UNIT-4
Abstract Data Types

4.1 The Concept of Abstraction – CO3

- An *abstraction* is a view or representation of an entity that includes only the most significant attributes
- The concept of *abstraction* is fundamental in programming (and computer science)
- Nearly all programming languages support *process abstraction* with subprograms
- Nearly all programming languages designed since 1980 support *data abstraction*

4.2 Introduction to Data Abstraction – CO3

- An A*bstract Data Type* is a user-defined data type that satisfies the following two conditions:
  - The representation of, and operations on, objects of the type are defined in a single syntactic unit
  - 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

Advantages of Data Abstraction

- Advantage of the first condition
  - Program organization, modifiability (everything associated with a data structure is together), and separate compilation
- Advantage the second condition
  - Reliability--by hiding the data representations, user code cannot directly access objects of the type or depend on the representation, allowing the representation to be changed without affecting user code

Language Requirements for ADTs:

- A syntactic unit in which to encapsulate the type definition
- A method of making type names and subprogram headers visible to clients, while hiding actual definitions
- Some primitive operations must be built into the language processor

Design Issues:

- Can abstract types be parameterized?
- What access controls are provided?

4.3 Language Examples CO2, CO3, CO4

Language Examples: Ada

- The encapsulation construct is called a *package*
  - Specification package (the interface)
  - Body package (implementation of the entities named in the specification)
- Information Hiding
  - The spec package has two parts, public and private
  - The name of the abstract type appears in the public part of the specification package. This part may also include representations of unhidden types

- The representation of the abstract type appears in a part of the specification called the *private* part
- More restricted form with *limited private types*
  - Private types have built-in operations for assignment and comparison
  - Limited private types have NO built-in operations
- Reasons for the public/private spec package:
  1. The compiler must be able to see the representation after seeing only the spec package (it cannot see the private part)
  2. Clients must see the type name, but not the representation (they also cannot see the private part)
- Having part of the implementation details (the representation) in the spec package and part (the method bodies) in the body package is not good

One solution: make all ADTs pointers

Problems with this:

1. Difficulties with pointers
2. Object comparisons
3. Control of object allocation is lost

An Example in Ada

```
package Stack_Pack is
    type stack_type is limited private;
    max_size: constant := 100;
    function empty(stk: in stack_type) return Boolean;
    procedure push(stk: in out stack_type; elem:in Integer);
    procedure pop(stk: in out stack_type);
    function top(stk: in stack_type) return Integer;
    private -- hidden from clients
    type list_type is array (1..max_size) of Integer;
    type stack_type is record
    list: list_type;
    topsub: Integer range 0..max_size) := 0;
    end record;
end Stack_Pack
```

Language Examples: C++

- Based on C **struct** type and Simula 67 classes
- The class is the encapsulation device
- All of the class instances of a class share a single copy of the member functions
- Each instance of a class has its own copy of the class data members
- Instances can be static, stack dynamic, or heap dynamic
- Information Hiding
  - *Private* clause for hidden entities
  - *Public* clause for interface entities
  - *Protected* clause for inheritance (Chapter 12)
- Constructors:
  - Functions to initialize the data members of instances (they *do not* create the objects)
  - May also allocate storage if part of the object is heap-dynamic
  - Can include parameters to provide parameterization of the objects
  - Implicitly called when an instance is created
  - Can be explicitly called

- Name is the same as the class name
- Destructors
  - Functions to cleanup after an instance is destroyed; usually just to reclaim heap storage
  - Implicitly called when the object's lifetime ends
  - Can be explicitly called
  - Name is the class name, preceded by a tilde (~)

An Example in C++

```
class stack {
    private:
        int *stackPtr, maxLen, topPtr;
    public:
        stack() { // a constructor
            stackPtr = new int [100];
            maxLen = 99;
            topPtr = -1;
        };
        ~stack () {delete [] stackPtr;};
        void push (int num) {...};
        void pop () {...};
        int top () {...};
        int empty () {...};
}
```

Evaluation of ADTs in C++ and Ada

- C++ support for ADTs is similar to expressive power of Ada
- Both provide effective mechanisms for encapsulation and information hiding
- Ada packages are more general encapsulations; classes are types
- Friend functions or classes - to provide access to private members to some unrelated units or functions
  - Necessary in C++

Language Examples: Java

- Similar to C++, except:
  - All user-defined types are classes
  - All objects are allocated from the heap and accessed through reference variables
  - Individual entities in classes have access control modifiers (private or public), rather than clauses
  - Java has a second scoping mechanism, package scope, which can be used in place of friends
- All entities in all classes in a package that do not have access control modifiers are visible throughout the package

An Example in Java

```
class StackClass {
    private:
        private int [] *stackRef;
        private int [] maxLen, topIndex;
        public StackClass() { // a constructor
            stackRef = new int [100];
            maxLen = 99;
            topPtr = -1;
        };
        public void push (int num) {...};
        public void pop () {...};
        public int top () {...};
        public boolean empty () {...};
}
```

Language Examples: C#

- Based on C++ and Java
- Adds two access modifiers, *internal* and *protected internal*
- All class instances are heap dynamic
- Default constructors are available for all classes
- Garbage collection is used for most heap objects, so destructors are rarely used
- structs are lightweight classes that do not support inheritance
- Common solution to need for access to data members: accessor methods(getter and setter)
- C# provides *properties* as a way of implementing getters and setters without requiring explicit method calls

C# Property Example

```
public class Weather {
    public int DegreeDays { //** DegreeDays is a property
        get {return degreeDays;}
        set {
            if(value < 0 || value > 30)
                    Console.WriteLine("Value is out of range: {0}", value);
            else degreeDays = value;}
    }
    private int degreeDays;
    …
}
…
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
…
w.DegreeDays = degreeDaysToday;
…
oldDegreeDays = w.DegreeDays;
```

4.4 Parameterized Abstract Data Types - CO4

- Parameterized ADTs allow designing an ADT that can store any type elements (among other things)
- Also known as generic classes
- C++, Ada, Java 5.0, and C# 2005 provide support for parameterized ADTs

Parameterized ADTs in Ada

- Ada Generic Packages
  - Make the stack type more flexible by making the element type and the size of the stack generic

```
generic
Max_Size: Positive;
type Elem_Type is private;
package Generic_Stack is
Type Stack_Type is limited private;
function Top(Stk: in out StackType) return Elem_type;
…
end Generic_Stack;
Package Integer_Stack is new Generic_Stack(100,Integer);
Package Float_Stack is new Generic_Stack(100,Float);
```

Parameterized ADTs in C++

- Classes can be somewhat generic by writing parameterized constructor functions

```
class stack {
    …
```

```
        stack (int size) {
        stk_ptr = new int [size];
        max_len = size - 1;
        top = -1;
        };
        …
    }
    stack stk(100);
```

- The stack element type can be parameterized by making the class a templated class

```
template <class Type>
class stack {
    private:
        Type *stackPtr;
        const int maxLen;
        int topPtr;
    public:
        stack() {
            stackPtr = new Type[100];
            maxLen = 99;
            topPtr = -1;
        }
    …
}
```

Parameterized Classes in Java 5.0

- Generic parameters must be classes
- Most common generic types are the collection types, such as LinkedList and ArrayList
- Eliminate the need to cast objects that are removed
- Eliminate the problem of having multiple types in a structure

Parameterized Classes in C# 2005

- Similar to those of Java 5.0
- Elements of parameterized structures can be accessed through indexing

Summary of ADT

- The concept of ADTs and their use in program design was a milestone in the development of languages
- Two primary features of ADTs are the packaging of data with their associated operations and information hiding
- Ada provides packages that simulate ADTs
- C++ data abstraction is provided by classes
- Java's data abstraction is similar to C++
- Ada, C++, Java 5.0, and C# 2005 support parameterized ADTs

4.5 Object-Oriented Programming – CO4

- Abstract data types
- Inheritance
  - Inheritance is the central theme in OOP and languages that support it
- Polymorphism

Inheritance

- Productivity increases can come from reuse
  - ADTs are difficult to reuse—always need changes
  - All ADTs are independent and at the same level
- Inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts

- Inheritance addresses both of the above concerns--reuse ADTs after minor changes and define classes in a hierarchy

Object-Oriented Concepts

- ADTs are usually called *classes*
- Class instances are called *objects*
- A class that inherits is a *derived class* or a *subclass*
- The class from which another class inherits is a parent class or *superclass*
- Subprograms that define operations on objects are called *methods*
- Calls to methods are called *messages*
- The entire collection of methods of an object is called its *message protocol* or *message interface*
- Messages have two parts--a method name and the destination object
- In the simplest case, a class inherits all of the entities of its parent
- Inheritance can be complicated by access controls to encapsulated entities
  - A class can hide entities from its subclasses
  - A class can hide entities from its clients
  - A class can also hide entities for its clients while allowing its subclasses to see them
- Besides inheriting methods as is, a class can modify an inherited method
  - The new one *overrides* the inherited one
  - The method in the parent is *overriden*
- There are two kinds of variables in a class:
  - *Class variables* - one/class
  - *Instance variables* - one/object
- There are two kinds of methods in a class:
  - *Class methods* – accept messages to the class
  - *Instance methods* – accept messages to objects
- Single vs. Multiple Inheritance
- One disadvantage of inheritance for reuse:
  - Creates interdependencies among classes that complicate maintenance

Dynamic Binding

- A *polymorphic variable* can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants
- When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method will be dynamic
- Allows software systems to be more easily extended during both development and maintenance

Dynamic Binding Concepts

- An *abstract method* is one that does not include a definition (it only defines a protocol)
- An *abstract class* is one that includes at least one virtual method
- An abstract class cannot be instantiated

4.6 Design Issues for OOP Languages – CO4

- The Exclusivity of Objects
- Are Subclasses Subtypes?
- Type Checking and Polymorphism
- Single and Multiple Inheritance
- Object Allocation and DeAllocation

- Dynamic and Static Binding
- Nested Classes

The Exclusivity of Objects

- Everything is an object
  - Advantage - elegance and purity
  - Disadvantage - slow operations on simple objects
- Add objects to a complete typing system
  - Advantage - fast operations on simple objects
  - Disadvantage - results in a confusing type system (two kinds of entities)
- Include an imperative-style typing system for primitives but make everything else objects
  - Advantage - fast operations on simple objects and a relatively small typing system
  - Disadvantage - still some confusion because of the two type systems

Are Subclasses Subtypes?

- Does an "is-a" relationship hold between a parent class object and an object of the subclass?
  - If a derived class is-a parent class, then objects of the derived class must behave the same as the parent class object
- A derived class is a subtype if it has an is-a relationship with its parent class
- Subclass can only add variables and methods and override inherited methods in "compatible" ways

Type Checking and Polymorphism

- Polymorphism may require dynamic type checking of parameters and the return value
  - Dynamic type checking is costly and delays error detection
- If overriding methods are restricted to having the same parameter types and return type, the checking can be static

Single and Multiple Inheritance

- Multiple inheritance allows a new class to inherit from two or more classes
- Disadvantages of multiple inheritance:
  - Language and implementation complexity (in part due to name collisions)
  - Potential inefficiency - dynamic binding costs more with multiple inheritance (but not much)
- Advantage:
  - Sometimes it is quite convenient and valuable

Allocation and DeAllocation of Objects

- From where are objects allocated?
  - If they behave line the ADTs, they can be allocated from anywhere
- Allocated from the run-time stack
- Explicitly create on the heap (via new)
  - If they are all heap-dynamic, references can be uniform thru a pointer or reference variable
- Simplifies assignment - dereferencing can be implicit
  - If objects are stack dynamic, there is a problem with regard to subtypes
- Is deallocation explicit or implicit?

Dynamic and Static Binding

- Should all binding of messages to methods be dynamic?
  - If none are, you lose the advantages of dynamic binding
  - If all are, it is inefficient
- Allow the user to specify

Nested Classes

- If a new class is needed by only one class, there is no reason to define so it can be seen by other classes
  - Can the new class be nested inside the class that uses it?
  - In some cases, the new class is nested inside a subprogram rather than directly in another class
- Other issues:
  - Which facilities of the nesting class should be visible to the nested class and vice versa

4.6 Support for OOP in Smalltalk – CO4

- Smalltalk is a pure OOP language
  - Everything is an object
  - All objects have local memory
  - All computation is through objects sending messages to objects
  - None of the appearances of imperative languages
  - All objected are allocated from the heap
  - All deallocation is implicit
- Type Checking and Polymorphism
  - All binding of messages to methods is dynamic
- The process is to search the object to which the message is sent for the method; if not found, search the superclass, etc. up to the system class which has no superclass
  - The only type checking in Smalltalk is dynamic and the only type error occurs when a message is sent to an object that has no matching method
- Inheritance
  - A Smalltalk subclass inherits all of the instance variables, instance methods, and class methods of its superclass
  - All subclasses are subtypes (nothing can be hidden)
  - All inheritance is implementation inheritance
  - No multiple inheritance
- Evaluation of Smalltalk
  - The syntax of the language is simple and regular
  - Good example of power provided by a small language
  - Slow compared with conventional compiled imperative languages
  - Dynamic binding allows type errors to go undetected until run time
  - Introduced the graphical user interface
  - Greatest impact: advancement of OOP

4.7 Support for OOP in C++ - CO4

- General Characteristics:
  - Evolved from C and SIMULA 67
  - Among the most widely used OOP languages
  - Mixed typing system
  - Constructors and destructors
  - Elaborate access controls to class entities
- Inheritance
  - A class need not be the subclass of any class
  - Access controls for members are
  - Private (visible only in the class and friends) (disallows subclasses from being subtypes)

- Public (visible in subclasses and clients)
- Protected (visible in the class and in subclasses, but not clients)

- In addition, the subclassing process can be declared with access controls (private or public), which define potential changes in access by subclasses
  - Private derivation - inherited public and protected members are private in the subclasses
  - Public derivation public and protected members are also public and protected in subclasses

Inheritance Example in C++

```
class base_class {
    private:
        int a;
        float x;
    protected:
        int b;
        float y;
        public:
        int c;
        float z;
};
class subclass_1 : public base_class { ... };
// In this one, b and y are protected and
// c and z are public
class subclass_2 : private base_class { ... };
// In this one, b, y, c, and z are private,
// and no derived class has access to any
// member of base_class
```

- A member that is not accessible in a subclass (because of private derivation) can be declared to be visible there using the scope resolution operator (**::**), **e.g.**,

```
class subclass_3 : private base_class {
    base_class :: c;
    ...
}
```

- One motivation for using private derivation
  - A class provides members that must be visible, so they are defined to be public members; a derived class adds some new members, but does not want its clients to see the members of the parent class, even though they had to be public in the parent class definition
- Multiple inheritance is supported
  - If there are two inherited members with the same name, they can both be referenced using the scope resolution operator
- Dynamic Binding
  - A method can be defined to be virtual, which means that they can be called through polymorphic variables and dynamically bound to messages
  - A pure virtual function has no definition at all
  - A class that has at least one pure virtual function is an *abstract class*
- Evaluation
  - C++ provides extensive access controls (unlike Smalltalk)
  - C++ provides multiple inheritance
  - In C++, the programmer must decide at design time which methods will be statically bound and which must be dynamically bound
- Static binding is faster!
  - Smalltalk type checking is dynamic (flexible, but somewhat unsafe)
  - Because of interpretation and dynamic binding, Smalltalk is ~10 times slower than C++

4.8 Support for OOP in Java – CO4

- Because of its close relationship to C++, focus is on the differences from that language
- General Characteristics
  - All data are objects except the primitive types
  - All primitive types have wrapper classes that store one data value
  - All objects are heap-dynamic, are referenced through reference variables, and most are allocated with new
  - A finalize method is implicitly called when the garbage collector is about to reclaim the storage occupied by the object
- Inheritance
  - Single inheritance supported only, but there is an abstract class category that provides some of the benefits of multiple inheritance (interface)
  - An interface can include only method declarations and named constants, **e.g.**,

public interface Comparable {
public int comparedTo (Object b);}

  - Methods can be **final** (cannot be overriden)
- Dynamic Binding
  - In Java, all messages are dynamically bound to methods, unless the method is final (i.e., it cannot be overriden, therefore dynamic binding serves no purpose)
  - Static binding is also used if the methods is static or private both of which disallow overriding
- Several varieties of nested classes
- All are hidden from all classes in their package, except for the nesting class
- Nested classes can be anonymous
- A local nested class is defined in a method of its nesting class
  - No access specifier is used
- Evaluation
  - Design decisions to support OOP are similar to C++
  - No support for procedural programming
  - No parentless classes
  - Dynamic binding is used as "normal" way to bind method calls to method definitions
  - Uses interfaces to provide a simple form of support for multiple inheritance

4.9 Support for OOP in C# -CO4

- General characteristics
  - Support for OOP similar to Java
  - Includes both classes and structs
  - Classes are similar to Java's classes
  - structs are less powerful stack-dynamic constructs (**e.g.**, no inheritance)
- Inheritance
  - Uses the syntax of C++ for defining classes
  - A method inherited from parent class can be replaced in the derived class by marking its definition with new
  - The parent class version can still be called explicitly with the prefix base: base.Draw()
- Dynamic binding
  - To allow dynamic binding of method calls to methods:

- The base class method is marked virtual
- The corresponding methods in derived classes are marked override
  - Abstract methods are marked abstract and must be implemented in all subclasses
  - All C# classes are ultimately derived from a single root class, Object
- Nested Classes
  - A C# class that is directly nested in a nesting class behaves like a Java static nested class
  - C# does not support nested classes that behave like the non-static classes of Java
- Evaluation
  - C# is the most recently designed C-based OO language
  - The differences between C#'s and Java's support for OOP are relatively minor

4.10 Support for OOP in Ada 95 – CO4

- General Characteristics
  - OOP was one of the most important extensions to Ada 83
  - Encapsulation container is a package that defines a *tagged type*
  - A tagged type is one in which every object includes a tag to indicate during execution its type (the tags are internal)
  - Tagged types can be either private types or records
  - No constructors or destructors are implicitly called
- Inheritance
  - Subclasses can be derived from tagged types
  - New entities are added to the inherited entities by placing them in a record definition
  - All subclasses are subtypes
  - No support for multiple inheritance
- A comparable effect can be achieved using generic classes

Example of a Tagged Type

```
Package Person_Pkg is
type Person is tagged private;
procedure Display(P : in out Person);
    private
    type Person is tagged
        record
            Name : String(1..30);
            Address : String(1..30);
            Age : Integer;
        end record;
    end Person_Pkg;
    with Person_Pkg; use Person_Pkg;
    package Student_Pkg is
    type Student is new Person with
    record
        Grade_Point_Average : Float;
        Grade_Level : Integer;
    end record;
procedure Display (St: in Student);
end Student_Pkg;
// Note: Display is being overridden from Person_Pkg
```

- Dynamic Binding
  - Dynamic binding is done using polymorphic variables called classwide types
- For the tagged type **Prtdon**, the classwide type is **Person' class**
  - Other bindings are static

  - Any method may be dynamically bound
  - Purely abstract base types can be defined in Ada 95 by including the reserved word abstract
- Evaluation
  - Ada offers complete support for OOP
  - C++ offers better form of inheritance than Ada
  - Ada includes no initialization of objects (**e.g.**, constructors)
  - Dynamic binding in C-based OOP languages is restricted to pointers and/or references to objects; Ada has no such restriction and is thus more orthogonal

Implementing OOPs Constructs

- Two interesting and challenging parts
  - Storage structures for instance variables
  - Dynamic binding of messages to methods

Instance Data Storage

- Class instance records (CIRs) store the state of an object
  - Static (built at compile time)
- If a class has a parent, the subclass instance variables are added to the parent CIR
- Because CIR is static, access to all instance variables is done as it is in records
  - Efficient

Dynamic Binding of Methods Calls

- Methods in a class that are statically bound need not be involved in the CIR; methods that will be dynamically bound must have entries in the CIR
  - Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR
  - The storage structure is sometimes called *virtual method tables* (vtable)
  - Method calls can be represented as offsets from the beginning of the vtable

Summary of OOPs

- OO programming involves three fundamental concepts: ADTs, inheritance, dynamic binding
- Major design issues: exclusivity of objects, subclasses and subtypes, type checking and polymorphism, single and multiple inheritance, dynamic binding, explicit and implicit de-allocation of objects, and nested classes
- Smalltalk is a pure OOL
- C++ has two distinct type system (hybrid)
- Java is not a hybrid language like C++; it supports only OO programming
- C# is based on C++ and Java
- Implementing OOP involves some new data structures

Concurrency

- 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

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

Categories of Concurrency

- 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 (*quasi-concurrency) have a single thread of control*

Motivations for Studying Concurrency

- Involves a different way of designing software that can be very useful— many real-world situations involve concurrency
- Multiprocessor computers capable of physical concurrency are now widely used

Subprogram-Level Concurrency

- A *task* or *process* 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
- Tasks usually work together

Two General Categories of Tasks

- *Heavyweight tasks* execute in their own address space
- *Lightweight tasks* all run in the same address space
- 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 Synchronization

- A mechanism that controls the order in which tasks execute
- Two kinds of synchronization
  - *Cooperation* synchronization
  - *Competition* synchronization
- Task communication is necessary for synchronization, provided by:
  - 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*: Two or more tasks must use some resource that cannot be simultaneously used, **e.g.**, a shared counter
  - Competition is usually provided by mutually exclusive access (approaches are discussed later)
  -

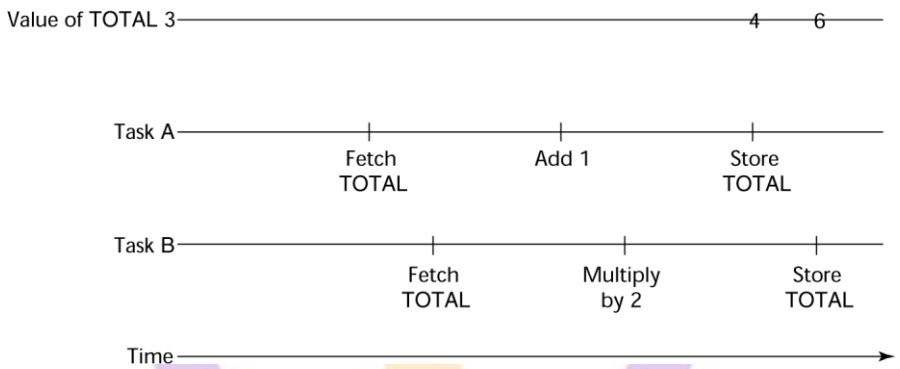

Figure 6.1 Need for Competition Synchronization

Scheduler

- 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

Task Execution States

- *New* - created but not yet started
- *R*eady - ready to run but not currently running (no available processor)
- *Running*
- *Blocked* - has been running, but cannot now continue (usually waiting for some event to occur)
- *Dead* - no longer active in any sense

Liveness and Deadlock

- *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

- Competition and cooperation synchronization
- Controlling task scheduling
- How and when tasks start and end execution
- How and when are tasks created

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: emptyspots and fullspots
- The semaphore counters are used to store the numbers of empty spots and full spots in the buffer
- DEPOSIT must first check emptyspots to see if there is room in the buffer
- If there is room, the counter of emptyspots is decremented and the value is inserted
- If there is no room, the caller is stored in the queue of emptyspots
- When DEPOSIT is finished, it must increment the counter of fullspots
- FETCH must first check fullspots to see if there is a value
  - If there is a full spot, the counter of fullspots is decremented and the value is removed
  - If there are no values in the buffer, the caller must be placed in the queue of fullspots
  - When FETCH is finished, it increments the counter of emptyspots
- The operations of FETCH and DEPOSIT on the semaphores are accomplished through two semaphore operations named *wait* and *release*

Semaphores: Wait Operation

```
wait(aSemaphore)
    if aSemaphore's counter > 0 then
            decrement aSemaphore's counter
    else
            put the caller in aSemaphore's queue
            attempt to transfer control to a ready task
    -- if the task ready queue is empty,
    -- deadlock occurs
end
```

Semaphores: Release Operation

```
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
```

Producer Consumer Code

```
semaphore fullspots, emptyspots;
        fullstops.count = 0;
        emptyspots.count = BUFLEN;
        task producer;
        loop
                -- produce VALUE --
                wait (emptyspots); {wait for space}
                DEPOSIT(VALUE);
                release(fullspots); {increase filled}
        end loop;
end producer;
```

Producer Consumer Code

```
task consumer;
    loop
        wait (fullspots);{wait till not empty}}
        FETCH(VALUE);
        release(emptyspots); {increase empty}
        -- consume VALUE --
    end loop;
end consumer;
```

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*
- Note that wait and release must be atomic!

Producer Consumer Code

```
semaphore access, fullspots, emptyspots;
    access.count = 0;
    fullstops.count = 0;
    emptyspots.count = BUFLEN;
    task producer;
    loop
        -- produce VALUE --
        wait(emptyspots); {wait for space}
        wait(access); {wait for access)
        DEPOSIT(VALUE);
        release(access); {relinquish access}
        release(fullspots); {increase filled}
    end loop;
end producer;
```

Producer Consumer Code

```
task consumer;
    loop
        wait(fullspots);{wait till not empty}
        wait(access); {wait for access}
        FETCH(VALUE);
        release(access); {relinquish access}
        release(emptyspots); {increase empty}
        -- consume VALUE --
    end loop;
end consumer;
```

Evaluation of Semaphores

- Misuse of semaphores can cause failures in cooperation synchronization, **e.g.**, the buffer will overflow if the wait of fullspots 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

- Ada, Java, C#
- The idea: encapsulate the shared data and its operations to restrict access
- A monitor is an abstract data type for shared data

Competition Synchronization

- Shared data is resident in the monitor (rather than in the client units)
- All access resident in the monitor

- Monitor implementation guarantee synchronized access by allowing only one access at a time
- Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call

Cooperation Synchronization

- Cooperation between processes is still a programming task
  - Programmer must guarantee that a shared buffer does not experience underflow or overflow

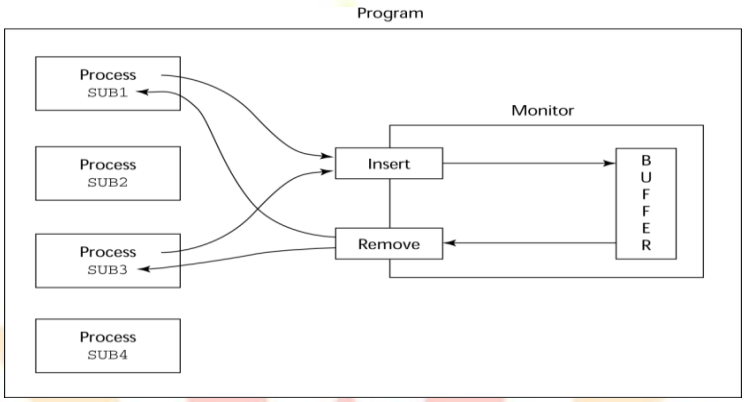

Figure 6.2 Cooperation Synchronization

Evaluation of Monitors

- A better way to provide competition synchronization than are semaphores
- Semaphores can be used to implement monitors
- Monitors can be used to implement semaphores
- Support for cooperation synchronization is very similar as with semaphores, so it has the same problems

Message Passing

- Message passing is a general model for concurrency
  - It can model both semaphores and monitors
  - It is not just for competition synchronization
- Central idea: task communication is like seeing a doctor--most of the time she waits for you or you wait for her, but when you are both ready, you get together, or *rendezvous*

Message Passing Rendezvous

- To support concurrent tasks with message passing, a language needs:
  - A mechanism to allow a task to indicate when it is willing to accept messages
  - A way to remember who is waiting to have its message accepted and some "fair" way of choosing the next message
- When a sender task's message is accepted by a receiver task, the actual message transmission is called a *rendezvous*

Ada Support for Concurrency

- The Ada 83 Message-Passing Model
  - Ada tasks have specification and body parts, like packages; the spec has the interface, which is the collection of entry points:

```
task Task_Example is
        entry ENTRY_1 (Item : in Integer);
end Task_Example;
```

Task Body

- The body task describes the action that takes place when a rendezvous occurs
- A task that sends a message is suspended while waiting for the message to be accepted and during the rendezvous
- Entry points in the spec are described with accept clauses in the body accept

```
entry_name (formal parameters) do
        ...
end entry_name
```

Example of a Task Body

```
task body Task_Example is
begin
        loop
                accept Entry_1 (Item: in Float) do
                ...
                end Entry_1;
        end loop;
end Task_Example;
```

Ada Message Passing Semantics

- The task executes to the top of the accept clause and waits for a message
- During execution of the accept clause, the sender is suspended
- accept parameters can transmit information in either or both directions
- Every accept clause has an associated queue to store waiting messages

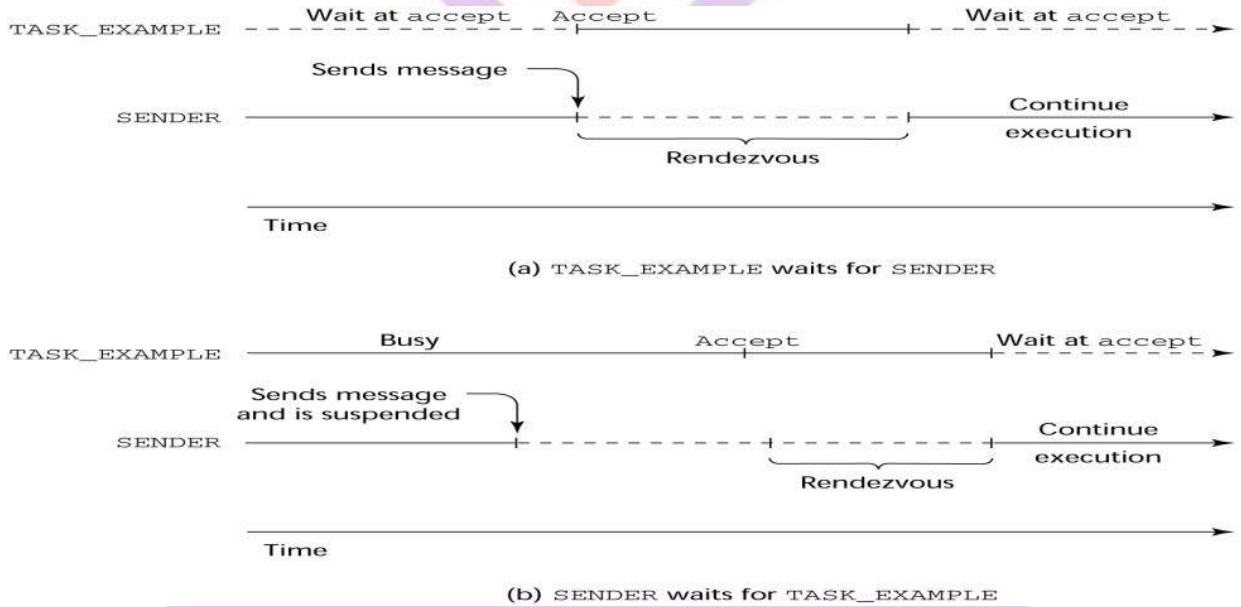

Figure 6.3 Rendezvous Time Lines

Message Passing: Server/Actor Tasks

- A task that has accept clauses, but no other code is called a *server task* (the example above is a server task)
- A task without accept clauses is called an *actor task*
  - An actor task can send messages to other tasks
  - Note: A sender must know the entry name of the receiver, but not vice versa (asymmetric)

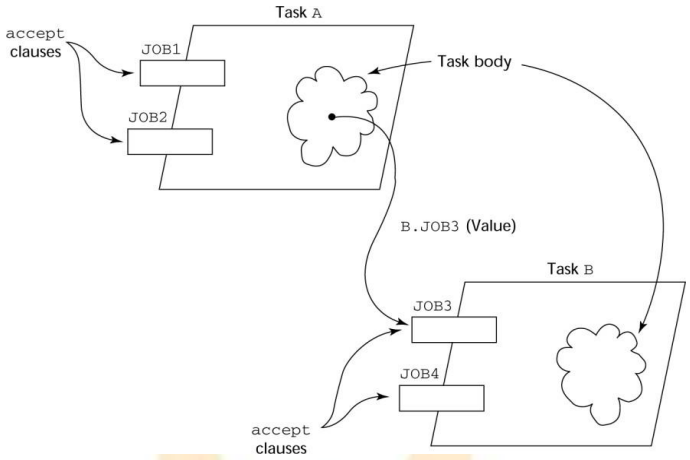

Figure 6.4 Graphical Representation of a Rendezvous

Example: Actor Task

```
task Water_Monitor; -- specificationtask body body Water_Monitor is -- body
begin
        loop
                if Water_Level > Max_Level
                        then Sound_Alarm;
                end if;
                        delay 1.0; -- No further execution
                        -- for at least 1 second
        end loop;
end Water_Monitor;
```

Multiple Entry Points

- Tasks can have more than one **entry** point
  - The specification task has an entry clause for each
  - The task body has an accept clause for each entry clause, placed in a select clause, which is in a loop

A Task with Multiple Entries

```
task body Teller is
        loop
                select
                        accept Drive_Up(formal params) do
                        ...
                        end Drive_Up;
                        ...
                        or
                        accept Walk_Up(formal params) do
                        ...
                        end Walk_Up;
                        ...
                end select;
        end loop;
end Teller;
```

Semantics of Tasks with Multiple accept Clauses

- If exactly one entry queue is nonempty, choose a message from it
- If more than one entry queue is nonempty, choose one, nondeterministically, from which to accept a message
- If all are empty, wait
- The construct is often called a selective wait
- Extended accept clause - code following the clause, but before the next clause
  - Executed concurrently with the caller

Cooperation Synchronization with Message Passing

- Provided by Guarded **accept** clauses
  when not Full(Buffer) =>
  accept Deposit (New_Value) do
- An accept clause with a with a when clause is either *open* or *closed*
  - A clause whose guard is true is called *open*
  - A clause whose guard is false is called *closed*
  - A clause without a guard is always open

Semantics of select with Guarded accept Clauses:

- select first checks the guards on all clauses
- If exactly one is open, its queue is checked for messages
- If more than one are open, non-deterministically choose a queue among them to check for messages
- If all are closed, it is a runtime error
- A select clause can include an else clause to avoid the error
  - When the else clause completes, the loop repeats

Example of a Task with Guarded accept Clauses

- Note: The station may be out of gas and there may or may not be a position available in the garage

```
task Gas_Station_Attendant is
        entry Service_Island (Car : Car_Type);
        entry Garage (Car : Car_Type);
end Gas_Station_Attendant;
```

Example of a Task with Guarded accept Clauses

```
task body Gas_Station_Attendant is
begin
        loop
                select
                        when Gas_Available =>
                                accept Service_Island (Car : Car_Type) do
                                Fill_With_Gas (Car);
                        end Service_Island;
                        or
                        when Garage_Available =>
                                accept Garage (Car : Car_Type) do
                                Fix (Car);
                        end Garage;
                        else
                                Sleep;
                end select;
        end loop;
end Gas_Station_Attendant;
```

Competition Synchronization with Message Passing

- Modeling mutually exclusive access to shared data
- Example--a shared buffer
- Encapsulate the buffer and its operations in a task
- Competition synchronization is implicit in the semantics of accept clauses
  - Only one accept clause in a task can be active at any given time

Task Termination

- The execution of a task is *completed* if control has reached the end of its code body
- If a task has created no dependent tasks and is completed, it is *terminated*
- If a task has created dependent tasks and is completed, it is not terminated until all its dependent tasks are terminated

The terminate Clause

- A terminate clause in a select is just a terminate statement
- A terminate clause is selected when no accept clause is open
- When a terminate is selected in a task, the task is terminated only when its master and all of the dependents of its master are either completed or are waiting at a terminate
- A block or subprogram is not left until all of its dependent tasks are terminated

Message Passing Priorities

- The priority of any task can be set with the pragma **priority** pragma Priority (expression);
- The priority of a task applies to it only when it is in the task ready queue

Binary Semaphores

- For situations where the data to which access is to be controlled is NOT encapsulated in a task

```
task Binary_Semaphore is
        entry Wait;
        entry release;
end Binary_Semaphore;
task body Binary_Semaphore is
begin
        loop
        accept Wait;
        accept Release;
        end loop;
end Binary_Semaphore;
```

Concurrency in Ada 95

- Ada 95 includes Ada 83 features for concurrency, plus two new features
  - Protected objects: A more efficient way of implementing shared data to allow access to a shared data structure to be done without rendezvous
  - Asynchronous communication

Ada 95: Protected Objects

- A *protected object* is similar to an abstract data type
- Access to a protected object is either through messages passed to entries, as with a task, or through protected subprograms
- A protected procedure provides mutually exclusive read-write access to protected objects
- A protected function provides concurrent read-only access to protected objects

Asynchronous Communication

- Provided through asynchronous select structures
- An asynchronous select has two triggering alternatives, an entry clause or a delay
  - The entry clause is triggered when sent a message
  - The delay clause is triggered when its time limit is reached

Evaluation of the Ada

- Message passing model of concurrency is powerful and general
- Protected objects are a better way to provide synchronized shared data
- In the absence of distributed processors, the choice between monitors and tasks with message passing is somewhat a matter of taste
- For distributed systems, message passing is a better model for concurrency

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

Competition Synchronization with Java Threads

- A method that includes the synchronized modifier disallows any other method from running on the object while it is in execution

```
...
public synchronized void deposit( int i) {...}
public synchronized int fetch() {...}
...
```

- The above two methods are synchronized which prevents them from interfering with each other
- If only a part of a method must be run without interference, it can be synchronized through synchronized statement

```
synchronized (expression)
statement
```

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 sophisticated synchronization

C#'s Concurrency Evaluation

- An advance over Java threads, **e.g.**, any method can run its own thread
- Thread termination is cleaner than in Java
- Synchronization is more sophisticated

Statement-Level Concurrency

- Objective: Provide a mechanism that the programmer can use to inform compiler of ways it can map the program onto multiprocessor architecture
- Minimize communication among processors and the memories of the other processors

High-Performance Fortran

- A collection of extensions that allow the programmer to provide information to the compiler to help it optimize code for multiprocessor computers
- Specify the number of processors, the distribution of data over the memories of those processors, and the alignment of data

Primary HPF Specifications

- Number of processors

  !HPF$ PROCESSORS procs (n)
- Distribution of data

  !HPF$ DISTRIBUTE (kind) ONTO procs :: identifier_list

  – kind can be BLOCK (distribute data to processors in blocks) or CYCLIC (distribute data to processors one element at a time)

- Relate the distribution of one array with that of another

  ALIGN *array1_element* WITH *array2_element*

Statement-Level Concurrency Example

```
REAL list_1(1000), list_2(1000)
INTEGER list_3(500), list_4(501)
!HPF$ PROCESSORS proc (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs ::
list_1, list_2
!HPF$ ALIGN list_1(index) WITH
list_4 (index+1)
...
list_1 (index) = list_2(index)
list_3(index) = list_4(index+1)
```

- FORALL statement is used to specify a list of statements that may be executed concurrently

```
FORALL (index = 1:1000)
list_1(index) = list_2(index)
```

- Specifies that all 1,000 RHSs of the assignments can be evaluated before any assignment takes place

Summary

- Concurrent execution can be at the instruction, statement, or subprogram level
- Physical concurrency: when multiple processors are used to execute concurrent units
- Logical concurrency: concurrent united are executed on a single processor
- Two primary facilities to support subprogram concurrency: competition synchronization and cooperation synchronization
- Mechanisms: semaphores, monitors, rendezvous, threads
- High-Performance Fortran provides statements for specifying how data is to be distributed over the memory units connected to multiple processors

Exception Handling & Logic Programming Language

Introduction to 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

4.11 Basic Concepts – CO3

- Many languages allow programs to trap input/output errors (including EOF)
- An *exception* is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing
- The special processing that may be required after detection of an exception is called *exception handling*
- The exception handling code unit is called an *exception handler*

Exception Handling Alternatives

- 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 (user defined, software detected)
- Alternatives:
  - Send an auxiliary parameter or use the return value to indicate the return status of a subprogram
  - 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 handling encourages programmers to consider many different possible errors
- Exception propagation allows a high level of reuse of exception handling code

Design Issues

- 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?
- How and where exception handlers specified and what are their scope?
- How is an exception occurrence bound to an exception handler?
- Can information about the exception be passed to the handler?
- Where does execution continue, if at all, after an exception handler completes its execution? (continuation vs. resumption)
- Is some form of finalization provided?

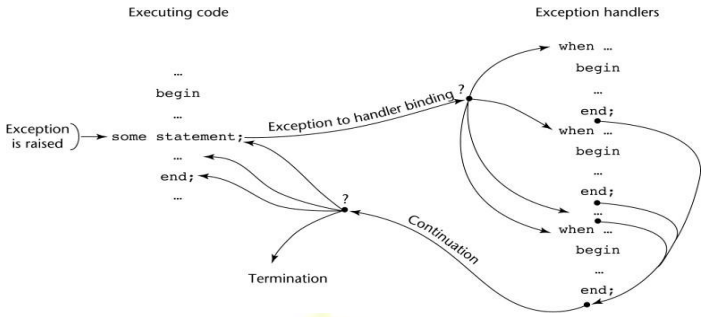

Figure 7.1 Exception Handling Control Flow

4.12 Exception Handling in Ada – CO3

- The frame of an exception handler in Ada is either a subprogram body, a package body, a task, or a block
- Because exception handlers are usually local to the code in which the exception can be raised, they do not have parameters

Ada Exception Handlers

- Handler form:

  when exception_choice{|exception_choice} => statement_sequence
  ...
  [when others =>
  statement_sequence]
  exception_choice form:
  exception_name | others

- Handlers are placed at the end of the block or unit in which they occur

Binding Exceptions to Handlers

- If the block or unit in which an exception is raised does not have a handler for that exception, the exception is propagated elsewhere to be handled
  - Procedures - propagate it to the caller
  - Blocks - propagate it to the scope in which it appears
  - Package body - propagate it to the declaration part of the unit that declared the package (if it is a library unit, the program is terminated)
  - Task - no propagation; if it has a handler, execute it; in either case, mark it "completed"

Continuation

- The block or unit that raises an exception but does not handle it is always terminated (also any block or unit to which it is propagated that does not handle it)

Other Design Choices

- User-defined Exceptions form:
  exception_name_list : exception;
- Raising Exceptions form:
  raise [exception_name]
  - (the exception name is not required if it is in a handler--in this case, it propagates the same exception)
- Exception conditions can be disabled with:
  pragma SUPPRESS(exception_list)

Predefined Exceptions

- CONSTRAINT_ERROR - index constraints, range constraints, etc.
- NUMERIC_ERROR - numeric operation cannot return a correct value (overflow, division by zero, etc.)

- PROGRAM_ERROR - call to a subprogram whose body has not been elaborated
- STORAGE_ERROR - system runs out of heap
- TASKING_ERROR - an error associated with tasks

Evaluation

- The Ada design for exception handling embodies the state-of-the-art in language design in 1980
- A significant advance over PL/I
- Ada was the only widely used language with exception handling until it was added to C++

4.13 Exception Handling in C++ - CO3

- Added to C++ in 1990
- Design is based on that of CLU, Ada, and ML

C++ Exception Handlers

- Exception Handlers Form:

```
try {
-- code that is expected to raise an exception
}
catch (formal parameter) {
-- handler code
}
...
catch (formal parameter) {
-- handler code
}
```

The catch Function

- catch is the name of all handlers--it is an overloaded name, so the formal parameter of each must be unique
- The formal parameter need not have a variable
  - It can be simply a type name to distinguish the handler it is in from others
- The formal parameter can be used to transfer information to the handler
- The formal parameter can be an ellipsis, in which case it handles all exceptions not yet handled

Throwing Exceptions

- Exceptions are all raised explicitly by the statement: *throw [expression];*
- The brackets are metasymbols
- A throw without an operand can only appear in a handler; when it appears, it simply re-raises the exception, which is then handled elsewhere
- The type of the expression disambiguates the intended handler

Unhandled Exceptions

- An unhandled exception is propagated to the caller of the function in which it is raised
- This propagation continues to the main function

Continuation

- After a handler completes its execution, control flows to the first statement after the last handler in the sequence of handlers of which it is an element
- Other design choices

- All exceptions are user-defined
- Exceptions are neither specified nor declared
- Functions can list the exceptions they may raise
- Without a specification, a function can raise any exception (the throw clause)

Evaluation

- It is odd that exceptions are not named and that hardware- and system software-detectable exceptions cannot be handled
- Binding exceptions to handlers through the type of the parameter certainly does not promote readability

4.13 Exception Handling in Java – CO3

- Based on that of C++, but more in line with OOP philosophy
- All exceptions are objects of classes that are descendants of the Throwable class

Classes of Exceptions

- The Java library includes two subclasses of Throwable :
  - Error
    - Thrown by the Java interpreter for events such as heap overflow
    - Never handled by user programs
  - Exception
    - User-defined exceptions are usually subclasses of this
    - Has two predefined subclasses, IOException and RuntimeException
      e.g., ArrayIndexOutOfBoundsException and NullPointerException

Java Exception Handlers

- Like those of C++, except every catch requires a named parameter and all parameters must be descendants of Throwable
- Syntax of try clause is exactly that of C++
- Exceptions are thrown with throw, as in C++, but often the throw includes the new operator to create the object, as in: *throw new MyException();*

Binding Exceptions to Handlers

- Binding an exception to a handler is simpler in Java than it is in C++
  - An exception is bound to the first handler with a parameter is the same class as the thrown object or an ancestor of it
- An exception can be handled and rethrown by including a throw in the handler (a handler could also throw a different exception)

Continuation

- If no handler is found in the method, the exception is propagated to the method's caller
- If no handler is found (all the way to main), the program is terminated
- To ensure that all exceptions are caught, a handler can be included in any try construct that catches all exceptions
  - Simply use an Exception class parameter
  - Of course, it must be the last in the try construct

Checked and Unchecked Exceptions

- The Java throws clause is quite different from the throw clause of C++
- Exceptions of class Error and RunTimeException and all of their descendants are called unchecked exceptions; all other exceptions are called checked exceptions
- Checked exceptions that may be thrown by a method must be either:
  - Listed in the throws clause, or
  - Handled in the method

Other Design Choices

- A method cannot declare more exceptions in its throws clause than the method it overrides
- A method that calls a method that lists a particular checked exception in its throws clause has three alternatives for dealing with that exception:
  - Catch and handle the exception
  - Catch the exception and throw an exception that is listed in its own throws clause
  - Declare it in its throws clause and do not handle it

The finally Clause

- Can appear at the end of a try construct
- Form:

  finally {..}
- Purpose: To specify code that is to be executed, regardless of what happens in the try construct

Example

- A try construct with a finally clause can be used outside exception handling

```
try {
for (index = 0; index < 100; index++) {
...
if (...) {
return;
} //** end of if
} //** end of try clause
finally {
...
} //** end of try construct
```

Assertions

- Statements in the program declaring a boolean expression regarding the current state of the computation
- When evaluated to true nothing happens
- When evaluated to false an AssertionError exception is thrown
- Can be disabled during runtime without program modification or recompilation
- Two forms
  - assert *condition*;
  - assert *condition*: *expression*;

Evaluation

- The types of exceptions makes more sense than in the case of C++
- The throws clause is better than that of C++ (The throw clause in C++ says little to the programmer)
- The finally clause is often useful
- The Java interpreter throws a variety of exceptions that can be handled by user programs

Summary of Exception Handling

- Ada provides extensive exception-handling facilities with a comprehensive set of built-in exceptions.
- C++ includes no predefined exceptions. Exceptions are bound to handlers by connecting the type of expression in the throw statement to that of the formal parameter of the catch function
- Java exceptions are similar to C++ exceptions except that a Java exception must be a descendant of the Throwable class. Additionally Java includes a finally clause

4.14 Logic Programming Introduction CO4

- *Logic* programming languages, sometimes called *declarative* programming languages
- Express programs in a form of symbolic logic
- Use a logical inferencing process to produce results
- *Declarative* rather that *procedural*:
  - Only specification of *results* are stated (not detailed *procedures* for producing them)

Proposition

- A logical statement that may or may not be true
  - Consists of objects and relationships of objects to each other

Symbolic Logic

- Logic which can be used for the basic needs of formal logic:
  - Express propositions
  - Express relationships between propositions
  - Describe how new propositions can be inferred from other propositions
- Particular form of symbolic logic used for logic programming called *predicate calculus*

Object Representation

- Objects in propositions are represented by simple terms: either constants or variables
- *Constant*: a symbol that represents an object
- *Variable*: a symbol that can represent different objects at different times
  - Different from variables in imperative languages

Compound Terms

- *Atomic propositions* consist of compound terms
- *Compound term*: one element of a mathematical relation, written like a mathematical function
  - Mathematical function is a mapping
  - Can be written as a table

Parts of a Compound Term

- Compound term composed of two parts
  - Functor: function symbol that names the relationship
  - Ordered list of parameters (tuple)
- Examples:

  student(jon)
  like(seth, OSX)
  like(nick, windows)
  like(jim, linux)

Forms of a Proposition

- Propositions can be stated in two forms:
  - *Fact*: proposition is assumed to be true
  - *Query*: truth of proposition is to be determined
- Compound proposition:
  - Have two or more atomic propositions
  - Propositions are connected by operators

Logical Operators

Name	Symbol	Example	Meaning
Negation	¬	¬a	a not b
Conjunction	∩	a∩b	a and b
Disjunction	∪	a∪b	a or b
Equivalence	≡	a≡b	a is equivalent to b
Implication	⊃ ⊂	a⊃b a⊂b	a implies b b implies a

Quantifiers

Name	Example	Meaning
universal	∀X.P	For all X, P is true
existential	∃X.P	There exists a value of X such that P is true

Clausal Form

- Too many ways to state the same thing
- Use a standard form for propositions
- *Clausal form*:
  - B1∪ B2 ∪...∪ Bn ⊂ A1∩ A2 ∩...∩ Am
  - means if all the As are true, then at least one B is true
- *Antecedent*: right side
- *Consequent*: left side

Predicate Calculus and Proving Theorems

- A use of propositions is to discover new theorems that can be inferred from known axioms and theorems
- *Resolution*: an inference principle that allows inferred propositions to be computed from given propositions

Resolution

- *Unification*: finding values for variables in propositions that allows matching process to succeed
- *Instantiation*: assigning temporary values to variables to allow unification to succeed
- After instantiating a variable with a value, if matching fails, may need to *backtrack* and instantiate with a different value

Theorem Proving

- Basis for logic programming
- When propositions used for resolution, only restricted form can be used
- *Horn clause* - can have only two forms
  - *Headed*: single atomic proposition on left side
  - *Headless*: empty left side (used to state facts)
- Most propositions can be stated as Horn clauses

An Overview of Logic Programming

- Declarative semantics
  - There is a simple way to determine the meaning of each statement
  - Simpler than the semantics of imperative languages
- Programming is nonprocedural
  - Programs do not state now a result is to be computed, but rather the form of the result

The Origins of Prolog

- University of Aix-Marseille
  - Natural language processing
- University of Edinburgh
  - Automated theorem proving

4.15 The Basic Elements of ProLog CO4

- Edinburgh Syntax
- *Term*: a constant, variable, or structure
- *Constant*: an atom or an integer
- *Atom*: symbolic value of Prolog
- Atom consists of either:
  - a string of letters, digits, and underscores beginning with a lowercase letter
  - a string of printable ASCII characters delimited by apostrophes

Terms: Variables and Structures

- *Variable*: any string of letters, digits, and underscores beginning with an uppercase letter
- *Instantiation*: binding of a variable to a value
  - Lasts only as long as it takes to satisfy one complete goal
- *Structure*: represents atomic proposition

  functor(*parameter list*)

Fact Statements

- Used for the hypotheses
- Headless Horn clauses

  female(shelley).
  male(bill).
  father(bill, jake).

Rule Statements

- Used for the hypotheses
- Headed Horn clause
- Right side: *antecedent* (***if*** part)
  - May be single term or conjunction
- Left side: *consequent* (***then*** part)
  - Must be single term
- *Conjunction*: multiple terms separated by logical AND operations (implied)

Example Rules

ancestor(mary,shelley):- mother(mary,shelley).

- Can use variables (*universal objects*) to generalize meaning:

  parent(X,Y):- mother(X,Y).
  parent(X,Y):- father(X,Y).
  grandparent(X,Z):- parent(X,Y), parent(Y,Z).
  sibling(X,Y):- mother(M,X), mother(M,Y),
  father(F,X), father(F,Y).

Goal Statements

- For theorem proving, theorem is in form of proposition that we want system to prove or disprove – *goal statement*
- Same format as headless Horn

  man(fred)
- Conjunctive propositions and propositions with variables also legal goals

  father(X,mike)

Inferencing Process of Prolog

- Queries are called goals

- If a goal is a compound proposition, each of the facts is a subgoal
- To prove a goal is true, must find a chain of inference rules and/or facts.

For goal Q:

```
B :- A
C :- B
...
Q :- P
```

- Process of proving a subgoal called matching, satisfying, or resolution

Approaches

- *Bottom-up resolution, forward chaining*
  - Begin with facts and rules of database and attempt to find sequence that leads to goal
  - Works well with a large set of possibly correct answers
- *Top-down resolution, backward chaining*
  - Begin with goal and attempt to find sequence that leads to set of facts in database
  - Works well with a small set of possibly correct answers
- Prolog implementations use backward chaining

Subgoal Strategies

- When goal has more than one subgoal, can use either
  - Depth-first search: find a complete proof for the first subgoal before working on others
  - Breadth-first search: work on all subgoals in parallel
- Prolog uses depth-first search
  - Can be done with fewer computer resources

Backtracking

- With a goal with multiple subgoals, if fail to show truth of one of subgoals, reconsider previous subgoal to find an alternative solution: *backtracking*
- Begin search where previous search left off
- Can take lots of time and space because may find all possible proofs to every subgoal

Simple Arithmetic

- Prolog supports integer variables and integer arithmetic
- is operator: takes an arithmetic expression as right operand and variable as left operand

```
A is B / 17 + C
```

- Not the same as an assignment statement!

Example

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :-    speed(X,Speed),
                    time(X,Time),
                    Y is Speed * Time.
```

Trace

- Built-in structure that displays instantiations at each step
- *Tracing model* of execution - four events:
  - *Call* (beginning of attempt to satisfy goal)
  - *Exit* (when a goal has been satisfied)