Chapter 10
Implementing Subprograms and Scoping Mechanisms

1. Recursion and Activation Record Instances

- **Activation Record Instance:** A runtime structure containing data for a specific subprogram call, including local variables, parameters, return address, and execution status.
- **Dynamic Link:** Points to the activation record of the caller, enabling access to dynamic scoping.
- **Static Link:** Points to the static parent of the current subprogram, facilitating static scoping and access to non-local variables.
- **Trace Output for Recursive Programs:**
  - Recursive calls dynamically create multiple activation record instances on the runtime stack.
  - Each record is linked using the dynamic and static links to ensure both dynamic and static variable resolutions.
  - Example: Computing factorial using recursion stacks activation records, with each maintaining its local variables (`n`) and returning a result through the stack.

2. Dynamic Scoping Mechanisms

Dynamic scoping resolves variable references based on the runtime call chain rather than lexical structure.

- **Deep Access:**
  - Locates variables by traversing the dynamic chain of activation records.
  - Pros: Simple implementation, as it relies on existing activation record structures.
  - Cons: Variable lookup can be slow for deep nesting; the runtime cost depends on the call chain length.
- **Shallow Access:**
  - Uses a central table where each variable name has an entry pointing to its most recent value.
  - Pros: Access is faster since the table is directly queried, and updates are efficient.
  - Cons: Requires maintaining a global structure, which increases memory usage and implementation complexity.

3. Key Comparisons

- **Deep Access:** Optimized for systems with infrequent non-local references but costly when dynamic chains are long.
- **Shallow Access:** Favored for frequent access to non-local variables, with predictable access times.

This foundational understanding is critical for implementing robust and efficient subprograms in dynamic and static scoping environments

Ch 11 Abstraction, Information Hiding, and Encapsulation in OOP

1. Abstraction

- **Definition:** Abstraction refers to representing only the essential features of an entity while hiding its complexities. It provides a simplified model that captures the most significant attributes.
- **Importance in Programming:**
  - Enables clear and simplified design.
  - Programming languages use abstraction for process abstraction (subprograms) and data abstraction (abstract data types or classes).
- **Example in OOP:** A `Stack` class in C++ or Java abstracts the stack operations (`push`, `pop`) without exposing the internal array management.

2. Information Hiding

- **Definition:** The principle of restricting access to certain details of an implementation to prevent misuse or dependency on the internal representation.
- **Features in OOP:**
  - **Access Modifiers:**
    - `private` (C++, Java, C#): Hides members from external code.
    - `protected`: Allows access within inheritance hierarchies.
    - `internal` (C#): Limits access to within the same assembly.
  - **Encapsulation:** Combines data and methods in a class, ensuring that internal workings are not exposed directly.

- **Example:** In C++, marking variables `private` ensures that external functions cannot modify them directly, only through controlled access via public methods.

3. **Encapsulation**

- **Definition:** Encapsulation is the bundling of data and related methods into a single unit (class) and restricting direct access to some of the object's components.
- **Benefits:**
  - Promotes modular design.
  - Enhances security by preventing unintended interference.
  - Facilitates code maintainability and scalability.
- **Example in OOP:**
  - C++: Uses classes with public and private sections.
  - Java: Employs `class` constructs with access control via modifiers.

Synchronization and Cooperation

1. **Synchronization**: Coordination to manage the order of task execution in concurrent programming. Synchronization ensures tasks access shared resources and complete in a predictable, conflict-free manner.
   - **Cooperation Synchronization**: Tasks depend on each other to complete specific actions. Example: **Producer-Consumer Problem** (producer fills a buffer, consumer empties it).
   - **Competition Synchronization**: Tasks compete for a shared resource that cannot be used simultaneously. Example: Mutual exclusion in accessing a shared counter.
2. **Sync vs. Competition Sync**:
   - **Cooperation Sync**: Used when Task A depends on Task B to complete before it can proceed. For instance, ensuring buffer availability before adding/removing items.
   - **Competition Sync**: Used when tasks contend for a critical resource. It ensures only one task accesses it at a time, avoiding conditions like inconsistent updates.

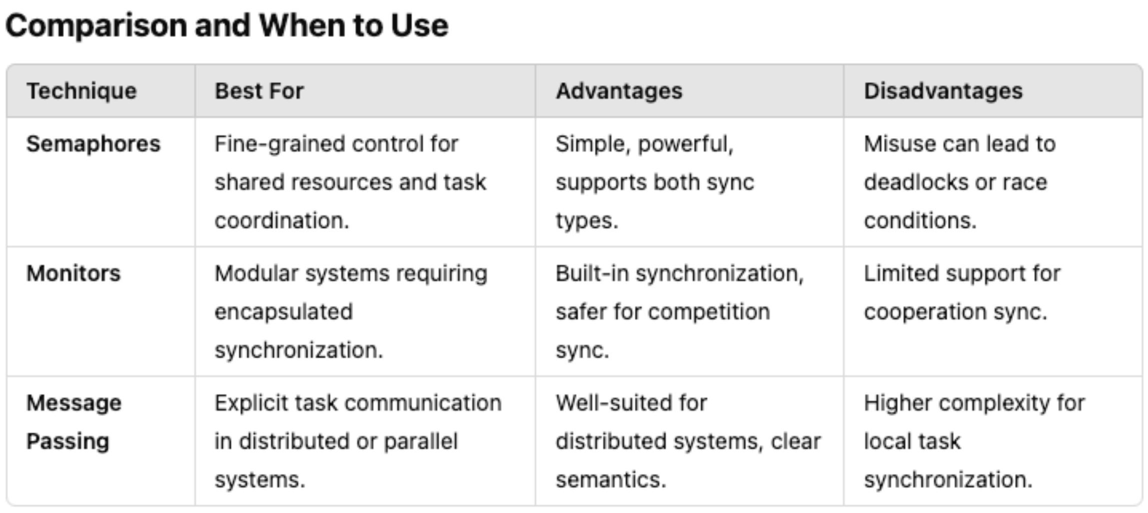
Techniques for Handling Concurrency Issues

1. **Semaphores**:
   - Introduced by Dijkstra in 1965, semaphores consist of:
     - A counter.
     - A queue for storing task descriptors (which track execution state).
   - **Key Operations**:
     - **Wait (P)**: Decrements the semaphore counter or places the calling task in a queue if the counter is zero.
     - **Release (V)**: Increments the semaphore counter or allows a waiting task to proceed.
   - **Applications**:
     - **Cooperation Sync**: Used to control dependent actions like producer-consumer. Counters track buffer states (e.g., `emptyspots` and `fullspots`).
     - **Competition Sync**: Binary semaphores (`0` or `1`) ensure mutual exclusion for critical resources.
2. **Example**:
   - **Producer** waits for empty spots, deposits data, and increments the full spots.
   - **Consumer** waits for full spots, fetches data, and increments the empty spots.
3. **Evaluation**:
   - Effective but prone to misuse (e.g., missing releases causing deadlocks or overflows).

2. **Monitors**:
   - Encapsulate shared data and operations in a single construct to restrict access.
   - Supported in languages like **Java**, **C#**, and **Ada**.
   - Guarantees synchronized access by:
     - Allowing only one task to access the monitor at a time.
     - Queueing subsequent calls if the monitor is busy.
   - **Applications**:
     - Effective for **competition sync**, as access is inherently exclusive.

        - Requires careful programming for **cooperation sync**, such as ensuring buffer overflows/underflows do not occur.

2. **Evaluation**:
    - A safer and more structured alternative to semaphores for competition synchronization.
3. **Message Passing**:
    - A model for task communication where tasks send and receive messages (rendezvous).
    - **Core Mechanisms**:
        - **Task readiness**: Tasks indicate when they are willing to accept messages.
        - **Rendezvous**: Actual message transmission when both sender and receiver are ready.
    - **Applications**:
        - **Cooperation Sync**: Guarded accept clauses ensure tasks synchronize only when preconditions (like buffer state) are met.
        - **Competition Sync**: Synchronization is implicit, as tasks accept messages one at a time.
    - **Best For**:
        - Distributed systems or when task communication is central.



Comparison and When to Use

Technique	Best For	Advantages	Disadvantages
Semaphores	Fine-grained control for shared resources and task coordination.	Simple, powerful, supports both sync types.	Misuse can lead to deadlocks or race conditions.
Monitors	Modular systems requiring encapsulated synchronization.	Built-in synchronization, safer for competition sync.	Limited support for cooperation sync.
Message Passing	Explicit task communication in distributed or parallel systems.	Well-suited for distributed systems, clear semantics.	Higher complexity for local task synchronization.

Dynamic Binding and Its Role

Dynamic binding (also known as *late binding*) is the process where the method to be executed is determined at runtime, rather than compile time. This feature allows for polymorphism in object-oriented programming.

In Java, dynamic binding occurs when:

1. A method is overridden in a subclass.
2. A call is made to the method using a reference of the parent class pointing to an object of the subclass.

How It Works Internally: Class Instance Records and Vtables

1. **Class Instance Records (CIRs):**
    - These are memory layouts that store information about an object of a class.
    - They contain references to instance variables, including a pointer to the *vtable* for the object's class.
2. **Vtables (Virtual Tables):**

- A **vtable** is a table of function pointers used for dynamic method dispatch.
- Each class has its vtable that stores pointers to the methods defined in that class. If a subclass overrides a method, the vtable entry for that method in the subclass replaces the parent's pointer.

Dynamic Binding Workflow with Vtables:

1. At runtime, when a method is called on an object, the JVM uses the object's CIR to locate the appropriate vtable.
2. The vtable is checked to find the method implementation for the specific class of the object.
3. The function pointer in the vtable is invoked, ensuring the correct method for the object's actual class is executed.

```
class Parent {
   void display() {
      System.out.println("Display method in Parent");
   }
}

class Child extends Parent {
   @Override
   void display() {
      System.out.println("Display method in Child");
   }
}

public class DynamicBindingExample {
   public static void main(String[] args) {
      Parent obj = new Child(); // Parent reference, Child object
      obj.display(); // Dynamic binding
   }
}
```

Output:

```
Display method in Child
```

- Here, `obj` is of type `Parent`, but the method `display` from `Child` is invoked due to dynamic binding.

Reflection in Java

Reflection is a feature in Java that allows a program to inspect or modify the behavior of classes, methods, and fields at runtime. It is provided by the `java.lang.reflect` package.

Key Features of Reflection:

1. **Inspecting Class Structure:**
   - Retrieve information about fields, methods, constructors, etc.
2. **Accessing Private Members:**
   - You can bypass access restrictions to access private fields/methods.
3. **Dynamic Method Invocation:**
   - Invoke methods at runtime, even if they were not known during compile time.

```
import java.lang.reflect.Method;

class Example {
   private void display() {
        System.out.println("Private display method called.");
    }
}

public class ReflectionExample {
    public static void main(String[] args) throws Exception {
        Example obj = new Example();

        // Get the Class object
        Class<?> clazz = obj.getClass();

        // Get the private method 'display'
        Method method = clazz.getDeclaredMethod("display");

        // Make it accessible
        method.setAccessible(true);

        // Invoke the method on the object
        method.invoke(obj);
    }
}
```

Output:

```
Private display method called.
```

Use Cases of Reflection:

- **Framework Development:** Frameworks like Spring and Hibernate use reflection for dependency injection and ORM mapping.
- **Dynamic Proxy:** Reflection is used to create dynamic proxies at runtime.
- **Testing:** Tools like JUnit use reflection to discover and execute test cases.

Reflection Considerations:

1. **Performance Overhead:** Reflection is slower because it involves type inspection at runtime.
2. **Security Risks:** It can bypass normal access controls, making the code vulnerable if misused.
3. **Complexity:** Debugging and maintenance become harder with excessive use of reflection.