Support for object-oriented programming

- Object oriented programming had its roots in SIMULA 67, but was not fully developed until the evolution of small talk.
- Small talk is considered as a base model for a purely object-oriented programming language.
- A language that is object-oriented must provide, support for three key language features:
  1) Abstract data types
  2) Inheritance
  3) Dynamic binding of method calls to methods.

Abstract datatype

- An abstraction is a view or representation of an entity that includes only the most significant attributes.
- There are two fundamental kinds of abstraction:
  1) process abstraction
  2) Data abstraction.
- process abstraction is achieved through subprograms, which provide a way for a program to specify a process, without specifying the details of how it performs the task.
- Data abstraction is specifying methods, which manipulate the data. Uncesary details about the data types will be hidden from the outside units.

- An abstract data type is a data structure, which encloses data items and has subprograms (operations) that manipulate its data.

- program units that use an abstract data type can declare variables of that type.

  Ex:- An instance of an abstract data type is called an object.

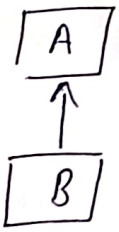
Inheritance

- Abstract data types can be used for reuse.
- The problems with the reuse of abstract data types is as follows:
  1) A new use of the ADT, might need atleast some minor modifications.
  2) All the existing ADT's might not be suitable for the problem, being addressed by the program.
- Inheritance offers solution to both modification problem and ~~problem~~ program organization problem.
- Inheritance allows inheriting the data and functionality of some existing type and is also allowed to modify some of those entities and add new entities.
- The abstract data types in object-oriented languages are usually called classes and class instances of abstract data types are called objects.

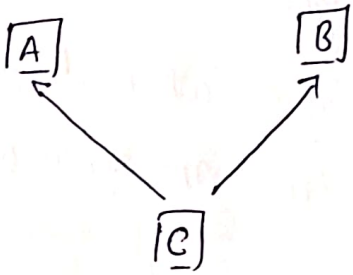
- A class that is defined from another class is derived class and the class from which it is derived is called parent class or Superclass.

- The subprograms that define the operations on objects of a class are called methods.

- If a new class is a subclass of a single parent class, then the derivation process is called single inheritance.



- If a new class has more than one parent class, the process is called multiple inheritance



- If the subclass modifies the behaviour of one or more inherited methods, such methods are called overridden methods.

Dynamic Binding

- This refers to the binding of method calls to the method definitions.

Ex:- Static binding

```
class A
{
    void print()
    {
        System.out.println (" A's class");
    }
}
```

```
class B extend class A
{
    void print()
    {
        Sy.out.println ("B's class");
    }
}
```

```
Public Static void main ( String args[] )
{
    A a1 = new A();
    a1. print();
}
```

- If the method "print" which is defined in both classes, is called through the p then the run-time system must determine, during execution, which method should be called. (A's or B's). It is done by determining which type object is currently referenced by the reference. This reference is called polymorphic reference.
- Abstract methods, Abstract classes.

Design issues for object-oriented languages

- A number of issues must be considered when designing the programming language features to support inheritance and dynamic binding.

1) purely object-oriented or hybrid

- A language designer must choose whether to design the language using purely object-oriented feature or mix all other type of concepts.
- When dealing with pure object-orientation, everything from a simple scalar integer to a complete software system is an object.
- The advantage of this choice is pure uniformity of the language. But disadvantage is that even simple operations must be implemented using objects.

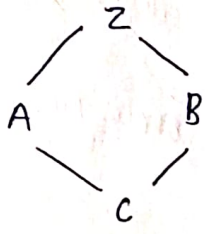
- Another alternative to the pure object-orientation is to implement simple operations using other programming paradigms and larger constructs using objects.
- This choice provides the speed of operations on primitive values. But this alternative also leads to complications in the implementation of the language.

2) Single and Multiple Inheritance

- This issue involves, whether language allows multiple inheritance in addition to simple inheritance.
- The reasons not to include multiple inheritance are :

a) Complexity :-

- If a class has two unrelated parents and this class inherits from both class A and class B a method named display. If C needs to reference both versions of display, how can it be done.
- Another issues arises, if A and B are derived from a common parent.



⇒ diamond inheritance.

- If 'C' inherits a method present in both A and B, which one should 'C' inherit, whether from A or B.

3) Allocation and deallocation of objects

- There are design issues concerning the allocation and deallocation of objects.
- The first is the place from which objects are allocated.
- This means they could be allocated from the run-time stack or from heap with an operator such as new.
- The second is if the allocation is from heap, how deallocation is done? implicit explicit.

4) Dynamic and Static Binding

- The question here is whether all binding of messages to methods is dynamic.
- The alternative is to allow the user to specify whether a specific binding is to be dynamic or static. Static bindings are faster.

5) Nested classes

- whether the language design supports nesting of classes.
- primary motivation for nesting class definition is for information hiding.
- The class in which the new class is nested is called the nesting class.
- what facilities of nesting class are visible to nested class

6) Initialization of objects

- The initialization is whether and how objects are initialized to values when they are created.
- The question is whether objects must be initialized manually or through some implicit mechanism.

Are Subclasses subtypes?

- Does an "is-a" relationship hold between a derived class and its parent c
- The derived class objects should be behaviorally equivalent to the parent class objects

Ex:- Subtypes of Ada - subtype Small_Int is Integer range -100 .. 100;

Implementation of object-oriented constructs

- There are at least two parts of language support for object-oriented programming

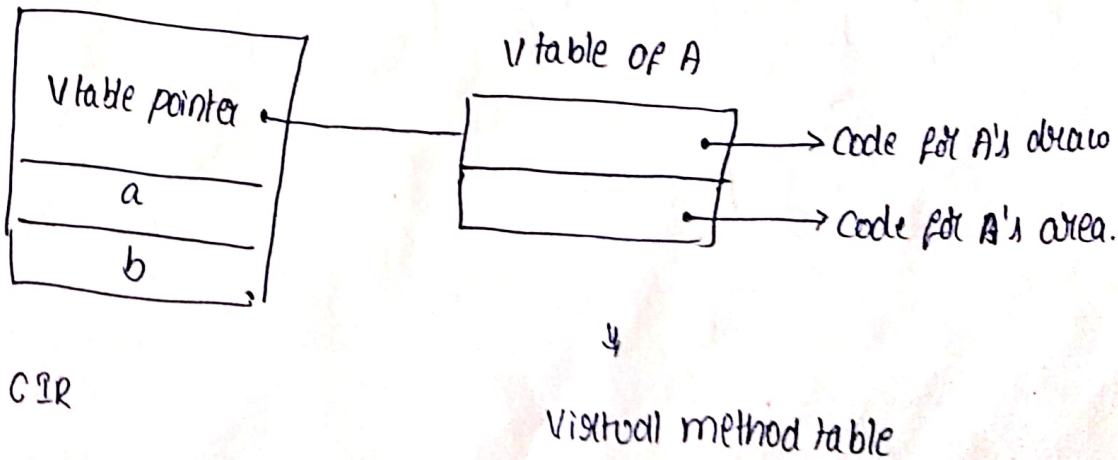
1) Storage structures for instance variables
2) Dynamic bindings of methods.

Instance data storage

- In C++, classes are defined as extensions of C's record structures.
- This similarity suggests a storage structure for the instance variables of a class. This form is called a class instance record (CIR)
- The structure of a CIR is static, so it is built at compile time.

Ex:

```
Public class A
{
  public int a,b;
  public void draw() { }
  public int area() { }
}
```



A's CIR

Virtual method table

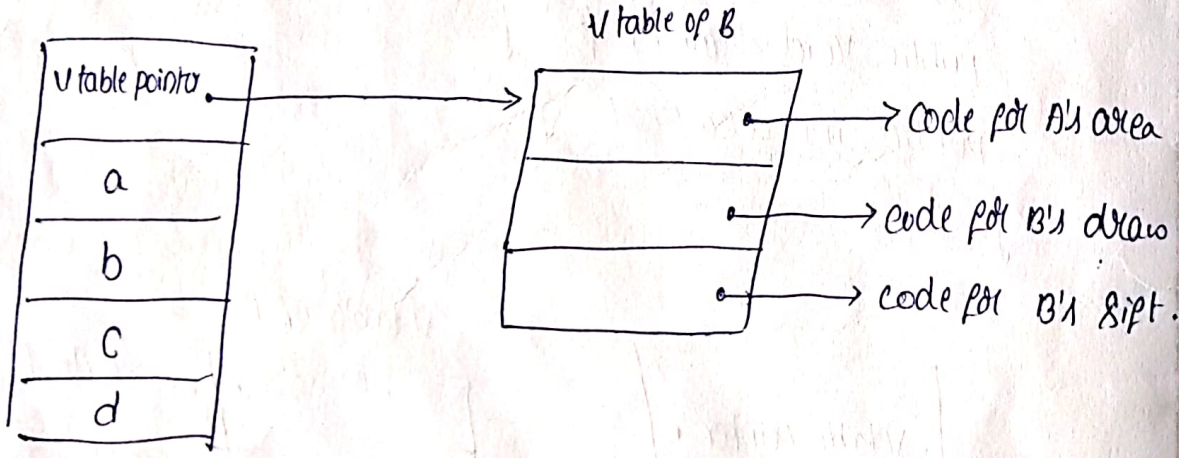
Dynamic binding of method calls to methods

- Methods in a class that are statically bound need not be involved in the CIR for the class.
- However methods that will be dynamically bound must have entries in this structure.

Simple inheritance

```
- public class B extends A
  {
    Public int c, d;
    public void draw () { }
    public void Sift() { }
  }
```

B's CIR

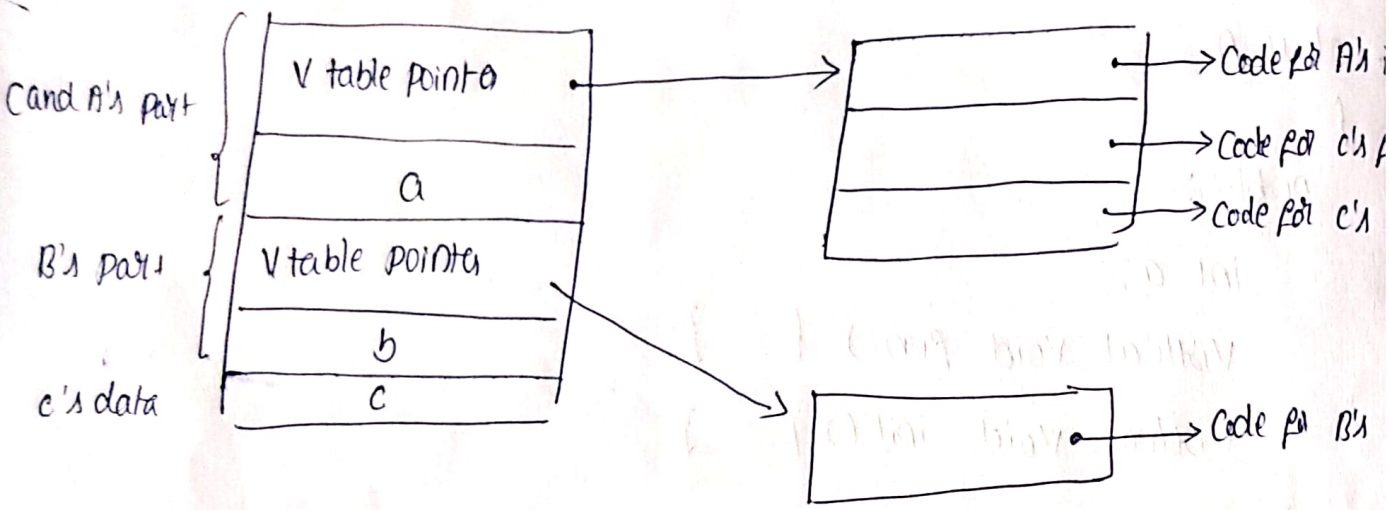


Multiple inheritance

```
- class A
{
  public:
     int a;
     virtual void fun() { }
     virtual void init() { }
 };

 class B
 {
  public:
     int b;
     virtual void sum() { }
  };
 class C : public A, public B
 {
   public:
     int c;
     virtual void fun() { }
     virtual void dud() { }
   };
```

CIR of C


C and A's part
V table pointer
a
B's part
V table pointer
b
c's data
c
Code for A's
Code for c's
Code for C's
Code for B's

Concurrency UNIT-4

- Concurrency refers to the ability of different parts or units of a program, algorithm or problem to be executed in out-of-order without affecting the final outcome.
- This allows parallel execution of the concurrent units, which significantly improves the overall speed of execution.
- Concurrency in software execution can occur at four different levels:
  1) instruction level (Execution of two or more machine instructions simultaneously)
  2) Statement level (Two or more high level language statements)
  3) Sub-program level (Two or more subprograms)
  4) program-level. (Two or more programs)

Categories of concurrency

There are two distinct categories of concurrent unit control. These

1) physical concurrency
2) Logical concurrency.

physical concurrency

In this type, there are more than one physical processors available and several program units (subprograms) from the same program literally execute simultaneously.

Logical concurrent

- In this there are no multiple physical processors, but the programmer and the software assume that there are multiple processors whose

in fact the actual execution of programs is taking place in an interleaved fashion on a single processor.

- From a programmer's and language designer's point of view, logical concurrency is same as physical concurrency.

Motivations for the use of concurrency

- There are various reasons to design concurrent software systems:

1) when a physical machine has multiple processors. The programs are designed to use this concurrent hardware and improve the speed of execution of programs. Effective utilization of hardware.

2) Even in a single processor system, concurrent execution of a program is faster than the same program written for sequential execution.

3) problem domains with natural concurrency, require concurrent software systems to implement them.

4) Applications that are distributed over several machines, either locally or through internet required concurrency

Subprogram-level concurrency

- A task (or process) is a unit of program, similar to a subprogram, that can be in concurrent execution with other units of the same program.

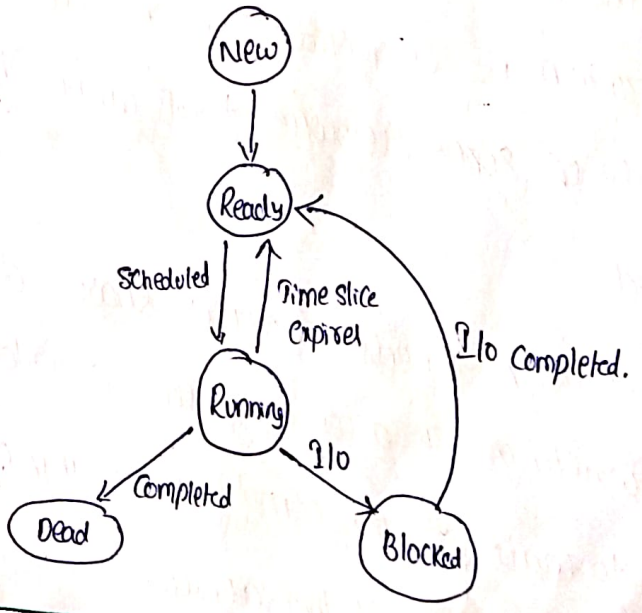
Each task in a program can support one thread of control.
Tasks usually work together
Tasks fall into two general categories:

1) heavyweight
2) Light weight.

- A heavyweight task has its own address space, while a light weight tasks all run in the same address space.
- A task is disjoint if it does not communicate or affect Execution of another task.
- Synchronization is a mechanism that controls the order in which tasks execute.

- Tasks can be in several different states:

1. New: A task is in the new state when it has been created but has not yet begun its execution.
2. Ready: A ready task is ready to run, but is not currently running. Either it has not been given processor time by the scheduler or it had run previously, but was blocked in one of any ways. like I/o.
3. Running :- A running task is one that is currently executing : that it has a processor.
4. Blocked : A task that is blocked, has been running, but that execution was interrupted by one of several different events, the most common is input or output operation.
5. Dead : A dead task is no longer active. A task dies when its execution is completed or killed by the program.



Design issues for concurrency

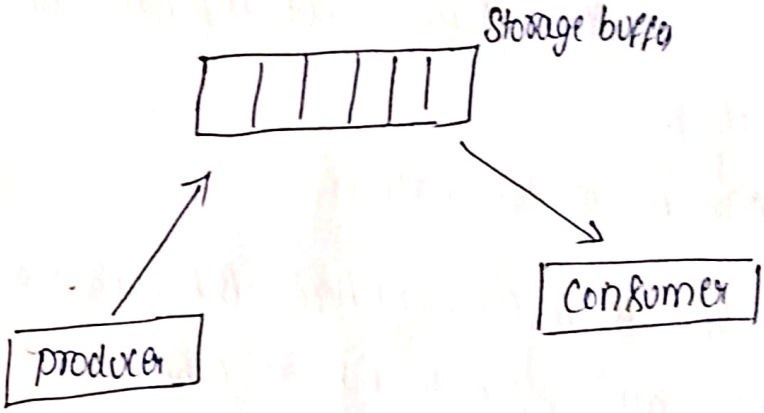
1) How an application can influence task scheduling.
2) How and when tasks start and end their executions and how and when they are created?
3) what are the Synchronization mechanisms supported?

Semaphores

- A semaphore is a simple mechanism that can be used to provide synchronization of tasks.
- Synchronization is a mechanism that controls the order in which tasks execute.
- Two kinds of synchronization are required when tasks share data:
    1) cooperation Synchronization
    2) competition Synchronization.
- cooperation synchronization is needed between task A and task B when task A must wait for task B to complete some specific activity before task A can begin or continue its execution.
  Ex:- producer-consumer probl
- competition synchronization is required between two tasks when both require the use of some resource that cannot be simultaneously used.
    - Specifically, if task A needs to access shared data location x, wh task B is also waiting for accessing same data location x, both the tasks may have to wait for the completion of any other processing on x, before either A or B gets the access.

Cooperation Synchronization

- This can be illustrated by a common problem called the producer-consumer problem.
- In this one program unit produces some data value or resource and another uses it. produced data are usually placed in a storage buffer by the producing unit and removed from the buffer by the consuming unit.



- The sequence of stores and removals from the buffer have to be synchronized.
- The consumer unit must not be allowed to take data from the buffer if the buffer is empty. In the same way, the producer unit cannot be allowed to place a new data in the buffer, if the buffer is full.

competition Synchronization

- It prevents two tasks from accessing a shared data structure at exactly the same time.
- To clarify the competition problem, consider a scenario:
  - Suppose a task A has the statement Total t=1
  - Task B has statement Total x=2.
  - Total is a shared integer variable

- At the machine language level, each task may accomplish its operation on Total with the following three-step procedure:

1) Fetch the value of Total.

2) Perform the arithmetic operation.

3) Put the new value back in Total.

- Without competition synchronization, operations performed by Task A and B on total, four different values could depending on the order of steps of operation.

- Assume Total has value 3.

1) If task A completes before B begins → 8

2) If task B completes before A begins → 7

3) If both A and B fetches the total values and modifies and if A completes and puts value back first → 6

4) if B completes and puts value back first → 4.

- A situation that leads to these problems is sometimes called a race condition.

- The behaviour of the program depends on which task arrives first.

- There are three alternative answers to design issues for concurrency:

1) Semaphores

2) Monitors

3) Message passing.

Semaphores

- A semaphore is a simple mechanism that can be used to provide synchronization of tasks.
- To provide limited access to a data structure, guards can be placed around the code that accesses the structure.
- A guard allows the guarded code to be executed only when a specified condition is true.
- So, a guard can be used to allow only one task to access a shared data structure at a time.
- Semaphores is an implementation of a guard.
- There are two operations provided for semaphores:
    1) wait
    2) Release.
- The wait semaphore subprogram is used to test the counter of a given semaphore variable. If the value is greater than zero, the caller can carry out its operation.
    - In this case, the counter value of the semaphore variable is decremented.
    - If the value of the counter is zero, the caller must be place on the waiting queue.
- The release semaphore subprogram is used by a task to allow some other task to have the resource.
    - The release increments its counter.

- There are two types of semaphores:

1) Binary semaphores.

2) Counting semaphores.

- The following are pseudocode descriptions of wait and release.

- wait (a semaphore)

  if 'a semaphore's counter > 0 then

    decrement a semphore's counter

  else

    put the a semaphore caller into the queue

  end if

- release (a semaphore)

  if 'a semaphore's queue is empty then

    increment a semphore's counter

  else

    ~~transfer~~ put the calling task in the task-ready queue.

  end if.

competition Synchronization

- The above operations can also be implemented to achieve competi Synchronization.

- In this semaphore variable indicate whether the buffer is currently being used or not.

- The wait statement allows the access only if the semaphore's counter has the value 1, which indicates that the shared buffer is not currently being accessed.

- If the semaphore's counter has a value of 0, there is a current access taking place and the task has to wait, so it is placed in queue of the semaphore.

- A semaphore that requires only a binary-valued counter (0,1) is called a binary semaphore.

Monitors

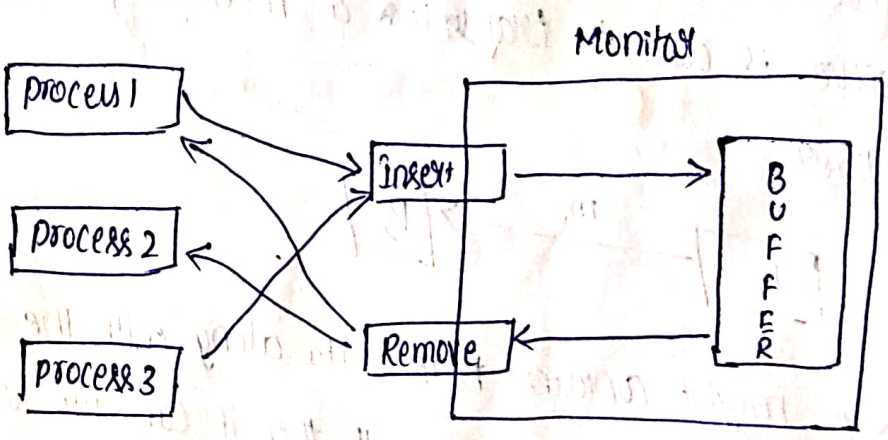
- The semaphore is an elegant synchronization tool for an ideal programmer who never makes mistakes: Never miss a wait or release

- In semaphores, operations on the shared data must not overlap. If a second operation begins earlier while the first operation is still in progress, the shared data can become corrupted. So, this problem is arises with semaphores.

- To solve this problem, we have to encapsulate shared data structures with their operations and hide their representations.

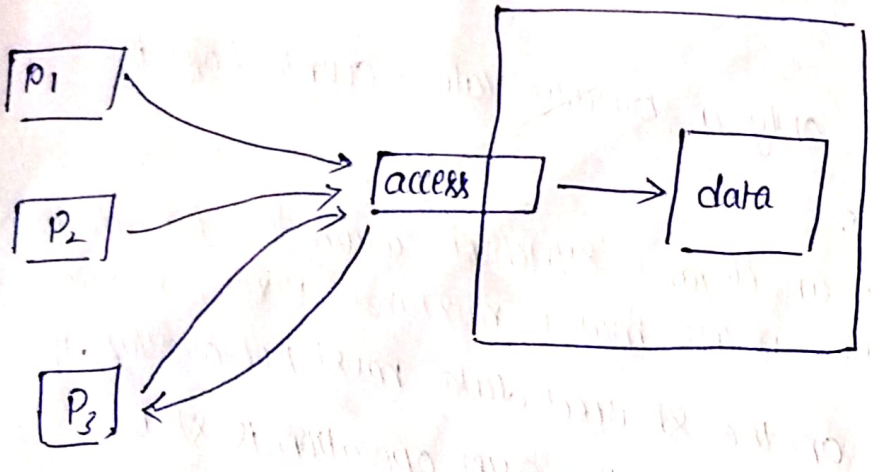
- Such a solution is monitors. Monitors appear exactly like an abstract data type with methods to access the shared data items.

Cooperation Synchronization



- The processes have to use the methods to access ~~and implement~~ the buffer and implement the cooperative synchronization.

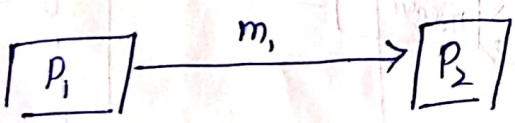
Competition Synchronization



- Monitors provide a better way to provide competition synchronization, where access will be provided to only one process among the many.

Message passing

- Message passing systems involve two primitive
  1) Send
  2) Recieve.
- The send primitive is used to transmit a message from one process to another



- Send message might contain ~~both~~ data along with the destination address. If the destination is default, then it can also transmit data.

Send (Data, Destination).

(or)

Send (Data)

- Recieve primitive, delivers the message to the destination process

Recieve (Data, destination address)

Recieve (Data).

Message passing can be of two types

1) Synchronous message passing

2) Asynchronous message passing.

- Synchronous message passing is otherwise involves blocking send where the sending process sends the message and gets blocked, untill the message is completely delivered at the destination.
- Asynchronous message passing is otherwise involves non-blocking send, where the sending process sends the message and continues with its work (not blocked).

Both cooperation and competition ~~execution~~ synchronization of tasks can be handled by message-passing model.

Java Threads

- The concurrent units in java are methods called "run" whose code can be run in concurrent execution with other such methods of other objects and with the main method.
- The process/task in which the run method executes is called a thread.

- Java's threads are lightweight tasks, i.e all the tasks will have the same address space.

Thread class

- To define a class with a run method, one can define a subclass of the predefined class Thread and override its run method.
- The base essentials of Thread are two methods named run and start.
- The code of the run method describes the actions of the thread. The start method starts its thread as a concurrent unit by calling its run method.
- when a program has multiple threads, a scheduler must determine which thread or threads will run at any given time.

priority of Threads

- Various threads can have varying priorities.
- A thread's default priority is same as the thread that created it.
- The priority of a thred can be changed with setPriority() method.
- The getPriority() method returns the current priority of a thread.
- The scheduler behaviour is controlled by the priority i.e execution of the threads will be performed based on the priorities.

Statement-level concurrency

- From the language design point of view, the objective of the design of statement-level concurrency is to provide a mechanism that the programmer can use to inform the compiler of the ways it can map the program onto a multiprocessor architecture.

High-performance Fortran (HPF)

- High performance Fortran is a collection of extensions to Fortran 90 that are meant to allow programmers to specify information to the compiler to help it optimize the execution of programs on multiprocessor computers.
- HPF includes statements and subprograms.

1) `! HPF $ PROCESSORS procs(n)`

- The above statement is used to specify the compiler the number of processors that can be used by the code generated for this program.

2) `! HPF $ DISTRIBUTE (BLOCK) ONTO procs :: identifier_list`

- The distribute statement specifies what data are to be distributed and the kind of distribution to be used.
- For example, if an array with 500 element named list is BLOCK distributed over 5 processors, the first 100 elements will be stored in the first processor, the second 100 in the second processor, so on.

3) FORALL (index = 1:1000)

list_1 (index) = list_2 (index)

END FORALL

- The FORALL statement specifies a sequence of assignment statements that may executed concurrently.

Exception Handling and Event Handling UNIT-4

- Most programming languages provide support for exception handling and event handling.
- Both exceptions and events occur at times that cannot be predetermined, and both are best handled with special language constructs and processes.

Exception handling

- Exception is any unusual event, erroneous or not, that is detectable by either hardware or software and that may require special processing.
- The special processing that may be required when an exception is detected is called exception handling.
- This processing is done by a code unit or segment called an exception handler.
- An exception is raised when its associated event occurs.
- Different exceptions require different exception handlers.
- Exception handling mechanisms are handled by either hardware or software.
- Hardware exception mechanisms are processed by CPU. It is intended to support error detection and redirects the program flow to error handling services of operating system.

Ex:- 1) Floating-point overflow/underflow

2) Integer overflow/underflow.

3) End of file error.

- There is a category of serious errors that are not detectable by hardware but can be detected by code generated by compiler.

Ex:- Array subscript range errors.

- programs may be notified when certain events occur and are detected by hardware or system software, so that they also can react to these events. These mechanisms are collectively called exception handling.

Handling of exceptions

- An exception detected within a program unit is often handled by the unit's caller or invoker.
- The various designs used to handle the exceptions are

1) ~~The exception handler is defined as a separate subprogram~~

i) Send an auxiliary parameter or use the return value to indicate the return status of a subprogram

Ex:- C standard library functions.

2) Pass a label parameter to subprograms

Ex:- FORTRAN.

3) Pass an exception handling subprogram to all subprograms.
- Have the handler defined as a separate subprogram; -Java.

Advantages of Exception handling

1) without exception handling, the code required to detect error conditions can make the program big and unmanagble.

2) Exception raised in one program unit to be handled in some other unit. This allows a single exception handler to be used for any number of different program units. This reuse can result in significant savings in development cost, program size and program complexity.

3) A language that supports exception handling encourages its users to consider all of the events that could occur during program execution and how can be handled.

- This is far better than not considering such possibilities and simply hoping nothing will go wrong.

4) programs with nonerroneous and unusual situations can be simplified with exception handling.

Design issues

The exception handling design issues can be summarized as follows:

1) How and where are exception handlers specified and what is their scope?
2) How is an exception occurance bound to an exception handler?
3) Can information about an exception be passed to the handler?
4) Where does execution continue, if at all, after an exception handler completes its execution? (Resumption)
5) Is some form of finalization provided?
6) How are user-defined exceptions specified?
7) If there are predefined ~~be explicitly~~ exceptions, should there be default exception handlers for programs that do not provide their own?
8) Can predefined exceptions be explicitly raised?
9) Are hardware-detectable errors treated as exceptions that may be handled?
10) Are there any predefined exceptions?
11) Should it be possible to disable predefined exceptions?

Event handling

- Event handling is similar to exception handling.
- In both cases, the handlers are implicitly called by the occurance of something either an exception or an event.
- While exceptions are created either explicitly by user code or implicitly by hardware or software, events are created by external actions, such as user interactions through a graphical user Interface (GUI).
- In event-driven programming, parts of the program are executed at completely unpredictable times, often triggered by user interactions with the executing program.
- An event is a notification that something specific has occured, such as a mouse click on a graphical button.
- An event handler is a segment of code that is executed in response to the appearance of an event. Event handlers enable a program to be responsive to user actions.
- Event handling is helpful in various situations:
  1) Many web browser documents presented to browser users are now dynamic. Such a documents may require internal computations associated with these button clicks are performed by event handlers that react to the click events.
  2) Another common use of event handlers is to check for simple errors and omissions in the elements of a form, either when they are changed or when the form is submitted to the webserver for processing.

Checked vs Unchecked Exceptions

Checked :- These are the exceptions that are checked at compile time. If some method code within the method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

Ex:- Java

Java Exceptions

- Exception handling in java is performed by using try, catch blocks;
- Try block contains set of statements where exception can occur.
- Catch block is where you handle the exceptions. A single try block can have several catch blocks associated with it.

Ex:-

```
class exp1
{
  P.S.V.m (String args[])
   {
     int num1, num2;
     try
      {
        num1=0;
        num2= 62 / num1;
        S.O.P.("Inside try");
      }
     catch (ArithmeticException e)
      {
        S.O.P("You should not divide by zero");
      }
```

```
        catch (Exception e)
        {
            // generic exception handler
            S.o.p(" Exception occurred");
        }
    S.o.p(" out of try-catch");
    }
}
```

unchecked

- unchecked exceptions are those that are not checked at compile time.
- In C++, all exceptions are unchecked, so it is up to the programmer to deal with exceptions.
- In Java, runtime exceptions are unchecked exceptions.

```
class main
{
 p.s.v.m( s a[])
 {
   int x=0;
   int y=10;
   int z=y/x;
 }
}
```