not the same as the undecorated one, which could cause issues in systems relying on object identity.
- This identity change must be carefully considered, especially when equality and identity are critical.

6.5

Here is a summarized version of the text in enumerated points:

1. **Co-existence of Decorator and Composite Patterns:**

- The **Decorator** and **Composite** patterns, though distinct, can co-exist in a type hierarchy.
- If both patterns implement the same component interface, they can work together to solve design problems based on composition.

2. **Class and Object Diagrams:**

- A class diagram illustrates a type hierarchy with a leaf, a composite, and two decorators.
- An object diagram shows an example of both a decorated composite and a composite of a decorated object.

3. **Classic Scenario for Decorator and Composite:**

- The classic use case for these patterns is in the development of a drawing feature (e.g., for drawing tools or slideshow applications).
- In this scenario, the component type is a `Figure` with a `draw()` method. Leaf classes represent concrete figures like rectangles, ellipses, and text boxes.

4. **Composite Figure Design:**

- A `CompositeFigure` supports grouping individual figures into an aggregate, which can itself be grouped with other figures or groups.
- This grouping allows treating a group as a single figure in the overall structure.

5. **Decorator Figure Design:**

- The **Decorator** pattern allows enhancing figures by adding additional features, such as a border.

- The decorator modifies the behavior of the figure by decorating it while still maintaining the core behavior.

6. Draw Method in Composite and Decorator:

- In the **Composite** pattern, the draw() method calls the draw() method of each contained figure:

```
java

public void draw() {
    for (Figure figure : aFigures) {
        figure.draw();
    }
}
```

- In the **Decorator** pattern, the draw() method first delegates to the original figure's draw() method and then adds extra behavior (like drawing a border):

```
java

public void draw() {
    aFigure.draw();
    // Additional code to draw the border
}
```

7. Illustration of Behavior Realization:

- The behavior of both patterns' implementations (composite and decorator) is clearly shown through the draw() method in their respective designs.

6.6

Here is a summarized version of the text in enumerated points:

1. Dynamic Structures and Object Identity:

- Working with combinations of objects in complex object graphs has implications for object identity and copying designs.

2. Limitations of Copy Constructors:

- Copy constructors work for many cases but are limited in polymorphic designs, where the concrete types of objects are unknown.

- For example, copying objects in a list of polymorphic types (like CardSource) requires knowing their concrete types, which complicates the process.

3. Problems with Static Copy Constructor Approach:

- Using branching statements to call different constructors for each possible concrete type breaks polymorphism and reduces extensibility. This approach becomes unwieldy when new subtypes are introduced.

4. Need for Polymorphic Copying:

- A polymorphic solution is needed to copy objects without knowing their concrete types.
- This is achieved by adding a copy() method to the CardSource interface, requiring all subclasses to implement it.

5. Implementing the copy() Method:

- The copy() method allows creating a deep copy of objects without knowing their concrete type.
- Concrete classes must implement this copy() method to ensure the correct copying behavior for each object type.

6. Copying Non-Recursive Structures (Deck):

- For non-recursive structures like Deck, the copy() method is simple:

```
java

public Deck copy() {
    Deck copy = new Deck();
    copy.aCards = new CardStack(aCards);
    return copy;
}
```

- This method uses the CardStack copy constructor to copy the internal structure of the Deck.

7. Covariant Return Types:

- Java 5 introduced **covariant return types**, allowing the return type of copy() to be more specific than the interface's method return type.

- This avoids unnecessary downcasting when copying objects and allows direct assignment to the concrete type:

```
java  
  
Deck copy = deck.copy();
```

8. Copying Recursive Structures (Decorators):

- For recursive structures like decorators, the copy() method must be implemented recursively.
- Example for LoggingDecorator:

```
java  
  
public LoggingDecorator copy() {  
    return new LoggingDecorator(aSource.copy());  
}
```

9. Copying State in Decorators (MemorizingDecorator):

- In a decorator like MemorizingDecorator, both the decorated element and the additional state (aDrawnCards) must be copied:

```
java  
  
public MemorizingDecorator copy() {  
    MemorizingDecorator copy = new MemorizingDecorator(aElement.copy());  
    copy.aDrawnCards = new ArrayList<>(aDrawnCards);  
    return copy;  
}
```

10. Copying Composite Structures (CompositeCardSource):

- For composite structures like CompositeCardSource, the copy() method involves copying each element in the composite:

```

java

public CardSource copy() {
    CompositeCardSource copy = new CompositeCardSource();
    copy.aElements = new ArrayList<>();
    for (CardSource source : aElements) {
        copy.aElements.add(source.copy());
    }
    return copy;
}

```

11. Polymorphic Copying of All Subtypes:

- The copy() method supports polymorphic copying of all CardSource subtypes, ensuring the correct behavior for complex object graphs.

6.7

Here is a summarized version of the text in enumerated points:

1. Polymorphic Copying:

- Polymorphic copying allows for versatile object creation and can be used in composition-based designs for various purposes.

2. Polymorphic Instantiation Use Case:

- A specialized use of polymorphic copying is polymorphic instantiation, such as creating a fresh CardSource for each new game in a card game scenario.

3. Challenges with Hard-Coding Types:

- Hard-coding the specific type of card source (e.g., new Deck()) works, but limits flexibility, especially if we want to change the type at runtime.

4. Issues with SWITCH Statement:

- Using a SWITCH statement to handle different types breaks polymorphism, reduces extensibility, and complicates the design.

5. Metaprogramming Solution:

- Metaprogramming, such as passing a Class<T> parameter to specify the type of card source, can be used but tends to be fragile and error-prone.

6. Polymorphic Copying as a Solution:

- Instead of hard-coding or using metaprogramming, polymorphic copying can be employed to create new instances by copying a prototype object.

7. PROTOTYPE Design Pattern:

- The PROTOTYPE pattern is useful when the type of object to be created is unknown at compile time.
- It involves storing a reference to a prototype object and using polymorphic copying to generate new instances.

8. GameModel Example with PROTOTYPE:

- In the GameModel scenario, a CardSource prototype is injected into the class.
- The newGame() method creates a new CardSource by copying the prototype, avoiding hard-coding or branching.

9. Dependency Injection:

- The prototype is injected into the GameModel via the constructor, and new CardSource objects are created by copying the prototype.
- A setter method can be added to change the prototype at runtime if needed.

10. Advantages of the PROTOTYPE Pattern:

- The PROTOTYPE pattern avoids adding control flow logic (like branching statements) in the client class.
- It uses polymorphism to create objects based on the current prototype, rather than checking the state or using conditionals.

11. Class Diagram for PROTOTYPE:

- A class diagram summarizes the solution, showing the client, the prototype (interface), and the products (objects created by copying the prototype).

6.8

Here is a summarized version of the text in enumerated points:

1. Command Concept:

- A command represents a cohesive action (e.g., saving a file, drawing a card) and is typically implemented as a method or function in code.

2. Context for Command Usage:

- In sophisticated applications, commands may need to be managed in various ways, such as storing a history of commands for undo/redo, batch execution, or integration with a user interface.

3. COMMAND Design Pattern:

- The COMMAND pattern provides a way to manage commands as objects, with an interface defining an execute() method and possibly additional methods like undo().

4. Simple Solution Template:

- The pattern involves defining command objects with an execute() method, and the client refers to these commands through the command interface.

5. Challenges in Applying the Pattern:

- Several implementation-dependent design choices must be made, including:
 - **Access to Command Target:** How the command accesses the object it modifies (e.g., by storing a reference within the command object or passing it to the execute() method).
 - **Data Flow:** Commands that produce results (like drawing a card) must handle returning the result, as the default void return type does not support this.
 - **Command Execution Correctness:** Ensure that commands execute in the correct order and handle issues like re-executing commands or respecting preconditions.
 - **Encapsulation of Target Objects:** Some operations might not be available through the target object's public interface, requiring additional solutions (e.g., creating a createDrawCommand() method in the Deck class).
 - **Storing Data:** For operations like undoing a command, the state affected by the command (e.g., the drawn card) must be stored for later restoration.

6. Example Design - Draw Command:

- A command to draw a card from the deck is implemented using a factory method in the Deck class. This creates an anonymous class representing the command:
 - `execute()` draws a card and stores it in a field.
 - `undo()` pushes the card back onto the deck.
- The command is executed and undone using the `execute()` and `undo()` methods.

7. Code Example:

- The Deck class creates a `createDrawCommand()` method, which returns a command that draws a card and stores it for potential undoing:

```
java

public Command createDrawCommand() {
    return new Command() {
        Card aDrawn = null;
        public Optional<Card> execute() {
            aDrawn = draw();
            return Optional.of(aDrawn);
        }
        public void undo() {
            aCards.push(aDrawn);
            aDrawn = null;
        }
    };
}
```

8. Command Execution and Undo:

- The command can be executed and undone in the following way:

```

java

Deck deck = new Deck();
Command command = deck.createDrawCommand();
Card card = command.execute().get();
command.undo();

```

- The command object retains a reference to the Deck instance, allowing the execute() and undo() methods to interact with the deck's state.

9. Benefits of Using Command Objects:

- Using command objects provides flexibility in how and when commands are executed, enabling advanced features like undo/redo, history tracking, and batch processing.

6.9

Here is a summarized version of the text in enumerated points:

1. Aggregation and Delegation Chains:

- Software designs using aggregation often lead to long delegation chains between objects, where each object delegates functionality to another.
- Example: A GameModel object manages four piles of cards through the Foundations class, which in turn holds references to four CardStack instances.

2. Delegation for Adding Cards:

- A design may require the GameModel to manage the entire process of adding a card to a pile by navigating through the delegation chain.
- Example: The GameModel calls aFoundations.getPile(FIRST).getCards().add(pCard) to add a card to a pile.

3. Violation of Information Hiding:

- This design violates the principle of information hiding by requiring GameModel to understand and navigate the delegation chain (e.g., accessing getPile() and getCards()).
- The design exposes too much of the internal structure of intermediate classes like Foundations and CardStack.

4. **MESSAGE CHAIN Antipattern:**

- This design pattern can lead to the **MESSAGE CHAIN** antipattern, where methods rely on long chains of method calls to access data or functionality, breaking encapsulation.

5. **Law of Demeter:**

- The **Law of Demeter** is a design guideline that recommends methods should only access:
 - Instance variables of the implicit parameter.
 - Arguments passed to the method.
 - Any new objects created within the method.
 - (If needed) globally available objects.
- This helps avoid the problems associated with message chains.

6. **Improving the Design:**

- To adhere to the Law of Demeter, intermediate classes in the delegation chain should provide the full service needed by the client, rather than exposing internal objects.
- Example: Instead of returning internal structures like a `List<Card>`, Foundations would directly provide the service needed to add a card to the pile.

7. **Solution Illustration:**

- The improved design would prevent objects from returning references to their internal structures, and instead, offer the necessary functionality directly to the client at each step in the delegation chain, improving encapsulation.

Chapter 7

7.1

Here is a summarized version of the text in enumerated points:

1. **Polymorphism in Design:**

- Polymorphism helps make a design extensible by decoupling client code from the concrete implementation of functionality.
- Example: A GameModel depends on a general CardSource interface, which can have different concrete implementations like Deck, MemorizingDeck, and CircularDeck.

2. **Extensibility via Polymorphism:**

- The design is extensible because GameModel can work with any card source that implements the CardSource interface.
- Polymorphism in Java relies on subtyping, where different implementations of CardSource are subtypes of the interface.

3. **Weakness in Implementation:**

- Despite the clean design in terms of polymorphism, the concrete implementations of the CardSource interface share similar code and structure.
- All three CardSource implementations (e.g., Deck, MemorizingDeck, and CircularDeck) hold references to a CardStack and share similar methods like isEmpty() and draw(), leading to duplicated code.

4. **Issue of Duplicated Code:**

- The shared implementation across the CardSource classes leads to **duplicated code** (also known as code clones), which should be avoided in software design.

5. **Improving Redundancies:**

- To address code duplication, object-oriented programming languages support **inheritance**, a mechanism that facilitates code reuse and avoids redundant code.

6. **Inheritance for Code Reuse:**

- Inheritance allows defining new classes (subclasses) based on existing ones (superclasses), thereby reusing code and extending functionality.
- By using inheritance, classes do not need to repeat declarations of class members, as they inherit from the base class.

7. Revised Design with Inheritance:

- A revised design uses inheritance, where MemorizingDeck and CircularDeck are subclasses of a common Deck base class, thus avoiding duplicated code.

7.2

Here is a summarized version of the text in enumerated points:

1. Declaring Inheritance:

- In Java, the extends keyword is used to declare a subclass, e.g., public class MemorizingDeck extends Deck.

2. Effect of Inheritance:

- A subclass inherits from its superclass, and objects of the subclass are created using both the subclass and superclass declarations.
- The run-time type of an object remains the type specified during instantiation, even when assigned to a variable of a different type (superclass or interface).

3. Compile-Time vs. Run-Time Types:

- Compile-time type refers to the variable's declared type, while run-time type refers to the actual type of the object at execution.
- An object's run-time type remains unchanged, while its compile-time type can change depending on the variable it is assigned to.

4. Example of Compile-Time and Run-Time Types:

- Example: Deck deck = new MemorizingDeck(); assigns an object of MemorizingDeck (run-time type) to a Deck variable (compile-time type).
- The isMemorizing() method can check the run-time type using instanceof.

5. Downcasting:

- Downcasting is necessary when converting a superclass reference to a subclass reference. It is required when a variable of a superclass type is assigned an object of a subclass type, but the subclass-specific methods cannot be accessed directly.
- Example: `MemorizingDeck memorizingDeck = (MemorizingDeck) deck;` performs a downcast to access subclass-specific methods like `getDrawnCards()`.

6. **Risks of Downcasting:**

- Downcasting is unsafe because it assumes the object is of the subclass type. If the assumption is incorrect, a `ClassCastException` will occur.
- To ensure safety, downcasting is often checked with `instanceof` before casting: `if(deck instanceof MemorizingDeck) { ... }.`

7. **Singly-Rooted Class Hierarchy:**

- Java supports single inheritance, meaning a class can only inherit from one class, but classes can have multiple ancestors through the inheritance chain.
- Every class in Java is part of a single-rooted class hierarchy, with `Object` as the root class. Therefore, every class inherits from `Object`.

8. **Transitive Subtyping:**

- Due to transitive subtyping, objects of a subclass (e.g., `MemorizingDeck`) can be stored in variables of the superclass type (e.g., `Deck`), or even `Object`, the root class.

7.3

Here is a summarized version of the text in enumerated points:

1. **Field Inheritance:**

- Inheritance includes field declarations, meaning that a subclass inherits all fields from its superclass.
- For example, an object of `MemorizingDeck` will have both `aCards` (inherited from `Deck`) and `aDrawnCards` (declared in `MemorizingDeck`).

2. **Field Accessibility:**

- Field accessibility is static, meaning it depends on the source code visibility (e.g., private, protected).
- Fields can be made protected to allow access by subclasses, or accessed via getter methods to maintain encapsulation.
- To respect encapsulation, it's recommended to keep fields private unless there is a clear need to make them protected.

3. Field Initialization:

- Fields are initialized from the superclass to the most specific class (top-down).
- For example, in MemorizingDeck, aCards (from Deck) is initialized before aDrawnCards.

4. Constructor Call Order:

- The first statement in any constructor calls the superclass constructor (super()), which initializes fields inherited from the superclass.
- Constructor calls are chained from the superclass down to the subclass, ensuring that field initialization happens in the correct order.

5. Handling Input Data in Constructors:

- If initialization requires input data (e.g., cards), constructors can pass data to the superclass constructor using super(pCards).
- This ensures that the superclass fields (e.g., aCards in Deck) are properly initialized with the input values.

6. Example of Constructor Initialization:

- The Deck class provides a constructor that can initialize aCards with input data (Card[] pCards).
- The MemorizingDeck class calls super(pCards) to pass input data up to Deck's constructor, allowing proper initialization of aCards.

7. Incorrect Constructor Behavior:

- If the superclass constructor is called via a new statement within a subclass constructor (e.g., new Deck(pCards)), a new, separate instance of Deck would be created, leading to issues.

- The correct approach is to use `super(pCards)` to initialize the superclass within the same object instance.

7.4

Here is a summarized version of the key points in the text:

1. **Difference Between Inheriting Methods and Fields:**

- Methods do not represent the state of an object, unlike fields, so they do not require initialization. The focus of method inheritance is on applicability, not state management.

2. **Applicability of Inherited Methods:**

- By default, methods in a superclass are applicable to instances of its subclass. For example, a `shuffle()` method in `Deck` can be called on a `MemorizingDeck` instance.

3. **Method Invocation (this vs Explicit Parameter):**

- In instance methods, `this` refers to the target object (implicitly), while an explicit parameter can be used to call the method (e.g., `Deck.shuffle(memorizingDeck)`).

4. **Overriding Methods for Custom Behavior:**

- If an inherited method doesn't meet the subclass's requirements, like `draw()` in `Deck`, it can be overridden in the subclass (e.g., `MemorizingDeck`) to provide the desired behavior.

5. **Private Fields and Access in Subclasses:**

- Private fields (e.g., `aCards` in `Deck`) are not accessible in subclasses. To work around this, fields can be made protected, though this may weaken encapsulation.

6. **Dynamic Dispatch for Method Selection:**

- The Java Virtual Machine selects the most specific overridden method based on the runtime type of the object, not the declared type. This process is known as dynamic dispatch or dynamic binding.

7. **Bypassing Dynamic Dispatch Using `super`:**

- To specifically invoke a superclass method (e.g., `Deck.draw()`), the `super` keyword is used to statically bind the method call, bypassing dynamic dispatch.

8. Avoiding Recursion with `super`:

- When overriding methods, it's crucial not to cause infinite recursion. Using `super.draw()` ensures the correct method from the superclass is called without recursion.

9. Method Signature Matching and the `@Override` Annotation:

- To override a method correctly, the signature must match. Mistakes in the method signature (e.g., `hashCode()` instead of `hashCode()`) can lead to bugs. The `@Override` annotation helps avoid such errors by prompting the compiler to check if the method truly overrides another.

7.5

Here is a summary of the key points in the text:

1. Overloading vs. Overriding:

- Overriding allows different versions of a method based on the run-time type of the implicit parameter.
- Overloading allows different versions of a method based on the types of explicit parameters.

2. Example of Overloading:

- A common example of overloading is found in libraries like `java.lang.Math`, which provides multiple versions of methods like `abs()` for different primitive types (e.g., `int` and `double`).

3. Constructor Overloading:

- Constructors can also be overloaded, as in the example of `MemorizingDeck`, which has three versions of the constructor:
 - Version 1: No arguments.
 - Version 2: Takes a `CardSource` argument.
 - Version 3: Takes a `MemorizingDeck` argument.

4. Selection of Overloaded Methods/Constructors:

- The specific method or constructor is selected based on the number and static types of the explicit arguments.
- The most specific applicable version is chosen.

5. **Example of Constructor Overloading in Action:**

- Three different constructor calls for `MemorizingDeck`:
 - The parameterless constructor (Version 1) is selected when no argument is passed.
 - For `newDeck1`, the constructor that takes a `MemorizingDeck` (Version 3) is selected.
 - For `newDeck2`, the constructor that takes a `Deck` (Version 2) is selected because `Deck` is not a subtype of `MemorizingDeck`.

6. **Confusion in Overloading with Type Hierarchies:**

- Overloading can become confusing when the types of parameters in the overloaded versions are related within a type hierarchy.
- The selection of an overloaded method does not consider the types of variables, only the static types of the arguments.

7. **Recommendation on Overloading:**

- Overloading is useful for organizing related alternatives (e.g., constructor overloading or library methods for different primitive types).
- Overloading should be avoided in cases where it leads to hard-to-understand code. Instead, using different method names can often provide the same functionality.

8. **Best Practices:**

- Overload methods sparingly and avoid complex overloads with related types in a hierarchy, as they can make code harder to understand.

7.6

Here is a summarized version of the key points in the text:

1. **Polymorphic Copying and Inheritance:**

- To implement polymorphic copying, it's crucial to have detailed information about an object's state to make an exact copy.
- Overriding the `copy()` method is necessary in the presence of inheritance to avoid issues with faulty copying behavior.

2. Problem with Inherited `copy()` Method:

- Inheritance of the `copy()` method can lead to errors. For example, calling `deck.copy()` on a `MemorizingDeck` object returns a `Deck` type, which violates object equality constraints (objects must be of the same type).

3. Need to Override `copy()` in Subclasses:

- To support polymorphic copying, the `copy()` method should be overridden in all leaf classes of the hierarchy (e.g., `MemorizingDeck` must override `copy()`).

4. Visibility Issues with Inherited Fields:

- A subclass may not access private fields of the superclass, leading to compilation errors when attempting to copy fields like `aCards`.
- Changing field visibility to `protected` may not be a viable solution, as it weakens encapsulation and may not be possible if inheriting from an `unmodifiable` class.

5. Java Cloning Mechanism:

- Java provides cloning via the `clone()` method in class `Object` to support polymorphic copying.
- Cloning involves overriding `Object#clone()` and making a super call to `super.clone()`.

6. Shallow Copying via `Object#clone()`:

- The `clone()` method performs a shallow copy of the object, which can be insufficient if the object contains mutable objects (e.g., `CardStack` in the `Deck` class).
- A deep copy is necessary for fields like `aCards`, which can be achieved by calling a copy constructor.

7. Problems with `clone()` in Inheritance:

- Using the inherited clone() method in Deck on a MemorizingDeck object results in a shallow copy of fields like aDrawnCards, leading to potential issues.

8. Constraints and Risks of Using clone():

- To use the cloning mechanism properly, classes must implement the Cloneable interface and handle exceptions.
- The Java cloning mechanism is complex and error-prone, requiring careful design and understanding to avoid mistakes.

9. Recommendation:

- It's recommended to study the technical documentation of cloning carefully before using it in designs due to its complexity and risk of misuse.

7.7

Here is a summarized version of the key points in the text:

1. Composition vs. Inheritance:

- **Composition:** A class like MemorizingDeck can implement CardSource and contain an instance of Deck, delegating method calls to the Deck object.
- **Inheritance:** A class like MemorizingDeck can extend Deck, inheriting fields and methods from Deck and overriding specific methods to add functionality.

2. Composition-based Implementation:

- MemorizingDeck aggregates a Deck and delegates method calls (e.g., shuffle(), draw()) to the Deck object.
- Example code shows how methods like isEmpty(), shuffle(), and draw() are implemented by delegating calls to the Deck object.

3. Inheritance-based Implementation:

- MemorizingDeck inherits from Deck, so it doesn't need to redefine all methods (e.g., isEmpty() is inherited directly).
- Only shuffle() and draw() are overridden to account for the memorization functionality.

4. Problem in Inheritance-based Implementation:

- A `NullPointerException` occurs in the `shuffle()` method due to the order of field initialization in the `MemorizingDeck` constructor.
- The `Deck` constructor is called first, and when it calls `shuffle()`, the `aDrawnCards` field in `MemorizingDeck` has not been initialized yet.

5. **Difference in Object Identity:**

- **Composition:** Involves two objects: a `Deck` object and a `MemorizingDeck` object (decorator pattern), meaning they have different identities.
- **Inheritance:** Creates a single `MemorizingDeck` object, which combines all fields and functionality.

6. **When to Choose Composition:**

- Composition offers greater run-time flexibility, allowing more configurations and runtime changes.
- It is better for designs that require flexibility or the ability to change configurations dynamically.

7. **When to Choose Inheritance:**

- Inheritance is better for designs that require compile-time configuration and access to internal state through class hierarchies.
- It supports finer-grained polymorphism, such as storing a `MemorizingDeck` in a `Deck` reference, which isn't possible with composition.

7.8

Here is the summary in enumerated points:

1. **Problem of Common Class Members:**

- Sometimes, grouping common members in a superclass makes the class non-instantiable.
- Example: The `Move` interface represents game actions (e.g., `perform()`, `undo()`), and the classes implementing it aggregate a `GameModel` object.

2. **Inheritance vs. Composition for Command Objects:**

- Inheritance could pull the `GameModel` field into a superclass, but a base class for commands (e.g., `CardMove`) may not be logical as commands are different from one another.

- A DiscardMove and CardMove are different moves, and inheriting from CardMove would lead to undesired behavior and bugs, such as inheriting unneeded methods like getDestination().

3. **Need for a New Base Class:**

- A new base class, like AbstractMove, is needed for command objects to avoid inappropriate inheritance hierarchies.
- This base class should not define behavior for perform() and undo(), but it could contain common functionality, like a reference to GameModel.

4. **Abstract Class Concept:**

- The AbstractMove class is made abstract, meaning it can't be instantiated and may define common methods or properties.
- It is declared as abstract in Java, and the common GameModel field is initialized in its constructor.

5. **Key Characteristics of Abstract Classes:**

- **Cannot Be Instantiated:** Abstract classes cannot be directly instantiated by the compiler.
- **Partial Implementation:** Abstract classes may not implement all methods from an interface they declare, leaving that for concrete subclasses.
- **Abstract Methods:** Abstract classes can have abstract methods that subclasses must implement.

6. **Constructor Handling in Abstract Classes:**

- The constructor of abstract classes can only be called from within constructors of subclasses.
- Typically, the constructor of an abstract class is declared protected, so it can be accessed by subclasses.
- In the example, AbstractMove's constructor is called by subclasses like CardMove to initialize the GameModel reference.

7. **Abstract Class Benefits:**

- Ensures that only subclasses with complete implementations are instantiated, while preventing direct instantiation of abstract classes.

- Provides a flexible design to represent abstract concepts (like AbstractMove for commands).

7.9

Here is the summary in enumerated points:

1. **Decorator Pattern Review:**

- In Section 6.4, the **Decorator pattern** was used to add features to objects at runtime via wrapper classes and composition, rather than inheritance.

2. **Multiple Decorators:**

- When using multiple decorator types, each decorator must aggregate an object to be decorated, creating redundancy that inheritance typically avoids.

3. **Combining Composition and Inheritance:**

- To avoid redundancy, inheritance can be used to pull up the aElement field into an abstract base class, AbstractDecorator.
- Concrete decorator subclasses only need to focus on the specific decoration logic.

4. **AbstractDecorator Class:**

- AbstractDecorator is an abstract class that implements CardSource and aggregates a CardSource object (the element to be decorated).
- It provides default delegation to the decorated CardSource through methods like draw() and isEmpty().

5. **Encapsulation in Decorators:**

- The aElement field is private in AbstractDecorator, meaning concrete decorators cannot access it directly.
- This encapsulation is acceptable since the decorated element is typically accessed through the interface methods inherited from AbstractDecorator.

6. **Example: LoggingDecorator:**

- LoggingDecorator is a concrete decorator that extends AbstractDecorator.

- It overrides the draw() method: it calls the inherited draw(), logs the card, and returns the card.
- It does not need to override isEmpty() because it inherits the correct behavior from AbstractDecorator.

7. Delegate Pattern in Action:

- In this design, each decorator class extends AbstractDecorator, overriding only the necessary methods to add functionality, while delegating common behavior to the base class.

7.10

Here is the summary in enumerated points:

1. Inheritance with Varying Algorithms:

- A common inheritance scenario arises when a base algorithm applies to all subclasses, but certain steps of the algorithm vary across subclasses.
- In the context of the Solitaire application, the perform() method needs to execute a common set of actions with subclass-specific variations.

2. Common Algorithm:

- The perform() method performs three actions:
 1. Adds the move to the undo stack in the GameModel.
 2. Executes the actual move (which varies by subclass).
 3. Logs the move (e.g., printing the move's description).
- This common behavior benefits from being centralized in the superclass to avoid code duplication and prevent errors.

3. Template Method Pattern:

- The **Template Method** pattern is used to solve the issue by providing a common algorithm in the superclass while allowing subclasses to implement specific variations.
- The superclass (AbstractMove) defines the perform() method as a final method and provides common steps, while subclasses define specific behavior through abstract methods.

4. **AbstractMove Implementation:**

- AbstractMove includes a final perform() method that calls common steps:
 - Pushes the move to the stack.
 - Calls an abstract execute() method for the specific move action.
 - Logs the move.
- The execute() method is abstract and must be implemented by subclasses to define move-specific logic.

5. **Final Methods and Classes:**

- In Java, final methods cannot be overridden by subclasses, ensuring that the common algorithm in perform() cannot be changed.
- Declaring the method perform() as final guarantees that steps like pushing to the stack and logging will always occur.
- Classes or methods declared final also prevent unintended inheritance, making the design more robust.

6. **Abstract Methods:**

- The execute() method is abstract because the actual move varies across subclasses.
- It is declared protected to ensure that only subclasses can access and implement it.

7. **Template Method Design Principles:**

- The common algorithm in the superclass is the **template method**, calling both concrete and abstract steps.
- Declaring the template method as final ensures the structure of the algorithm cannot be modified by subclasses.
- The abstract step method (execute()) should have a different signature from the template method to prevent recursive calls that could cause stack overflow.
- Abstract step methods should typically have protected access to restrict their use to subclasses.

8. Default Behavior in Subclasses:

- Not all step methods need to be abstract; some may have default behavior that can be overridden by subclasses (e.g., `log()` in `AbstractMove`).
- If a method has reasonable default behavior, it does not need to be abstract, and the superclass might not need to be abstract.

9. Client-Side Interaction:

- Clients interact with the `perform()` method, but the concrete steps (like `execute()`) are handled within the class hierarchy.
- The sequence diagram illustrates how the call to `perform()` ultimately leads to the execution of the abstract method in the subclass.

10. Inheritance Best Practices:

- The **Template Method** pattern provides a clean way to handle common behavior in the superclass while allowing subclasses to implement specific functionality.
- This design helps avoid code duplication and minimizes the risk of errors caused by inconsistent subclass implementations.

7.11

Here is a summarized version in enumerated points:

1. Inheritance as Code Reuse and Extensibility:

- Inheritance allows a subclass to reuse the code of its superclass and also act as a subtype of the superclass.
- Inheritance should only be used to extend the behavior of a superclass, not to restrict or misuse it.

2. Problems with Restricting Behavior in Subclasses:

- Using inheritance to limit behavior (e.g., disabling features in a subclass like shuffling in `UnshufflableDeck`) can create problems.
- This restricts polymorphism, where operations should work independently of the object's concrete type.

3. Liskov Substitution Principle (LSP):

- Subclasses should not restrict the behavior available in the superclass.
- Violations of LSP include:
 - Having stricter preconditions.
 - Having less strict postconditions.
 - Using more specific types for parameters or return types.
 - Making methods less accessible (e.g., public to protected).
 - Throwing more checked exceptions.

4. Example of LSP Violation: UnshufflableDeck:

- If shuffle() is overridden to do nothing or throw an exception, it disrupts the expected behavior when a Deck is passed as an argument to a method like shuffleAndDraw().
- This could lead to incorrect behavior or runtime exceptions.

5. Inappropriate Precondition in Subclass:

- In the DrawBestDeck, introducing stricter preconditions (e.g., requiring at least two cards to draw) violates LSP because it makes code unsafe for cases where only one card is available.
- The precondition check now has to be handled by clients, increasing complexity.

6. Inheritance Violating Method Signatures:

- Methods should not override base class methods to take more specific parameters or return less specific types, as this can lead to confusing behavior.
- For example, overriding init() in a MemorizingDeck class could lead to different behavior depending on the instance type.

7. Example of Violating Return Types:

- If a method in a subclass changes the return type to a more general type, it can break polymorphism, leading to assignment errors when clients expect specific return types.

8. Circle–Ellipse Problem:

- A classic LSP violation is the "Circle-Ellipse" problem, where a Circle class inherits from an Ellipse class but restricts clients from using ellipses with unequal dimensions, breaking the Liskov Substitution Principle.

9. Avoiding the Circle-Ellipse Problem:

- One solution is not to use inheritance between Circle and Ellipse. Instead, treat them as sibling classes in the type hierarchy or use composition.

10. Subclasses Not Being Proper Subtypes:

- Inheritance should only be used when a subclass truly represents a subtype of the superclass (i.e., "is-a" relationship).
- Inappropriate uses of inheritance occur when subtyping doesn't make sense, as seen in examples like Stack inheriting from Vector or Properties inheriting from Hashtable.

11. Composition Over Inheritance:

- When inheritance is not appropriate, composition should be favored to achieve code reuse without violating design principles like LSP.

Chapter 8

8.1

Here is a summarized version in enumerated points:

1. **Motivation for Inversion of Control (IoC):**

- IoC is needed when multiple stateful objects must remain consistent with each other.
- Example: Integrated development environments (IDEs) like Eclipse or IntelliJ IDEA, where changes in one view (e.g., source code editor) must immediately reflect in other views (e.g., Package Explorer, Outline view).

2. **Example: LuckyNumber Application:**

- In the LuckyNumber app, users can select a number (1-10) in different ways (e.g., digit, name, or slider).
- Any change made in one panel (e.g., typing or sliding) should be reflected in all other panels instantly.

3. **Challenge of View Synchronization:**

- The goal is to keep different views synchronized so changes in one panel automatically update all others.
- The app needs to be extendable to accommodate additional views (e.g., Roman numerals, binary notation).

4. **Problem of Pairwise Dependencies:**

- If the synchronization is implemented naively, it results in **pairwise dependencies** where each panel directly updates every other panel.
- This creates a situation where every change triggers updates across multiple panels.

5. **Limitations of Pairwise Dependencies:**

- **High Coupling:** Panels are tightly coupled, with each panel depending on many others. For instance, one panel may need `setDigit()` while another requires `setSliderValue()`.

- **Low Extensibility:** Adding or removing a panel requires modifying all other panels. For example, removing or adding a new panel (like a Roman numeral panel) requires changes to all other panels.

6. **Exponential Growth of Dependencies:**

- As the number of panels increases, the number of dependencies grows quadratically. For example:
 - With 3 panels: 6 dependencies.
 - With 5 panels: 20 dependencies.
- This leads to a significant increase in complexity.

7. **Poor Separation of Concerns:**

- The design results in tangled code that mixes dependency management with logic (e.g., updating a slider).
- This makes the code harder to understand, test, and maintain.

8. **Need for Inversion of Control:**

- The issue of tight coupling and low extensibility can be mitigated by applying Inversion of Control (IoC), which decouples the components and makes the system more flexible and maintainable.

8.2

Here is a summarized version of the text in enumerated points:

1. **Avoiding Pairwise Dependencies:**

- To avoid tight pairwise dependencies when synchronizing multiple representations of the same data, separate abstractions for storing, viewing, and changing the data.

2. **Model–View–Controller (MVC):**

- **Model:** Holds the unique data (e.g., the lucky number in the LuckyNumber app).
- **View:** Represents one view of the data (e.g., different ways of displaying the number).

- **Controller:** Manages the functionality to change the data stored in the model.

3. **Origin of MVC:**

- The MVC concept traces back to the late 1970s with researchers at Xerox PARC working on Smalltalk software.
- The term MVC is used loosely and can refer to a design pattern, architectural pattern, or just a general concept.

4. **MVC as a Guideline:**

- MVC should be seen as a guideline to separate concerns in software design, not as a strict solution template.
- This makes MVC more general than a design pattern, as it does not prescribe a specific solution.

5. **Flexibility in MVC Implementation:**

- There are many ways to implement MVC, depending on the context.
 - The **model** could be a single object or a collection of objects.
 - The **view** and **controller** can be separate objects or combined, with separation occurring at the interface level rather than object level (e.g., interface segregation).

6. **Challenges of Grasping MVC:**

- MVC can be difficult to understand when first learning software design because it lacks a concrete solution template.
- However, a related and more concrete design concept is the **Observer pattern**, which can aid in understanding the implementation of MVC.

8.3

Here's a summary of the Observer pattern in enumerated points:

1. **Observer Pattern Overview:**

- The Observer pattern allows objects to observe and react to changes in a subject (model).

- The object storing data is called the **model**, **subject**, or **observable**, and other objects observe the state of this data.

2. **Model-Observer Relationship:**

- The **Model** contains the data and notifies observers when the data changes.
- Observers register themselves to the model to receive updates about the data changes.

3. **Observer Interface:**

- Observers implement the Observer interface, which defines callback methods to respond to state changes.
- Example: `public interface Observer { void newNumber(int pNumber); }`

4. **Registering and Deregistering Observers:**

- Observers are added and removed using methods like `addObserver()` and `removeObserver()` in the **Model**.
- This creates loose coupling between the model and the observers.

5. **Notifying Observers:**

- When the model's state changes, it calls the `newNumber()` callback method on each observer.
- The **Hollywood Principle** ("Don't call us, we'll call you") is used here, where the model calls observers, not vice versa.

6. **Data Flow Strategies:**

- **Push Strategy:** The model provides data to observers through parameters in the callback method (e.g., `newNumber(int pNumber)`).
- **Pull Strategy:** Observers query the model directly for the data they need (e.g., `newNumber(Model pModel)`).

7. **Callback Method Design:**

- Callbacks inform observers of a state change and let them handle it.
- It's important to name callback methods to describe the state-change event (e.g., `newNumberAvailable`, `numberChanged`).

8. **Notification Methods:**

- A helper method in the **Model** (e.g., `notifyObservers()`) ensures observers are notified when the model's state changes.
- The notification method can be called after every state change or based on specific conditions (e.g., after batch changes).

9. **Observer Variability:**

- Observers can define multiple callback methods to handle different types of events (e.g., increased, decreased).
- Empty callback methods can be used when observers are not interested in certain events.

10. **Event Handling:**

- Callbacks can be designed to capture specific events of interest (e.g., increased, decreased, changedToMax, changedToMin).
- Observers can extend an **adapter class** to avoid implementing empty methods.

11. **Interface Segregation Principle (ISP):**

- To reduce unnecessary coupling, observers can implement specific interfaces for different event types (e.g., `ChangeObserver`, `BoundsReachedObserver`).

12. **Flexibility and Coupling:**

- Using the pull data-flow strategy increases the coupling between the model and observers, as observers can directly access the model.
- To minimize unnecessary coupling, interface segregation is recommended, where observers only need getter methods.

13. **Pattern Variations:**

- The **Observer pattern** can be adapted by:
 - Choosing between push or pull strategies for data flow.
 - Using a single observer interface or multiple, specialized ones.
 - Deciding if notifications should be public or private, depending on whether the client controls when notifications are issued.

14. **Summary of the Observer Pattern:**

- The **Observer pattern** allows multiple objects to observe changes in a model's state while minimizing coupling.
- It involves:
 - Defining callback methods on the observer interface.
 - Choosing between push, pull, or no data flow strategies.
 - Managing observer registration and notification.
 - Deciding on the flexibility of event handling and observer interfaces.

15. Code Example (Push Strategy):

- The model notifies observers using a list, calling `newNumber()` on each observer:

```
public class Model {
    private List<Observer> aObservers = new ArrayList<>();
    private int aNumber = 5;
    public void addObserver(Observer pObserver) {
        aObservers.add(pObserver);
    }
    public void removeObserver(Observer pObserver) {
        aObservers.remove(pObserver);
    }
    private void notifyObservers() {
        for (Observer observer : aObservers) {
            observer.newNumber(aNumber);
        }
    }
    public void setNumber(int pNumber) {
        aNumber = pNumber; // Set new value within valid range
        notifyObservers();
    }
}
```

8.4

Detailed Summary of Observer Design Variants:

1. Design Space of the OBSERVER Pattern:

- The **Observer** design pattern is highly flexible and adaptable, allowing for a variety of design alternatives for the observer and observable types.
- In this case, the **CardStack** class, which models a stack of cards, is used to illustrate the different design approaches. This class manages operations like **push()**, **pop()**, and **peek()**.
- The design must accommodate the needs of various observers. For example, two observers are considered:
 - **Counter**: Tracks the number of cards in the stack and detects when the last card is popped.
 - **AceDetector**: Detects when an Ace card is added to the stack.

2. Basic Design with Push Data-Flow:

- In this basic design, the **CardStack** class implements the observer pattern by introducing an abstract CardStackObserver interface with three callback methods:
 - pushed(Card pCard)
 - popped(Card pCard)
 - cleared()
- **Observable CardStack**:
 - The CardStack class manages a list of observers (aObservers) and notifies them when state-changing methods are called (e.g., push(), pop(), clear()).
 - When push(Card pCard) is called, for example, it updates the stack and then notifies all registered observers.
 - **Observer Code**:
 - The AceDetector only cares about the **push** event, so the popped() and cleared() methods are empty.
 - The Counter observer tracks the card count but has to replicate part of the **CardStack**'s state (the count of cards) to function correctly, leading to potential design issues:

- The **Counter** observer's state will only be accurate if attached to an empty CardStack, making the system brittle.

3. Design with Inheritance:

- To improve the basic design's flexibility and decouple the observer pattern from the core CardStack functionality, we can use **inheritance** to create an ObservableCardStack subclass.
- **ObservableCardStack** extends the base CardStack class and overrides the state-changing methods (e.g., pop(), push(), etc.) to include observer notifications after the state change.
 - This design allows the original CardStack to be used in cases where observer notification is not needed, and the ObservableCardStack to be used in contexts where observers are required.
- **Observer Adapter:**
 - To handle observers that don't need to implement all callbacks (such as AceDetector), an **adapter class** CardStackObserverAdapter provides empty implementations for all methods. Observers can then inherit from this adapter and only override the methods that are relevant to them (e.g., only pushed() for AceDetector).

4. Design with Pull Data-Flow:

- A **pull data-flow** strategy allows observers to **fetch** or **query** the state of the observable rather than relying on the observable to push updates.
- **CardStackView Interface:**
 - A new interface, CardStackView, is introduced, which provides methods that allow observers to query the state of the CardStack (e.g., peek(), isEmpty(), etc.) without coupling them to the stack's mutating methods like push() or pop().
 - Instead of passing a Card to the observer callbacks (e.g., in popped()), the observable passes a reference to itself (CardStackView), allowing the observer to query the current state directly.
- **Impact on Observers:**

- **AceDetector:** The observer now needs to query the observable (e.g., using `peek()`) to detect if the last card was an Ace, as it no longer has direct access to the card pushed to the stack.
- **Counter:** The observer no longer needs to replicate the stack size in a field, as the state is accessible via `CardStackView`. However, the `CardStackView` interface does not directly provide a `size()` method. This requires the observer to either:
 - **Option 1:** Modify the `CardStack/CardStackView` classes to add a `size()` method.
 - **Option 2:** Implement a helper method to calculate the size in the observer (`Counter`), which involves iterating over all the cards in the stack.

5. Design with Single Callback and Push/Pull Data-Flow:

- In this design, a **single callback method** (`actionPerformed`) is introduced to handle both **push** and **pull** data flows.
- The `actionPerformed` method takes:
 - An enumerated Action type (e.g., `PUSH`, `POP`, `CLEAR`) to distinguish event types.
 - An `Optional<Card>` to handle cases where no card is involved (e.g., for the `CLEAR` event).
 - A `CardStackView` to allow observers to query the state.
- **Observer Implementation:**
 - Observers like **AceDetector** only need to check the Action and perform actions like detecting an Ace from `peek()`.
 - **Counter:** With a single callback, the observer uses a switch statement to handle different types of events (`PUSH`, `POP`, `CLEAR`).
 - This leads to more complex observer logic, where observers must check the Action and perform corresponding operations for each event, resulting in a less maintainable design if many events are handled in a single method.

6. Trade-offs and Flexibility:

- **Flexibility and Generalization:** Each design approach has its own trade-offs between flexibility and complexity.
 - For instance, the **basic design with push data-flow** is simple but tightly couples observers to state changes and makes them brittle.
 - **Inheritance-based design** introduces more flexibility by separating concerns but requires subclassing.
 - **Pull data-flow** provides greater decoupling between observers and the observable but may require adding additional state-querying methods or inefficient iterations.
 - The **single callback with push/pull** strategy consolidates event handling into one method but introduces complexity and possible maintainability issues, especially if the number of event types grows.
- **Observer Design Context:** The decision of which approach to take depends on the requirements of the specific context:
 - In some cases, it may make sense to allow only one observer per event type, while in other cases, it may be more important to minimize coupling between the observer and observable, favoring **pull data-flow** or **single callback** designs.

8.5

Summary of GUI Development and the Observer Pattern:

1. GUI and Observer Pattern:

- GUI applications often rely heavily on the **Observer pattern** to manage user interactions and events.
- The section introduces GUI development concepts, focusing on JavaFX, but the ideas are applicable to other GUI frameworks as well.

2. Two Parts of a GUI Application:

- **Framework Code:** Comprises a component library and an application skeleton.
 - The component library includes reusable components like buttons, windows, etc.

- The application skeleton is responsible for managing low-level aspects like monitoring user inputs and displaying objects but does not perform visible tasks by itself.
- **Application Code:** Custom code written by developers to extend the application skeleton and define the specific functionality of the GUI.

3. Event-Driven Execution:

- Unlike script-based applications, GUI applications do not execute sequentially. The framework controls the flow of execution.
- The framework starts an event loop that continuously monitors for user input (e.g., clicks, key presses).
- The **Inversion of Control** principle is applied: the framework calls the application code in response to events rather than the application controlling the flow.

4. Launching the Framework:

- The GUI application is started by launching the framework via a special library method (e.g., `Application#launch` in JavaFX).
- This method instantiates the application class (e.g., `LuckyNumber`) and calls its `start()` method to begin the GUI application.

5. Application Code Structure:

- The application code is split into two parts:

1. **Component Graph:** Defines the user interface structure (buttons, text boxes, etc.).
2. **Event Handling Code:** Defines how the application should respond to user interactions (events).

6. Component Graph:

- The component graph represents the GUI and consists of objects that represent both visible and invisible elements.
- These objects are organized in a tree-like structure, with the root being the main window or GUI area.
- In JavaFX, the component graph is often instantiated within the `start(Stage pPrimaryStage)` method of the application class.

- **Design Patterns Used:** The construction of the component graph uses **polymorphism**, **COMPOSITE**, and **DECORATOR** patterns.

7. Event Handling in GUI Frameworks:

- GUI frameworks automatically detect system events (e.g., mouse clicks) and map them to interactions with components in the component graph.
- An **event** occurs when a user interacts with a GUI component (e.g., clicking a button).
- By default, the framework does nothing with events unless the application code provides specific event handlers (observers).

8. Observer Pattern for Event Handling:

- The **Observer pattern** is used to handle events like button clicks.
- The GUI component (e.g., a button) is the "subject" or "observable," and the event handler is the "observer."
- Developers register event listeners (observers) with GUI components to react to specific events (e.g., a button click).

9. Next Steps:

- The following sections will cover in more detail how to design component graphs and implement event handling in GUI applications.

8.6

Summary of the Component Graph in GUI Development:

1. Component Graph Overview:

- The **component graph** consists of objects that form the user interface, such as windows, textboxes, and buttons.
- It can be viewed from three perspectives: **User Experience**, **Source Code**, and **Run-time**.

2. User Experience Perspective:

- Represents what the user interacts with in the GUI.
- Not all objects in the component graph are visible; the user experience perspective only shows the visible components, not the complete graph.

3. Source Code Perspective:

- Focuses on the information available from the declarations of the component graph's objects, typically represented by a **class diagram**.
- Example: A class diagram of the **LuckyNumber** application shows the component graph's structure.
- **Polymorphism** is used, as the **Scene** class can accept any **Parent** subtype as a child object.
- The design applies the **COMPOSITE pattern**, as **Parent** is a subtype of **Node** that can hold child nodes (which can be any **Node**).
- The **GridPane** class organizes its children into a grid, and it can contain subtypes of **Node** such as **TextPanel**, **IntegerPanel**, and **SliderPanel**, representing different views in a Model-View-Controller (MVC) design.

4. Run-time Perspective:

- This represents the **instantiated component graph** at runtime, often shown as an object diagram.
- The actual objects created and linked at runtime form the user interface that is visible to the user.

5. Defining the Object Graph:

- The component graph is constructed in the **start()** method of the main application class (e.g., **LuckyNumber**).
- A **Model** instance is created, and a **GridPane** is used as the root to hold UI components like **SliderPanel**, **IntegerPanel**, and **TextPanel**.
- The **setScene()** method assigns the component graph to the **Stage**, and **show()** displays the GUI.

6. Code Example:

- In the **LuckyNumber** class, the **start()** method creates a **Model**, builds the component graph using **GridPane**, and displays it.
- Each component (like **SliderPanel**) is added to the **GridPane** with its specific layout properties.

7. IntegerPanel Design:

- **IntegerPanel** extends **Parent** and becomes part of the component graph.
- It aggregates a **TextField** but doesn't add it directly; it uses **getChildren().add(aText)** to add the **TextField** to the component graph.
- **IntegerPanel** also observes the **Model** (following the **Observer pattern**), updating the **TextField** whenever the model's value changes.

8. Design Insights:

- The **Model** is created within the **start()** method but not directly managed by the **LuckyNumber** class.
- Each UI component, like **IntegerPanel**, manages its reference to the model to avoid the **God Class** problem, ensuring a modular and maintainable design.

9. Component Hierarchy and Polymorphism:

- The design leverages inheritance and polymorphism for flexibility. For instance, **GridPane** can hold any **Node** subclass, and components like **SliderPanel** and **IntegerPanel** can be added to it seamlessly.

8.7

Summary of Event Handling and Observer Pattern in GUI Frameworks:

1. Objects in Component Graph as Observers:

- In GUI frameworks, objects in the component graph function as models in the **Observer pattern**.
- The framework continuously monitors input events and checks if they correspond to events observable by the application code.

2. Event Definition:

- Events are defined by the component library (e.g., **TextField** in JavaFX defines an action event triggered by pressing the ENTER key).
- In this context, **TextField** acts as the model in the Observer pattern.

3. Handling Action Events on TextField:

- To handle an action event, the process involves:
 1. **Define an event handler:** Create a class that extends **EventHandler**.

2. **Instantiate the handler:** Create an instance of the handler class.

3. **Register the handler:** Call the `setOnAction(handler)` method on the component (e.g., **TextField**).

- Note: A **TextField** can only have one observer for its action event.

4. **Strategies for Defining Event Handlers:**

- **Function Object Strategy:** Use an anonymous class or lambda expression for simple handlers that don't require storing complex data.
- **Delegate to a Component:** Declare the component (e.g., **IntegerPanel**) to implement the **Observer** and **EventHandler** interfaces for more complex handling logic.

5. **Example Using Function Object Strategy:**

- In the **LuckyNumber** application, **IntegerPanel** uses a function object (anonymous class) to handle the **TextField** action event.
- When the user presses ENTER, the handler parses the text input, sets the number in the model, and updates the UI.
- This results in two applications of the **Observer** pattern: one for the **Model** being observed by the panels, and another for the **TextField** being observed by the handler.

6. **Alternative: Delegate Event Handling to the Panel:**

- Alternatively, **IntegerPanel** could implement both **Observer** and **EventHandler**, making it directly responsible for handling events and reacting to changes in the model.
- The **handle** method is now declared in the **IntegerPanel** class, and **setOnAction(this)** is used to register the panel itself as the event handler.

7. **Code Implications:**

- In the delegation approach, the **handle** method is implemented within **IntegerPanel**.
- The **TextField** registers **IntegerPanel** directly as its event handler by calling **setOnAction(this)**.

8. **Preferred Approach for LuckyNumber:**

- The **function object strategy** is preferred for the **LuckyNumber** application because the handler code is simple and concise.
- Using function objects keeps the handler code close to the component initialization, making the code easier to read and maintain.

9. Recommendation:

- While both strategies are valid, using **function objects** for defining GUI event handlers is a common practice.
- For simplicity, unless there's a compelling reason to delegate handling to components, function objects are a good default choice.

8.8

1. **Inversion of Control (IoC)** can be used in design patterns beyond the **Observer pattern** and event handling mechanisms, such as the **Visitor pattern**, to decouple functionality from the objects it operates on.
2. The **Visitor pattern** is helpful when we want to add new operations to an object hierarchy without altering the objects' interfaces. It is particularly useful when the object graph can change dynamically.
3. In the example with **CardSource**, three different concrete card sources implement the **CardSource interface** but each has specific additional methods needed for their functionality, like **shuffle()** for **Deck** or **size()** for **CardSequence**.
4. Adding new methods directly to the **CardSource interface** can violate the **Interface Segregation Principle** by making the interface larger and potentially introducing unused methods, leading to **speculative generality**.
5. The **Visitor pattern** addresses this problem by allowing operations to be added in separate visitor classes that do not modify the original object classes. Each visitor implements one operation for each class in the hierarchy.
6. The **CardSourceVisitor** interface defines methods for each concrete class in the object hierarchy (e.g., **visitDeck()**, **visitCardSequence()**). Concrete visitors implement these methods to define behavior for specific operations.
7. The **PrintingVisitor** example shows how the visitor can print the cards in a **Deck** or **CardSequence** without modifying the card source classes, enabling functionality to be added in a single class.

8. **accept()** method integrates the **Visitor pattern** by enabling objects to call back the appropriate visit method on a visitor. The **accept()** method is usually defined in the common supertype (e.g., **CardSource**) of the hierarchy.
9. The **accept()** method in **Deck** or **CardSequence** calls the appropriate visit() method on the visitor. For **CompositeCardSource**, the accept method is more complex due to aggregation, and requires traversing the child elements.
10. The **traversal of the object graph** in the **Visitor pattern** can be done in two ways:
 - In the **accept()** method of aggregate classes like **CompositeCardSource**.
 - In the visitor's **visit()** method, which requires additional access to the internal structure of the aggregates.
11. Using **inheritance** in the **Visitor pattern** can reduce code duplication. An abstract visitor class can hold default traversal code (e.g., for **CompositeCardSource**) to avoid repeating the traversal logic in multiple concrete visitors.
12. **Data flow** in **Visitor-based operations** is handled by storing input data in the visitor and accumulating output data during the traversal. Methods in the visitor can retrieve and return the output data once traversal is complete.
13. A **CountingVisitor** example demonstrates how to compute the total number of cards in the source, storing the result within the visitor and retrieving it using a getter method.
14. A **ChecksContainmentVisitor** example shows how to check if a specific card exists in the card source hierarchy. It receives the card as input, stores the result internally, and provides a method (contains()) to access the result.
15. Efficiency can be improved in the **Visitor pattern** by adding logic to avoid unnecessary traversal. For example, in the **ChecksContainmentVisitor**, traversal of **CompositeCardSource** can be skipped if the card is already found.

Chapter 9

9.1

1. Object-Oriented Design and Limitations:

- Most software design principles are applied using classes and objects, consistent with the object-oriented programming (OOP) paradigm.
- However, there are situations where using objects for design solutions feels contrived, as illustrated by the use of function objects for sorting a list of cards.

2. Contrived Design Example:

- The `Collections.sort(...)` method requires a `Comparator<Card>` to compare cards, but what is provided is an object reference with methods for comparison, leading to a mismatch between the design goal (parameterizing sorting behavior) and the programming mechanism (passing an object).

3. Need for First-Class Functions:

- A better approach would be to pass the desired comparison behavior directly (as a function).
- This requires support for first-class functions, where functions can be treated as values—passed as arguments, stored in variables, and returned from other functions.

4. Java's Support for First-Class Functions:

- Since Java 8, Java emulates first-class functions through a syntax that allows method references.
- Example: Using `Card::compareByRank` to provide a function reference to the `sort` method, simplifying the code.

5. Syntactic Sugar vs. True First-Class Functions:

- The syntax of method references in Java provides the illusion of first-class functions but still internally converts the reference into an object (e.g., `Comparator<Card>`).

6. Significance of First-Class Functions:

- First-class functions enable higher-order functions (functions that take other functions as arguments).
- The use of higher-order functions can lead to clearer design, less code clutter, and better code reuse.

7. Higher-Order Functions and Functional Programming:

- Functions that take other functions as arguments are higher-order functions.
- The `Collections.sort` method is an example of a higher-order function.
- Designing applications around higher-order functions can lead to a functional programming (FP) style.

8. Functional Programming Paradigm:

- Full functional programming involves organizing computation by transforming data, ideally without mutating state.
- Functional programming is a broader paradigm that goes beyond just using higher-order functions.

9. Java and Functional Programming:

- Functional programming in Java, while limited, is a significant topic, and integrating functional elements into object-oriented design is important.
- The chapter introduces basic functional programming features to complement OOP.

10. Map-Reduce Programming Model:

- The final section of the chapter introduces the map–reduce programming model, which brings the reader closer to full functional programming in the context of this book.

9.2

1. Three mechanisms for first-class functions in Java:

- Functional interfaces
- Lambda expressions
- Method references

2. Functional Interfaces:

- A functional interface is an interface with a single abstract method (SAM).
- Example: Filter interface with an `accept(Card pCard)` method.
- Functional interfaces define function types, allowing behavior to be treated like functions.
- They can include static and default methods, but only one abstract method.
- Example: `Comparator<T>` is a functional interface, defining $(T,T) \rightarrow \text{int}$.
- Java 8 introduced common functional interfaces in `java.util.function`, such as `Predicate<T>`, representing functions that return a boolean.

3. Lambda Expressions:

- Lambda expressions provide a concise way to implement functional interfaces.
- They eliminate the need for anonymous classes and are more compact.
- Syntax: `(parameters) -> expression | block`.
- Example: `Predicate<Card> blackCardFilter = (Card card) -> card.getSuit().getColor() == Suit.Color.BLACK;`
- Java compiler checks lambda expressions for compatibility with functional interfaces, including parameter and return types.
- Lambda expressions allow omitting parameter types and parentheses for single parameters.

4. Method References:

- Method references are shorthand for calling methods directly in functional-style programming.
- Example: `cards.removeIf(Card::hasBlackSuit)` passes a reference to the `hasBlackSuit` method.
- Can reference static methods (e.g., `CardUtils::hasBlackSuit`), instance methods (e.g., `deck::topSameColorAs`), or instance methods of a specific object.
- Java allows method references in place of lambda expressions, reducing duplication.

- The method reference needs to match the functional interface's method signature, including parameter and return types.
- Method references offer an elegant and concise way to reuse existing methods in functional-style code.

9.3

Here is a summarized version of the text in enumerated points:

1. **First-Class Functions in Java:**

- First-class functions enable defining small pieces of behavior (like filtering or comparing).
- The principle of divide and conquer can be applied to create more complex behavior from simpler components.

2. **Initial Card Comparison:**

- A `Comparator<Card>` interface is used for comparing two cards based on their suit, using a lambda expression.

3. **Handling Ties in Comparison:**

- To handle ties (same suit), a secondary comparison by rank is added, creating a more complex lambda expression for sorting.

4. **Issues with Direct Implementation:**

- The direct approach requires writing multiple comparators to handle all possibilities (e.g., rank then suit, suit then rank, ascending/descending), leading to duplicated code.
- There are eight combinations of card comparison needed (by suit then rank, by rank then suit, and ascending/descending variations).

5. **Simplifying Comparators with Factory Methods:**

- A simpler approach involves creating factories for single-level comparisons (by rank or suit) and combining them for more complex comparisons (rank then suit, suit then rank).
- This reduces complexity but still involves repetition in code.

6. **Using Comparator Helper Methods:**

- The Comparator interface provides static and default methods (comparing() and thenComparing()) to compose comparison functions, reducing redundancy.
- Example: byRankComparator() can be rewritten using Comparator.comparing() to extract the rank.

7. Cascading Comparisons:

- The thenComparing() method allows cascading comparisons, such as comparing first by rank, then by suit.

8. Reversing Order of Comparisons:

- To reverse order (e.g., descending), the reversed() method is used, simplifying the code and avoiding duplication.

9. Final Simplified Solution:

- By composing comparators with comparing(), thenComparing(), and reversed(), the eight possible comparison orders can be expressed without duplication, resulting in clean and explicit code.

10. Eliminating Redundant Factory Methods:

- Removing the need for factory methods for comparators simplifies the code and reduces speculative generality (providing unnecessary services).

11. Using Comparator in Practice:

- Using methods like comparing(), thenComparing(), and reversed(), developers can directly define comparison behavior where needed, e.g., sorting cards by suit and rank.

12. Improving Code with Static Imports and Method References:

- Static imports eliminate the need to qualify static methods, and method references can simplify lambda expressions, further improving code clarity and conciseness.

13. Leveraging Overloaded Methods for Concise Code:

- thenComparing() can be used with a function that directly provides the value to compare, reducing boilerplate and making code more compact and readable.

14. General Principle for Functional Composition:

- The principle of composing first-class functions using library methods (like those in `java.util.function`) can be applied across various contexts, improving flexibility and readability.

15. Example with Predicate:

- A Predicate for filtering cards can be negated to create a filter for red cards, showcasing the power of function composition for common tasks.

9.4

Here is a summarized version of the text in enumerated points:

1. Deferred Processing with First-Class Functions:

- First-class functions allow us to defer processing until it is needed.
- The section explores three common types of deferred processing: supplying an object, consuming an object, and mapping an object to another object.

2. Infinite Card Source:

- An `InfiniteCardSource` is introduced, which needs to return an infinite number of cards.
- Instead of initializing all cards, it is initialized with a "card factory" (a function to return a `Card`), allowing deferred creation of cards.

3. Using Supplier for Deferred Object Creation:

- Java's `Supplier<T>` interface captures the behavior of a method that returns an object (`() -> Card` in this case).
- The `InfiniteCardSource` class is constructed using a `Supplier<Card>` to provide cards on demand.
- Examples include creating a random card source (`Card::random`) or a specific card source (e.g., `Ace of Hearts`).

4. Deferring Execution with Consumer:

- Another common deferred processing pattern is consuming an object. A `Consumer<Card>` is used to define what happens when a card is drawn.

- The ConsumingDecorator class wraps a CardSource and executes a function (e.g., printing the card) when a card is drawn.
- Example usage: ConsumingDecorator used to print every card drawn from a deck using System.out::println.

5. Method Reference Compatibility:

- The example with ConsumingDecorator demonstrates how method references (like System.out::println) are compatible with functional interfaces, even when their parameter types differ (e.g., Object → void vs. Card → void).

6. Function for Mapping Objects:

- For cases where both supplying and consuming are needed, a function type T → R (e.g., Function<T,R>) is used to map one type to another.
- Example: The Comparator.comparing() method uses a Function<Card, Suit> to extract the suit from a Card for comparison.

7. Using Function in Comparison:

- The Comparator.comparing() method creates a comparator based on a function that extracts a key for comparison.
- Example: Comparator<Card> bySuit = Comparator.comparing(Card::getSuit); compares cards based on their suit using the Function<Card, Suit>.

8. How Comparator.comparing() Works:

- The comparing method takes a Function<Card, Suit> as an argument and uses it to extract the Suit from a Card object.
- The function is applied only when the compare() method is called, demonstrating deferred processing of the comparison behavior.

9. Indirection in Function Application:

- When compare() is called, apply() (of the function) is executed on-demand, mapping the Card to its Suit.
- This approach allows comparison to happen on-demand without immediate execution when defining the comparator.

9.5

Here is a summarized version of the text in enumerated points:

1. Polymorphism in Design Patterns:

- Many design patterns rely on polymorphism for variation points, allowing different behaviors to be selected at runtime.
- Examples include the **STRATEGY** pattern (dynamic selection of algorithms) and the **OBSERVER** pattern (notifying observers dynamically).

2. Functional-Style Design for Behavior Parameterization:

- In functional-style design, first-class functions offer a different way to parameterize behavior by defining functions instead of creating objects and extending classes.
- This allows design patterns like **STRATEGY** and **OBSERVER** to be implemented using functions.

3. Functional-Style STRATEGY:

- In stateless cases, the **STRATEGY** pattern can be implemented using a functional interface like `Function<List<Card>, Card>`, where `apply` defines the card selection strategy.
- Example: `AutoPlayer` class uses a `Function` to decide which card to select from a list.
- Strategies can be defined on the fly, such as `cards -> cards.get(0)` for selecting the first card, or stored in utility classes for reuse (e.g., `CardSelection::lowestBlackCard`).

4. Limitations of General Functional Interfaces:

- Using general interfaces like `Function` can make the code less readable and less explicit, as it doesn't capture specific behaviors (e.g., handling empty lists).
- To improve clarity and design, a more specific functional interface like `CardSelectionStrategy` can be defined, which includes preconditions and postconditions, such as ensuring the list is not empty.

5. CardSelectionStrategy Interface:

- The CardSelectionStrategy interface defines a method select(List<Card> pCards) that returns an Optional<Card>.
- Standard strategies (e.g., first, lowestBlackCard, highestFaceCard) can be provided to handle specific card selection behaviors while guarding against empty lists.

6. Functional-Style OBSERVER:

- The **OBSERVER** pattern can be implemented using functional-style design, where the callback method for observers is defined by a function type like Card → void.
- Example: ObservableDeck class extends Deck and uses a Consumer<Card> to notify observers each time a card is drawn.

7. ObservableDeck Implementation:

- The ObservableDeck class accepts a Consumer<Card> to handle notifications when cards are drawn.
- Observers (e.g., System.out::println) can be passed to log each card drawn from the deck.

8. Comparison with ConsumingDecorator:

- The design of ObservableDeck contrasts with the ConsumingDecorator from earlier, as ObservableDeck uses inheritance, while ConsumingDecorator used aggregation.
- Both designs achieve similar functionality using the Consumer<Card> interface to inject behavior based on observable events.

9. Key Insight:

- First-class functions enable a functional rethinking of classic object-oriented design patterns like **STRATEGY** and **OBSERVER**, offering compact, flexible implementations.

9.6

Here's a summarized version of the key points from the provided text:

1. Functional-Style Programming in Object-Oriented Design:

- Functional programming is useful for tasks involving data transformations, such as processing sequences of data elements.
- An example is counting acronyms in text, where the transformation is filtering and counting occurrences.

2. Higher-Order Functions in Functional Programming:

- Higher-order functions apply general strategies to data, parameterized with specific context functions (e.g., checking for acronyms).
- Functional-style design enhances readability and reusability by decoupling general strategies from specific conditions.

3. Streams vs. Collections:

- A stream represents a flow of data, computed on-demand, unlike collections that store data.
- Streams can be infinite, whereas collections are finite.
- Streams are traversed only once, unlike collections which can be traversed multiple times.
- Streams can be parallelized, whereas collections require external iteration code.

4. Java's Stream API:

- Java 8 introduces the Stream API to support functional-style design, including first-class functions (lambdas, method references).
- Streams are created using the `stream()` method on collections.

5. Operations on Streams:

- Streams support useful operations like `count()`, `sorted()`, `limit()`, and `distinct()`.
- Streams can be combined using operations like `Stream.concat()` and transformed using methods like `filter()` and `map()`.

6. Higher-Order Functions on Streams:

- Methods like `forEach()` apply functions to all elements of the stream, with `forEachOrdered()` maintaining order.

- Terminal operations (e.g., `count()`, `allMatch()`, `anyMatch()`) process and reduce the stream to a result.

7. Filtering Streams:

- The `filter()` method creates new streams by applying predicates, allowing selective transformations (e.g., counting face cards).
- Method references (`Card::isFaceCard`) can be used for cleaner and more self-explanatory code.

8. Mapping Data Elements:

- `map()` transforms each element of a stream using a function.
- Specialized types like `IntStream` and `DoubleStream` handle numeric streams with additional operations (e.g., `sum()`).

9. FlatMap for One-to-Many Mapping:

- `flatMap()` is used to transform one element into a stream of multiple elements (e.g., extracting cards from decks).

10. Reducing Streams:

- Reductions aggregate the stream into a single result (e.g., `count()`, `sum()`).
- The `reduce()` method accumulates values using a binary operator, such as summing integers.
- Common reduction operations (`min`, `max`, `sum`) are directly supported by streams.

11. Collectors for Accumulating Results:

- Collectors provide a functional way to accumulate elements into a data structure (e.g., using `Collectors.toList()` to collect face cards into a list).
- This approach maintains a declarative style, avoiding explicit list manipulations.

This summary captures the key ideas related to functional-style programming in Java, particularly with respect to streams and higher-order functions.