

UNIT-4

OBJECT ORIENTED LANGUAGES

Abstraction:

The concept of abstraction is fundamental in programming

Nearly all programming languages support process abstraction with subprograms

Nearly all programming languages designed since 1980 have supported data abstraction with some kind of module

Encapsulation:

– Original motivation:

Large programs have two special needs:

Some means of organization, other than simply division into subprograms

Some means of partial compilation (compilation units that are smaller than the whole program)

Obvious solution: a grouping of subprograms that are logically related into a unit that can be separately compiled

– These are called encapsulations

Examples of Encapsulation Mechanisms:

Nested subprograms in some ALGOL-like languages (e.g., Pascal)

FORTRAN 77 and C - Files containing one or more subprograms can be independently compiled

– FORTRAN 90, C++, Ada (and other contemporary languages) - separately compilable modules

Definitions: An abstract data type is a user-defined datatype that satisfies the following two conditions:

Definition 1: The representation of and operations of objects of the type are defined in a single syntactic unit; also, other units can create objects of the type.

Definition 2: The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition.

Concurrency can occur at four levels:

- **Machine instruction level**
- **High-level language statement level**
- **Unit level**
- **Program level**

Because there are no language issues in instruction- and program-level concurrency, they are not addressed here

The Evolution of Multiprocessor Architectures:

Late 1950s - One general-purpose processor and one or more special-purpose processors for input and output operations

 **Early 1960s** - Multiple complete processors, used for program-level concurrency

– **Mid-1960s** - Multiple partial processors, used for instruction-level concurrency

– **Single-Instruction Multiple-Data (SIMD) machine** The same instruction goes to all processors, each with different data - e.g., *vector processors*

– **Multiple-Instruction Multiple-Data (MIMD) machines**, Independent processors that can be synchronized (unit-level concurrency)

Def: A thread of control: in a program is the sequence of program points reached as control flows through the program

Categories of Concurrency:

– **Physical concurrency** - Multiple independent processors (multiple threads of control)

– **Logical concurrency** - The appearance of physical concurrency is presented by time sharing one processor (software can be designed as if there were multiple threads of control)

Coroutines provide only quasiconcurrency

Reasons to Study Concurrency:

It involves a new way of designing software that can be very useful--many real-world situation involve concurrency

Computers capable of physical concurrency are now widely used

Fundamentals (for stmt-level concurrency) :

Def: A *task* is a program unit that can be in concurrent execution with other program units

– Tasks differ from ordinary subprograms in that:

A task may be implicitly started

When a program unit starts the execution of a task, it is not necessarily suspended

When a task's execution is completed, control may not return to the caller

Def: A task is *disjoint* if it does not communicate with or affect the execution of any other task in the program in any way Task communication is necessary for synchronization

– **Task communication can be through:**

Shared nonlocal

variables Parameters

Message passing

– **Kinds of synchronization:**

Cooperation

Task A must wait for task B to complete some specific activity before task A can continue its execution

e.g., the producer-consumer problem

Competition

When two or more tasks must use some resource that cannot be simultaneously used, a shared counter. A problem because operations are not atomic

Competition is usually provided by **mutually exclusive access** (methods are discussed later)

Providing synchronization requires a mechanism for delaying task execution

Task execution control is maintained by a program called the scheduler, which maps task execution onto available processors

Tasks can be in one of several different execution states:

New - created but not yet started

Runnable or ready - ready to run but not currently running (no available processor)

Running

Blocked - has been running, but cannot not continue (usually waiting for some event to occur)

Dead - no longer active in any sense

Liveness is a characteristic that a program unit may or may not have

In sequential code, it means the unit will eventually complete its execution

In a concurrent environment, a task can easily lose its liveness

If all tasks in a concurrent environment lose their liveness, it is called *deadlock*

– **Design Issues for Concurrency:**

- How is cooperation synchronization provided?
- How is competition synchronization provided?
- How and when do tasks begin and end execution?
- Are tasks statically or dynamically created? **Example:**

A buffer and some producers and some consumers

Technique: Attach two SIGNAL objects to the buffer, one for full spots and one for empty spot

Methods of Providing Synchronization:

Semaphores


Monitors

Message Passing

– Semaphores (Dijkstra - 1965)

- A **semaphore** is a data structure consisting of a counter and a queue for storing task descriptors
- Semaphores can be used to implement guards on the code that accesses shared data structures
- Semaphores have only two operations, wait and release (originally called P and V by Dijkstra)
- Semaphores can be used to provide both competition and cooperation synchronization

Cooperation Synchronization with Semaphores: *Example: A shared buffer*

 The buffer is implemented as an ADT with the operations **DEPOSIT** and **FETCH** as the only ways to access the buffer.

Use two semaphores for cooperation:

Empty spots and full spots

- The semaphore counters are used to store the numbers of empty spots and full spots in the buffer
- DEPOSIT must first check empty spots to see if there is room in the buffer
- If there is room, the counter of empty spots is decremented and the value is inserted
- If there is no room, the caller is stored in the queue of empty spots
- When DEPOSIT is finished, it must increment the counter of full spots

FETCH must first check full spots to see if there is a value

- If there is a full spot, the counter of full spots is decremented and the value is removed

- If there are no values in the buffer, the caller must be placed in the queue of full spots
- When FETCH is finished, it increments the counter of empty spots
- The operations of FETCH and DEPOSIT on the semaphores are accomplished through two semaphore operations named wait and release wait (aSemaphore)

if a Semaphore's counter > 0 then Decrement aSemaphore's counter else

Put the caller in aSemaphore's queue Attempt to transfer control to some ready task (If the task ready queue is empty, deadlock occurs) end

release(aSemaphore)

if aSemaphore's queue is empty then Increment aSemaphore's counter

else

Put the calling task in the task ready queue Transfer control to a task from aSemaphore's queue

end

- Competition Synchronization with Semaphores:

A third semaphore, named access, is used to control access (competition synchronization)

- The counter of access will only have the values 0 and 1
- Such a semaphore is called a **binary semaphore**

SHOW the complete shared buffer example - Note that wait and release must be atomic!

Evaluation of Semaphores:

- Misuse of semaphores can cause failures in cooperation synchronization
e.g., the buffer will overflow if the wait of full spots is left out
- Misuse of semaphores can cause failures in competition synchronization e.g., The program will deadlock if the release of access is left out.

Monitors :(Concurrent Pascal, Modula, Mesa)

The idea: encapsulate the shared data and its operations to restrict access

A *monitor* is an abstract data type for shared data show the diagram of monitor buffer operation,

- **Example language:** Concurrent Pascal

Concurrent Pascal is Pascal + classes, processes (tasks), monitors, and the queue data type (for semaphores)

Example language: Concurrent Pascal (continued) processes are types Instances are statically created by declarations

An instance is —started by `init`, which allocates its local data and begins its execution

– Monitors are also types Form:

```
type some_name = monitor (formal parameters) shared variables , local
procedures exported procedures (have entry in definition) initialization code
```

Competition Synchronization with Monitors:

- Access to the shared data in the monitor is limited by the implementation to a single process at a time; therefore, mutually exclusive access is inherent in the semantic definition of the monitor
- Multiple calls are queued

Cooperation Synchronization with Monitors:

Cooperation is still required - done with semaphores, using the queue data type and the built-in operations, `delay` (similar to `send`) and `continue` (similar to `release`)

`delay` takes a queue type parameter; it puts the process that calls it in the specified queue and removes its exclusive access rights to the monitor's data structure

Differs from `send` because `delay` always blocks the caller

`continue` takes a queue type parameter; it disconnects the caller from the monitor, thus freeing the monitor for use by another process.

It also takes a process from the parameter queue (if the queue isn't empty) and starts it,

Differs from `release` because it always has some effect (`release` does nothing if the queue is empty)

Java Threads

The concurrent units in Java are methods named `run`

- A `run` method code can be in concurrent execution with other such methods
- The process in which the `run` methods execute is called a *thread*

```
Class myThread extends Thread public void run () {...}
```

```
}
```

```
...
```

```
Thread myTh = new MyThread (); myTh.start();
```

Controlling Thread Execution

The Thread class has several methods to control the execution of threads

The yield is a request from the running thread to voluntarily surrender the processor

The sleep method can be used by the caller of the method to block the thread

The join method is used to force a method to delay its execution until the run method of another thread has completed its execution

Thread Priorities

- A thread's default priority is the same as the thread that create it.
- If main creates a thread, its default priority is NORM_PRIORITY
- Threads defined two other priority constants, MAX_PRIORITY and MIN_PRIORITY
- The priority of a thread can be changed with the methods setPriority

Cooperation Synchronization with Java Threads

Cooperation synchronization in Java is achieved via wait, notify, and notifyAll methods

- All methods are defined in Object, which is the root class in Java, so all objects inherit them
- The wait method must be called in a loop
- The notify method is called to tell one waiting thread that the event it was waiting has happened
- The notifyAll method awakens all of the threads on the object's wait list

Java's Thread Evaluation

- Java's support for concurrency is relatively simple but effective
- Not as powerful as Ada's tasks

C# Threads

- Loosely based on Java but there are significant differences
- Basic thread operations
- Any method can run in its own thread

~~– A thread is created by creating a Thread object~~

- Creating a thread does not start its concurrent execution; it must be requested through the Start method

- A thread can be made to wait for another thread to finish with Join
- A thread can be suspended with Sleep
- A thread can be terminated with Abort

Synchronizing Threads

- Three ways to synchronize C# threads
- The Interlocked class
- Used when the only operations that need to be synchronized are incrementing or decrementing of an integer
- The lock statement
- Used to mark a critical section of code in a thread lock (expression) { ... }
- The Monitor class
- Provides four methods that can be used to provide more

EXCEPTION HANDLING

In a language without exception handling:

When an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated

In a language with exception handling:

Programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing. Many languages allow programs to trap input/ output errors (including EOF) **Definition 1:**

An exception is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing

Definition 2: The special processing that may be required after the detection of an exception is called exception handling

Definition 3: The exception handling code unit is called an exception handler

Definition 4: An exception is raised when its associated event occurs

A language that does not have exception handling capabilities can still define, detect, raise, and handle exceptions

– Alternatives:

- Send an auxiliary parameter or use the return value to indicate the return status of a Subprogram

e.g., C standard library functions



Pass a label parameter to all subprograms (error return is to the passed label)
e.g., FORTRAN



Pass an exception handling subprogram to all subprograms

Advantages of Built-in Exception Handling:

- Error detection code is tedious to write and it clutters the program
- Exception propagation allows a high level of reuse of exception handling code

Design Issues for Exception Handling:

- How and where are exception handlers specified and what is their scope?
- How is an exception occurrence bound to an exception handler?
- Where does execution continue, if at all, after an exception handler completes its execution?
- How are user-defined exceptions specified?
- Should there be default exception handlers for programs that do not provide their own?
- Can built-in exceptions be explicitly raised?
- Are hardware-detectable errors treated as exceptions that can be handled?
- Are there any built-in exceptions?
- How can exceptions be disabled, if at all?

PL/I Exception Handling

Exception handler form:

```
EX: ON condition [SNAP] BEGIN; ... END;
```

- condition is the name of the associated exception
- SNAP causes the production of a dynamic trace to the point of the exception
- Binding exceptions to handlers

It is dynamic--binding is to the most recently executed ON statement

Continuation

- Some built-in exceptions return control to the statement where the exception was raised
- Others cause program termination

- User-defined exceptions can be designed to go to any place in the program that is labeled

Other design choices:

- User-defined exceptions are defined with: `CONDITION exception_name`
- Exceptions can be explicitly raised with: `SIGNAL CONDITION (exception_name)`
- Built-in exceptions were designed into three categories:

Those that are enabled by default but could be disabled by user code

Those that are disabled by default but could be enabled by user code

Those that are always enabled

Evaluation

The design is powerful and flexible, but has the following problems:

- Dynamic binding of exceptions to handler makes programs difficult to write and to read
- The continuation rules are difficult to implement and they make programs hard to read

LOGIC PROGRAM PARADIGM:

Based on logic and declarative programming 60's and early 70's, Prolog (Programming in logic, 1972) is the most well known representative of the paradigm.

- Prolog is based on Horn clauses and SLD resolution
- Mostly developed in fifth generation computer systems project
- Specially designed for theorem proof and artificial intelligence but allows general purpose computation.
- Some other languages in paradigm: ALF, Frill, Godel, Mercury, Oz, Ciao, Prolog, datalog, and CLP languages

Constrain Logic Programming:

Clause: disjunction of universally quantified literals, $\exists(L1 _ L2 _ \dots _ Ln)$

A logic program clause is a clause with exactly one positive literal $\exists(A _ \neg A1 _ \neg A2 \dots _ \neg An)$
 $_ \exists(A (A1 \wedge A2 \dots \wedge An)$

A goal clause: no positive literal $\exists(\neg A1 _ \neg A2 \dots _ \neg An)$

Proof: by refutation, try to un satisfy the clauses with a goal clause G. Find $\exists(G)$. Linear resolution for definite programs with constraints and selected atom. CLP on first order terms. (Horn clauses). Unification. Bidirectional. Backtracking. Proof search based on trial of all matching clauses

Prolog terms:

Atoms:

- 1 Strings with starting with a small letter and consist of o
[a-zA-Z 0-9]*
 - o a aDAM a1 2

- 2 Strings consisting of only punctuation

□ *** .+ .<.>.

- 3 Any string enclosed in single quotes (like an arbitrary string) o
'ADAM' 'Onur Sehitoglu' '2 * 4 < 6'

- Numbers

1234 12.32 12.23e-10

Variables:

- Strings with starting with a capital letter or and consist of

□ [a-zA-Z 0-9]*

- Adam adam A093

- is the universal match symbol. Not variable

Structures:

Starts with an atom head have one or more arguments (any term) enclosed in parenthesis, separated by comma structure head cannot be a variable or anything other than atom.

a(b) a(b,c) a(b,c,d) ++(12) +(*) *(1,a(b)) 'hello world'(1,2) p X(b) 4(b,c) a() ++() (3) ×
some structures defined as infix:

+(1,2) _ 1+2 , :-(a,b,c,d) _ a :- b,c,d Is(X,+(Y,1)) _ X is X + 1

Prolog terms:

Atoms:

- 1 Strings with starting with a small letter and consist of o
[a-zA-Z 0-9]*
 - o a aDAM a1 2

- 2 Strings consisting of only punctuation
 - *** .+ .<.>.

- 3 Any string enclosed in single quotes (like an arbitrary string) o
'ADAM' 'Onur Sehitoglu' '2 * 4 < 6'

- Numbers
1234 12.32 12.23e-10

Variables:

- Strings with starting with a capital letter or and consist of
 - [a-zA-Z 0-9]*

- Adam adam A093

- is the universal match symbol. Not variable

Structures:

Starts with an atom head have one or more arguments (any term) enclosed in parenthesis, separated by comma structure head cannot be a variable or anything other than atom.

a(b) a(b,c) a(b,c,d) ++(12) +(*) *(1,a(b)) 'hello world'(1,2) p X(b) 4(b,c) a() ++() (3) ×
some structures defined as infix:

+(1,2) _ 1+2 , :- (a,b,c,d) _ a :- b,c,d Is(X,+(Y,1)) _ X is X + 1

Static sugars:

Prolog interpreter automatically maps some easy to read syntax into its actual structure. List: $[a,b,c] _ .(a,(b,(c,[])))$

Head and Tail: $[H|T] _ .(H,T)$

String: $"ABC" _ [65,66,67]$ (ascii integer values) use `display (Term)`. to see actual structure of the term

Unification:

Bi-directional (both actual and formal argument can be instantiated).

if S and T are atoms or number, unification successful only if $S = T$

if S is a variable, S is instantiated as T , if it is compatible with current constraint store (S is instantiated to another term, they are unified)

if S and T are structures, successful if: head of $S =$ head of T they have same arity
unification of all corresponding terms are successful.

S : list of structures, P current constraint store

$s _ 2 S$, $\text{arity}(s)$: number of arguments of structure, $s _ 2 S$, $\text{head}(s)$: head atom of the structure,

$s _ 2 S$, $\text{argi}(s)$: i th argument term of the structure, $p _ P$: p is consistent with current constraint store. $S _ T; P = (S, T _ 2 A _ S, T _ 2 N) \wedge S = T \text{ ! true } \wedge S _ T \mid = P \text{ ! true}; S _ T \wedge P _ T _ 2 V \wedge S _ T \mid = P \text{ ! true};$

$S _ T \wedge P _ S, T _ 2 S \wedge \text{head}(S) = \text{head}(T) \wedge \text{arity}(S) = \text{arity}(T) \text{ !}$

$\forall i, \text{argi}(S) _ \text{argi}(T);$

P Unification Example: $X = a \text{ ! } p$ with $X = a(X,3) = a(X,3,2) \text{ ! } \times a(X,3) = b(X,3) \text{ ! } \times a(X,3) = a(3,X) \text{ ! } p$ with $X = 3$ $a(X,3) = a(4,X) \text{ ! } \times a(X,b(c,d(e,f))) = a(b(c,Y),X) \text{ ! } X = b(c,d(e,f)), Y = d(e,F)$

Declarations:

Two types of clauses:

$p_1(\text{arg1}, \text{arg2}, \dots) \text{ :- } p_2(\text{args}, \dots), p_3(\text{args}, \dots)$. means if p_2 and p_3 true, then p_1 is true. There can be arbitrary number of (conjunction of) predicates at right hand side.

$p(\text{arg1}, \text{arg2}, \dots)$. sometimes called a fact. It is equivalent to: $p(\text{arg1}, \text{arg2}, \dots) \text{ :- true}$.
 $p(\text{args}) \text{ :- } q(\text{args}); s(\text{args})$.

Is disjunction of predicates. q or s implies p . Equivalent to: $p(\text{args}) \text{ :- } q(\text{args})$.
 $p(\text{args}) \text{ :- } s(\text{args})$.

A prolog program is just a group of such clauses.

Lists Example:

– list membership $\text{memb}(X, [X|\text{Rest}]) . \text{memb}(X, [|\text{Rest}]) :- \text{memb}(X, \text{Rest})$.

– concatenation $\text{conc}([],L,L)$.

$\text{conc}([X|R], L, [X|R \text{ and } L]) :- \text{conc}(R, L, R \text{ and } L)$.

– second list starts with first list prefix of $([],)$. Prefix of $([X|R_x], [X|R_y]) :- \text{prefix}(R_x, R_y)$.

– second list contains first list Sublist $(L1,L2) :- \text{prefix}(L1,L2)$. Sublist $(L,[R]) :- \text{sublist}(L,R)$.

Procedural Interpretation:

For goal clause all matching head clauses (LHS of clauses) are kept as backtracking points (like a junction in maze search) Starts from first match. To prove head predicate, RHS predicates need to be proved recursively. If all RHS predicates are proven, head predicate is proven. When fails, prolog goes back to last backtracking point and tries next choice. When no backtracking point is left, goal clause fails. All predicate matches go through unification so goal clause variables can be instantiated.

Arithmetic and Operations:

$X = 3+1$ is not an arithmetic expression!

operators (is) force arithmetic expressions to be evaluated all variables of the operations needs to be instantiated.

12 is 3+X does not work!

Comparison operators force LHS and RHS to be evaluated:

$X > Y, X < Y, X >= Y, X <= Y, X := Y, X == Y$

is operator forces RHS to be evaluated: $X \text{ is } Y+3*Y$ Y needs to have a numerical value when search hits this expression. Note that $X \text{ is } X+1$ is never successful in Prolog. Variables are instantiated once.

Greatest Common Divisor: $\text{gcd}(m, n) = \text{gcd}(n, m - n)$ if $n < m$ $\text{gcd}(m, n) = \text{gcd}(n, m)$ if $m < n$ $\text{gcd}(X, X, X)$.

$\text{gcd}(X, Y, D) :- X < Y, Y1 \text{ is } Y - X, \text{gcd}(X, Y1, D)$.

$\text{gcd}(X, Y, D) :- Y < X, \text{gcd}(Y, X, D)$.

Deficiencies of Prolog:

Resolution order control

The closed-world assumption

The negation problem

Intrinsic limitations

Applications of Logic Programming

Relational database management systems

Expert systems

Natural language processing





