Module 5

Data Abstraction and Object Orientation:-Encapsulation, Inheritance, Constructors and Destructors, Aliasing, Overloading, Polymorphism, Dynamic Method Binding, Multiple Inheritance. Innovative features of Scripting Languages:-Scoping rules, String and Pattern Manipulation, Data Types, Object Orientation.

Data Abstraction and Object Orientation

Modules, allow a collection of subroutines to share a set of static variables.
Module *types*, allows the programmer to instantiate multiple instances of a given abstraction, and *classes*, allow the programmer to define families of related abstractions.
Module types and classes allow the module itself to *be* the abstract type.
Classes build on the module-as-type approach by adding mechanisms for *inheritance*,
This allow new abstractions to be defined as refinements or extensions to existing ones.
Dynamic method binding, allows a new version of an abstraction to display newly refined behaviour.
An instance of a class is known as an *object* ; languages and programming techniques based on classes are said to be *object-oriented*.
The stepwise evolution of data abstraction mechanisms suggests that object-oriented programming developed as an outgrowth of modules.
Encapsulation, inheritance, and dynamic method binding—have their roots in the Simula programming language.
Smalltalk employs a distinctive "message-based" programming model, with dynamic typing and unusual terminology and syntax.
In *multiple inheritance*, a class is defined in terms of more than one existing class.

Object-Oriented Programming

The abstraction provided by modules and module types has at least three important benefits:
1. It reduces *conceptual load* by minimizing the amount of detail that the programmer must think about at one time.
2. It provides *fault containment* by preventing the programmer from using a program component in inappropriate ways, and by limiting the portion of a program's text in which a given component can be used.
3. It provides a significant degree of *independence* among program components, making it easier to assign their construction to separate individuals.
Object-oriented programming can be seen as an attempt to enhance opportunities for code reuse by making it easy to define new abstractions as *extensions* or *refinements* of existing abstractions.
Each element of a list(doubly linked list) is an object of class list_node.
The class contains both *data members* -prev, next, head_node, and val and *subroutine members* -predecessor, successor, insert_before and remove.
Subroutine members are called *methods* in many object-oriented languages; data members are also called *fields*.

```
class list_node {
list_node* prev;
list_node* next;
list_node* head_node;
public:
int val; // the actual data in a node
list_node() { // constructor
prev = next = head_node = this; // point to self
val = 0; // default value
```

}
In C++, one can also simply declare an object of a given class:
list my_list;
list_node elem;
List class includes such an object (header) as a field.
When created with new, an object is allocated in the heap; when created via elaboration of a declaration it is allocated statically or on the stack, depending on lifetime.
In either case, creation causes the invocation of a programmer-specified initialization routine, known as a *constructor*.
In C++ and its descendants, Java and C#, the name of the constructor is the same as that of the class itself.
C++ also allows the programmer to specify a *destructor* method that will be invoked automatically when an object is destroyed.
Public and Private Members: The public label within the list of members of list_node separates members required by the implementation of the abstraction from members available to users of the abstraction.
C++ also provides a private label, so the publicly visible portions of a class can be listed first if desired.

```
class list_node {
list_node* prev;
list_node* next;
list_node* head_node;
public:
int val;
list_node();
.....
}
```

Within a .cc file, the header of a method definition must identify the class to which it belongs by using a :: *scope resolution* operator:

```
void list_node::insert_before(list_node* new_node) {
if (!new_node->singleton())
.....
prev = new_node;
new_node->head_node = head_node;
}
```

Tiny Subroutines
Object-oriented programs tend to make many more subroutine calls than do ordinary imperative programs, and the subroutines tend to be shorter.
C# provides a *property* mechanism specifically designed to facilitate the declaration of methods (called *accessors*) to “get” and “set” values.
Users of the list_node class can now access the (private) val field through the (public) Val property as if it were a field:

```
list_node n;
...
int a = n.Val; // implicit call to get method
n.Val = 3; // implicit call to set method
```

A similar *indexer* mechanism can make objects of arbitrary classes look like arrays, with conventional subscript syntax in both l-value and r-value contexts.
Derived Classes

In an object-oriented language we have a better alternative: we can *derive* the queue from the list, allowing it to *inherit* pre-existing fields and methods:

```
class queue : public list { // derive from list
public:
void enqueue(list_node* new_node) {
append(new_node);
}
list_node* dequeue() {
.....................................
}
};
```

Here queue is said to be a *derived class* (also called a *child class* or *subclass*); list is said to be a *base class* (also called a *parent class* or *superclass*).
The derived class automatically has all the fields and methods of the base class.

General Purpose base class

Given an inheritance mechanism, we can create a general-purpose element base class that contains only the fields and methods needed to implement list operations.
We can use this general-purpose class to derive lists and queues with specific types of fields.

Overloaded Constructors

We have overloaded the constructor in int_list_node, providing two alternative implementations. One takes an argument, the other does not.
The programmer can create int_list_nodes with or without specifying an initial value:

```
int_list_node element1; // val = 0
int_list_node *e_ptr = new int_list_node(13); // val = 13
```

Modifying Base Class Methods

To *redefine* a method of a base class, a derived class simply declares a new version.
Suppose that we are creating an int_list_node class, but we want somewhat different semantics for the remove method.
gp_list_node::remove will throw a list_err exception if the node to be removed is not currently on a list.

```
class int_list_node : public gp_list_node {
public:
...
.............
};
```

A better approach is to leave the implementation details to the base class and simply catch the exception if it arises.
Object-oriented languages provide other means of accessing the members of a base class.
In Smalltalk, Objective-C, Java, and C#, one uses the keyword base or super:

```
gp_list_node::remove(); // C++
super.remove(); // Java
base.remove(); // C#
```

Containers/Collections

In object-oriented programming, an abstraction that holds a collection of objects of some given class is often called a *container*.
Common containers include sets, stacks, queues, and dictionaries.
In order to put an *arbitrary* object into, say, a list, we can adopt an alternative approach.

List nodes are separate objects containing *pointers* to the listed objects, rather than the data of the objects themselves.

Encapsulation and Inheritance

Encapsulation mechanisms enable the programmer to group data and the subroutines that operate on them together in one place, and to hide irrelevant details from the users of an abstraction.

Modules

Scope rules for data hiding were one of the principal innovations of Clu, Modula, Euclid, and other module-based languages.

Declaration and definition (header and body) of a module always appear together.

Ada, which also allows the headers and bodies of modules (called packages) to be separated, eliminates the problems of Modula-2 by allowing the header of a package to be divided into public and private parts.

A type can be exported opaquely by putting its definition in the private part of the header and simply
naming it in the public part.

```
package foo is              -- header
...
type T is private;
...
private                     -- definitions below here are inaccessible to users
...
type T is ... -- full definition
...
end foo;
```

Module types, as in Euclid, are somewhat more complicated: they allow a module to have an arbitrary number of *instances*.

The obvious implementation then resembles that of a record.

The "this" Parameter

A better technique is to create a single instance of each module subroutine, and to pass that instance, at run time, the address of the storage of the appropriate module instance.

This address takes the form of an extra, hidden first parameter for every module subroutine.

A Euclid call of the form
my_stack.push(x)
is translated as
push(my_stack, x) where my_stack is passed by reference.

In Java, C#, there is no need to manually identify code that needs to be in the header for implementation reasons.

The compiler is responsible for extracting this information automatically from the full text of the module.

Classes

To effect this hiding in C++, the definition of class queue can specify that its base class is to be private:

```
class queue : private list {
public:
.....................
void enqueue(gp_list_node* new_node);
```

.....

};
Here the appearance of private in the first line of the declaration indicates that public members of list will be visible to users of queue only if specifically made so by later parts of the declaration.
In addition to the public and private labels, C++ allows members of a class to be designated protected.
A protected member is visible only to methods of its own class or of classes derived from that class.
The protected keyword can also be used when specifying a base class:
class derived : protected base { ...

Visibility rules in C++:
-Any class can limit the visibility of its members.
-Public members are visible anywhere the class declaration is in scope.
-A derived class can restrict the visibility of members of a base class, but can never increase it. Private members of a base class are never visible in a derived class.
-A derived class that limits the visibility of members of a base class by declaring that base class protected or private can restore the visibility of individual members of the base class.

Derived classes in Eiffel can both restrict *and increase* the visibility of members of base classes. Every method called a feature in Eiffel can specify its own *export status.*
In the general case the status can be an arbitrary list of class names, in which case the feature is said to be *selectively available* to those classes and their descendants only.
Nesting (Inner Classes)
Many languages allow class declarations to nest.
Nesting serves simply as a means of information hiding.
Java takes a more sophisticated approach.
It allows a nested (*inner*) class to access arbitrary members of its surrounding class.
Each instance of the inner class must therefore *belong to* an instance of the outer class.

```
class Outer {
int n;
class Inner {
public void bar() { n = 1; }
}
Inner i;
Outer() { i = new Inner(); } // constructor
public void foo() {
.....
i.bar();
System.out.println(n); // prints 1
}
}
```

If there are multiple instances of Outer, each instance will have a different n, and calls to Inner.bar will access the appropriate n.
Java classes can also be nested inside methods.
Such a *local* class not only has access to (all) members of the surrounding class; it also has a copy of any final parameters and variables of the method in which it is nested.
Inner and local classes in Java are widely used to create *object closures,* we used them as handlers for events.
A local class in Java can be *anonymous*: it can appear, in-line, inside a call to new.

Type Extensions

Smalltalk, Eiffel, C++, Java etc.. were designed from an existing language without a strong encapsulation mechanism.

They all support a module as-type approach to abstraction, in which a single mechanism (the class) provides both encapsulation and inheritance.

Rather than alter the existing module mechanism, these languages provide inheritance and dynamic method binding through a mechanism for *extending* records.

To control access to the structure of types, we hide them inside Ada packages.

The procedures initialize, finalize, enqueue, and dequeue of gp_list.queue can convert their parameter self to a list_ptr, because queue is an extension of list.

Package gp_list.queue is said to be a *child* of package gp_list because its name is prefixed with that of its parent.

Extending without Inheritance

The desire to extend the functionality to an existing abstraction is one of the principal motivations for object-oriented programming.

For situations like these, C# 3.0 provides *extension methods*, which give the appearance of extending an existing class:

```
static class AddToString {
public static int toInt(this string s) {
return int.Parse(s);
}
}
```

An extension method must be static, and must be declared in a static class.

Its first parameter must be prefixed with the keyword this.

Constructors and Destructors

Most object-oriented languages provide some sort of special mechanism to *initialize* an object automatically at the beginning of its lifetime.

When written in the form of a subroutine, this mechanism is known as a *constructor*.

A constructor does not allocate space; it initializes space that has already been allocated.

A few languages provide a similar *destructor* mechanism to *finalize* an object automatically at the end of its lifetime.

Several issues are:-

Choosing a constructor: An object-oriented language may permit a class to have zero, one, or many distinct constructors.

References and values: If variables are references, then every object must be created explicitly, and it is easy to ensure that an appropriate constructor is called.

*Execution order: T*he compiler guarantees that the constructors for any base classes will be executed, outermost first, before the constructor for the derived class.

Garbage collection: Most object-oriented languages provide some sort of constructor mechanism. Destructors are comparatively rare.

Choosing a Constructor

Smalltalk, Eiffel, C++, Java, and C# all allow the programmer to specify more than one constructor for a given class.

In C++, Java, and C#, the constructors behave like overloaded subroutines.

Different constructors can have different names; code that creates an object must name a constructor explicitly.

```
.....
a, b : COMPLEX
...
!!b.new_cartesian(0, 1)
!!a.new_polar(pi/2, 1)
```

The !! operator is Eiffel's equivalent of new.
Because class COMPLEX specified constructor ("creator") methods, the compiler will insist that every use of !! specify a constructor name and arguments.
Smalltalk also adopts a programming model in which every operation is seen as being executed by some specific object in response to a request (a "message") from some other object.
Each class definition really introduces a *pair* of classes and a pair of objects to represent them.
Consider the standard class named Date.
Corresponding to Date is a single object (call it *D*) that performs operations on behalf of the class
Simply use the name of the class it represents:
todaysDate <- Date today
This code causes *D* to execute the today constructor of class Date, and assigns a reference to the newly created object into a variable named todaysDate.
D represents class Date.
Smalltalk says that *D* is an object of the *metaclass* Date class.
References and Values: Several object-oriented languages, including Simula, Smalltalk, Python,Ruby, and Java, use a programming model in which variables refer to objects.
With a reference model for variables every object is created explicitly, and it is easy to ensure that an appropriate constructor is called.
With a value model for variables object creation can happen implicitly as a result of elaboration.
If a C++ variable of class type foo is declared with no initial value, then the compiler will call foo's zero-argument constructor.
foo b; // calls foo::foo()

foo c = a; // calls foo::foo(foo&)
foo d = b; // calls foo::foo(bar&)
In recognition of this intent, a single-argument constructor in C++ is called a *copy constructor*.
It is important to realize here that the equals sign (=) in these declarations indicates initialization, not assignment.
Execution Order
When the programmer creates an object of class *B* -either via declaration or with a call to new, the creation operation specifies arguments for a *B* constructor.
C++ allow the header of the constructor of a derived class to specify base class constructor arguments:
foo::foo(*foo params*) : bar(*bar args*) {

...
The list *foo_ params* consists of formal parameters for this particular foo constructor.
class foo : bar {
.....
}
foo::foo(*foo params*) : bar(*bar args*), member1(*mem1 args*),
member2(*mem2 args*) {
...
Like C++, Java insists that a constructor for a base class be called before the constructor for a derived class.
The syntax is a bit simpler, however; the initial line of the code for the derived class constructor may consist of a "call" to the base class constructor:
super(*args*);
Java uses a reference model uniformly for all objects, any class members that are themselves objects will actually be *references*, rather than "expanded" objects.
Garbage Collection

When a C++ object is destroyed, the destructor for the derived class is called first, followed by those of the base class(es), in reverse order of derivation.
By far the most common use of destructors in C++ is manual storage reclamation.

```
.....
~name_list_node() {
if (name != 0) {
delete[] name; // reclaim space
}
}
```

The destructor in this class serves to reclaim space that was allocated in the heap by the constructor.
In languages with automatic garbage collection, there is much less need for destructors.
In fact, the entire idea of destruction is suspect in a garbage-collected language, because the programmer has little or no control over when an object is going to be destroyed.

Aliases

Two or more names that refer to the same object at the same point in the program are said to be *aliases.*
A name that can refer to more than one object at a given point in the program is said to be *overloaded.*
Eg. of aliases occur in the common blocks and equivalence statements of Fortran, and in the variant records and unions of languages like Pascal and C.
Consider the following code in C++.

```
double sum, sum_of_squares;
...
void accumulate(double& x) // x is passed by reference
{
sum += x;
sum_of_squares += x * x;
}
...
accumulate(sum);
```

If sum is passed as an argument to accumulate, then sum and x will be aliases for one another, and the program will probably not do what the programmer intended.
Aliases tend to make programs more confusing than they otherwise would be.
They also make it much more difficult for a compiler to perform certain important code improvements.
Consider the following C code:

```
int a, b, *p, *q;
...
a = *p; /* read from the variable referred to by p */
*q = 3; /* assign to the variable referred to by q */
b = *p; /* read from the variable referred to by p */
```

The initial assignment to a will, require that *p be loaded into a register.
Since accessing memory is expensive, the compiler will want to hang on to the loaded value and reuse it in the assignment to b.

Overloading

In C, the plus sign (+) is used to name several different functions, including signed and unsigned integer and floating-point addition.
Functions will be based on the same mathematical concept, but they take arguments of different types and perform very different operations on the underlying bits.
A slightly more sophisticated form of overloading appears in the enumeration constants of Ada.

In Figure, the constants oct and dec refer either to months or to numeric bases, depending on the context in which they appear. _
Within the symbol table of a compiler, overloading must be handled by arranging for the *lookup* routine to return a *list* of possible meanings for the requested name.
The semantic analyzer must then choose from among the elements of the list based on context.

```
declare
type month is (jan, feb, mar, apr, may, jun,
.....oct,nov, dec);
type print_base is (dec, bin, oct, hex);
mo : month;
pb : print_base;
begin
mo := dec;              -- the month dec (since mo has type month)
pb := oct;              -- the print_base oct (since pb has type print_base)
print(oct);                 -- error! insufficient context
                          -- to decide which oct is intended
```

Figure: Overloading of enumeration constants in Ada.
Overloaded enumeration constants allow the programmer to provide appropriate context explicitly.
In Ada, for example, one can say
print(month'(oct));
InModula-3 and C#, *every* use of an enumeration constant must be prefixed with a type name, even when there is no chance of ambiguity:
mo := month.dec;
pb := print_base.oct;
In Ada and C++, a given name may refer to an arbitrary number of subroutines in the same scope, so long as the subroutines differ in the number or types of their arguments.

```
struct complex {
.....
};
enum base {dec, bin, oct, hex};
.....
void print_num(int n) { ...
void print_num(int n, base b) { ...
void print_num(complex c) { ...
print_num(i); // uses the first function above
.....
```

Figure Simple example of overloading in C++.
In each case the compiler can tell which function is intended by the number and types of arguments.
Redefining Built-in Operators
In C++ and C#, which are object-oriented, A + B may be short for either operator+(A, B) or A.operator+(B).
In the latter case, A is an instance of a class (module type) that defines an operator+ function.
In C++:

```
class complex {
.....
...
public:
complex operator+(complex other) {
```

```
return complex(real + other.real, imaginary + other.imaginary);
}
...
};
...
complex A, B, C;
...
C = A + B; // uses user-defined operator+
```

Polymorphism and Related Concepts

In certain circumstances, we can pass arguments of multiple types to or return values of multiple types from a given named routine.

To compute the minimum of two values of either integer or floating-point type.

In C, we could get by with a single function:

```
double min(double x, double y) { ...
```

If the C function is called in a context that expects an integer (e.g., i = min(j,k)), the compiler will automatically convert the integer arguments (j and k) to floating-point numbers, call min, and then convert the result back to an integer.

Coercion is the process by which a compiler automatically converts a value of one type into a value of another type when that second type is required by the surrounding context.

C will perform these same coercions on arguments to functions.

Most scripting languages provide a very rich set of built-in coercions.

C++ allows the programmer to extend its built-in set with user-defined coercions.

Coercion allows the C compiler to modify the arguments to fit a *single* subroutine.

Polymorphism provides yet another option: it allows a single subroutine to accept *unconverted* arguments of multiple types.

In *parametric polymorphism* the code takes a type or set of types as a parameter, either explicitly or implicitly.

In *subtype polymorphism* the code is designed to work with values of some specific type *T*, but the programmer can define additional types to be extensions or refinements of *T*, and the polymorphic

code will work with these *subtypes* as well.

Explicit parametric polymorphism is also known as *genericity*.

Generic facilities appear in Ada, C++, Clu, Eiffel, Modula-3, Java, and C#, among others.

Subtype polymorphism is fundamental to object-oriented languages, in which subtypes (classes) are said to *inherit* the methods of their parent types.

Generics or explicit parametric polymorphism are usually, though not always, implemented by creating multiple copies of the polymorphic code, one specialized for each needed concrete type.

Inheritance or subtype polymorphism is almost always implemented by creating a single copy of the code, and by including in the representation of objects sufficient “metadata” i.e data about the data that the code can tell when to treat them differently.

Most Lisp implementations use a single copy of the code, and delay all semantic checks until run time.

With the *implicit* parametric polymorphism of Lisp, ML, and their descendants, the programmer need not specify a type parameter.

The Scheme definition of min looks like this:

```
(define min (lambda (a b) (if (< a b) a b)))
```

It makes no mention of types.

The typical Scheme implementation employs an interpreter that examines the arguments to min and determines, at run time, whether they support a < operator.

With overloading the programmer must write a separate copy of the code, by hand, for every type with a min operation.

Generics allow the compiler to create a copy automatically for every needed type.

The similarity of the calling syntax and of the generated code has led some authors to refer to overloading as *ad hoc* or special case *polymorphism.*

Dynamic Method Binding

In Ada terminology, a derived class that does not hide any publicly visible members of its base class is a *subtype* of that base class.

The ability to use a derived class in a context that expects its base class is called *subtype polymorphism.*

If we imagine an administrative computing system for a university, we might derive classes student and professor from class person:

```
class person { ...
class student : public person { ...
class professor : public person { ...
```

Because both student and professor objects have all the properties of a person object, we should be able to use them in a person context:

```
student s;
professor p;
...
```

We have multiple versions of our subroutine—student::print_mailing_label and professor::print_mailing_label, rather than the single, polymorphic person::print_mailing_label.

Which version we will get depends on the object:

```
s.print_mailing_label(); // student::print_mailing_label(s)
p.print_mailing_label(); // professor::print_mailing_label(p)

x->print_mailing_label(); // ??
y->print_mailing_label(); // ??
```

The choice of the method to be called depend on the types of the *variables* x and y, or on the classes of the *objects* s and p to which those variables refer.

The first option (use the type of the reference) is known as *static method binding*.

The second option (use the class of the object) is known as *dynamic method binding.*

Definitions in the derived classes are said to *override* the definition in the base class.

Dynamic method binding imposes run-time overhead.

While this overhead is generally modest, it is a concern for small subroutines in performance-critical applications.

Smalltalk, Objective-C,Modula-3, Python, and Ruby use dynamic method binding for all methods.

Overriding a method that uses dynamic binding and (merely) *redefining* a method that uses static binding.

C# requires explicit use of the keywords override and new whenever a method in a derived class overrides or redefines (respectively) a method of the same name in a base class.

Virtual and Nonvirtual Methods

In Simula, C++, and C#, which use static method binding by default, the programmer can specify that particular methods should use dynamic binding by labelling them as virtual.

Calls to virtual methods are *dispatched* to the appropriate implementation at run time, based on the class of the object, rather than the type of the reference.

In C++ and C#, the keyword virtual prefixes the subroutine declaration:

```
class person {
public:
virtual void print_mailing_label();
```

...
Rather than associate dynamic dispatch with particular methods, the Ada 95 programmer associates it with certain *references*.
A formal parameter or an access variable (pointer) can be declared to be of the *class-wide* type person'Class.
Here, all calls to all methods of that parameter or variable will be dispatched based on the class of the object to which it refers.

Abstract Classes

In most object-oriented languages it is possible to omit the body of a virtual method in a base class. In Java and C#, one does so by labeling both the class and the missing method as abstract:

```
abstract class person {
...
public abstract void print_mailing_label();
...
```

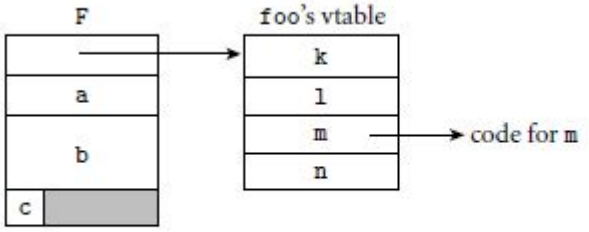
The notation in C++ is somewhat less intuitive: one follows the subroutine declaration with an "assignment" to zero.
C++ refers to abstract methods as *pure virtual* methods.
In Simula all virtual methods are abstract.
Regardless of declaration syntax, a *class* is said to be abstract if it has at least one abstract method.
The only purpose of an abstract class is to serve as a base for other, *concrete* classes.
A concrete class must provide a real definition for every abstract method it inherits.
Classes that have no members other than abstract methods—no fields or method bodies—are called *interfaces* in Java, C#, and Ada.

Member Lookup

Implementation of virtual methods:-

The representation of object F begins with the address of the vtable for class foo.
All objects of this class will point to the same vtable.
The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. The remainder of F consists of the representations of its fields.

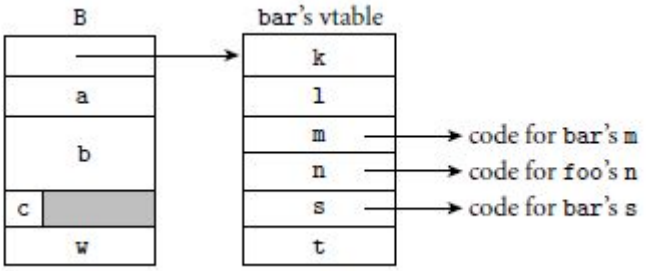
```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```



The most common implementation represents each object with a record whose first field contains the address of a *virtual method table* (*vtable*) for the object's class.

Implementation of single inheritance:-

```
class bar : public foo {
    int w;
public:
    void m();  //override
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```



The representation of object B begins with the address of its class's vtable.
The first four entries in the table represent the same members as they do for foo, except that one—m—has been overridden and now contains the address of the code for a different subroutine.
Additional fields of bar follow the ones inherited from foo in the representation of B.
Additional virtual methods follow the ones inherited from foo in the vtable of class bar.

Polymorphism

Dynamic method binding introduces polymorphism (specifically, *subtype* polymorphism) into any code that expects a reference to an object of some base class foo.
We can see an example in the gp_list_node class and its descendants.
By placing the structural aspects of an abstraction in a base class, we make it easy to create type-specific lists: int_list_node, float_list_node, student_list_node, etc.
Base class methods like predecessor and successor return references of the base class type.
We must perform an explicit type cast:

```
int_list_node_ptr q, r;
...
r = q->successor(); // error: type clash
gp_list_node_ptr p = q->successor();
cout << p.val; // error: gp_list_nodes have no val
r = (int_list_node_ptr) q->successor();
cout << r.val; // ok
```

The cast on the penultimate line here is both awkward and unsafe.
We can't use a dynamic_cast operation because gp_list_node has no virtual members.
We can redefine methods:

```
int_list_node* int_list_node::predecessor() { // redefine
return (int_list_node*) gp_list_node::predecessor();
```

Redefining all of the appropriate arguments and return types of base class methods in every derived class is tedious, and the code is still unsafe.
The compiler cannot verify type correctness.
Generics get around both problems.
In C++, we can write

```
template<class V>
class list_node {
list_node<V>* prev;
.....
public:
V val;
list_node<V>* predecessor() { ...
.....
void insert_before(list_node<V>* new_node) { ...
...
};
```

In a nutshell, generics exist for the purpose of abstracting over unrelated types, something that inheritance does not support.
If a derived class redefines the anchor, the parameters and return values are automatically redefined as well, without the need to specify them explicitly:

```
class gp_list_node ...
...
class gp_list
.....................................
header : gp_list_node                    -- to be redefined by derived classes
.....................................
head : like header is ...                -- methods
append(new_node : like header) is ...
...
end
...
class student_list_node inherit gp_list_node ...
...
class student_list
inherit gp_list
redefine header end
.....................................
header : student_list_node
            -- don't need to redefine head and append
end
```

Object Closures

Object closures can be used in an object-oriented language to achieve roughly the same effect as subroutine closures in a language with nested subroutines.
This is to encapsulate a method with *context* for later execution.
It should be noted that this mechanism relies, for its full generality, on dynamic method binding.
Here we have adapted the apply_to_A code of plus_x and rewritten in generic form:

```
template<class T>
class un_op {
public:
.....................................
plus_x(int n) : x(n) { }
virtual int operator()(int i) const { return i + x; }
};
void apply_to_A(const un_op<int>& f, int A[], int A_size) {
int i;
.....................................
}
.....................
```

Any object derived from un_op<int> can be passed to apply_to_A.
The "right" function will always be called because operator() is virtual.
A particularly useful idiom for many applications is to to encapsulate a method *and its arguments* in an object closure for later execution.

Multiple Inheritance

It can be useful for a derived class to inherit features from more than one base class.
It may then be desirable to derive class student from both person and gp_list_node.
In C++ we can say

class student : public person, public gp_list_node { ...
Now an object of class student will have all the fields and methods of both a person and a gp_list_node.
The declaration in Eiffel :
class student
inherit
person
gp_list_node
feature
... _
Multiple inheritance also appears in CLOS and Python.
Simula, Smalltalk, Objective-C, Modula-3, Ada 95 etc.. have only single inheritance.
Java, C#, and Ruby provide a limited, "mix-in" form of multiple inheritance, in which only one parent class is permitted to have fields.
Multiple inheritance with a common"grandparent"is known as *repeated* inheritance.
Repeated inheritance with separate copies of the grandparent is known as *replicated* inheritance.
Repeated inheritance with a single copy of the grandparent is known as *shared* inheritance.
Shared inheritance is the default in Eiffel.
Replicated inheritance is the default in C++.
Both languages allow the programmer to obtain the other option when desired.
The complexity can be reduced if we insist, like Java, C#, or Ada 2005, where one of the parent classes consist of methods only.
All three languages call such a class an interface.

Innovative features of Scripting languages

General-purpose scripting languages like Perl and Python are sometimes called *glue languages*, because they were originally designed to "glue" existing programs together to build a larger system.
With the growth of the World Wide Web, scripting languages have gained new prominence in the generation of dynamic content.
They are also widely used as *extension languages*, which allow the user to customize or extend the functionality of "scriptable" tools.
Scripting languages have two principal sets of ancestors.
In one set are the command interpreters or "shells" of traditional batch and "terminal" –command line computing.
In the other set are various tools for text processing and report generation.
Examples in the first set include IBM's JCL, the MS-DOS command interpreter, and the Unix sh and csh shell families.
Examples in the second set include IBM's RPG and Unix's sed and awk.
Several common characteristics of scripting languages:-
1. Both batch and interactive use
2. Economy of expression
3. Lack of declarations; simple scoping rules
4. Flexible dynamic typing
5. Easy access to other programs
6. Sophisticated pattern matching and string manipulation
7. High-level data types

Scoping rules

Names and Scopes:-

Most scripting languages except Scheme do not require variables to be declared.
With or without declarations, most scripting languages use dynamic typing.

Values are generally self-descriptive, so the interpreter can perform type checking at run time, or coerce values when appropriate.
Nesting and scoping conventions vary quite a bit.
Scheme, Python, JavaScript, and R provide the classic combination of nested subroutines and static (lexical) scope.
Tcl allows subroutines to nest, but uses dynamic scoping.
Named subroutines (methods) do not nest in PHP or Ruby.
Perl and Ruby join Scheme, Python, JavaScript, and R in providing first-class anonymous local subroutines.
Nested blocks are statically scoped in Perl.

```
i = 1; j = 3
def outer():
def middle(k):
def inner():
global i                    # from main program, not outer
i = 4
inner()
return i, j, k              # 3-element tuple
i = 2                       # new local i
return middle(j)            # old (global) j
print outer()
print i, j
```

Figure: A program to illustrate scope rules in Python.
There is one instance each of j and k, but two of i: one global and one local to outer.
The scope of the latter is all of outer, not just the portion after the assignment.
The global statement provides inner with access to the outermost i, so it can write it without defining a new instance.

Scope of an Undeclared Variable

In Python and R a variable that is written is assumed to be local, unless it is explicitly imported.
A variable that is only read in a given scope is found in the closest enclosing scope that contains a defining write.
Consider, for example, the Python program of Figure.
Here we have a set of nested subroutines, as indicated by indentation level.
The main program calls outer, which calls middle, which in turn calls inner.
Before its call, the main program writes both i and j.
Outer reads j ;to pass it to middle, but does not write it.
It does, however, write i.
There is no way in Python for a nested routine to write a variable that belongs to a surrounding but non global scope.
In Figure, inner could not be modified to write outer's i.
R provides a mechanism that is rather than declare i to be global, R uses a "super assignment" operator.
Where a normal assignment i <- 4 assigns the value 4 into a local variable i, the super assignment i<<- 4 assigns 4 into whatever i would be found under the normal rules of static (lexical) scoping.
Tcl not only makes the unusual choice of employing dynamic scoping, but also implements the choice in an unusual way.
Variables in calling scopes are never accessed automatically.
The programmer must ask for them explicitly.
Any variable that is not declared is global in Perl by default.
Variables declared with the local operator are dynamically scoped.

Variables declared with the my operator are statically scoped.
The difference can be found, in which subroutine outer declares two local variables, lex and dyn.
The former is statically scoped; the latter is dynamically scoped.
Perl allows the programmer to force the use of a global variable with the our operator, whose name is intended to contrast with my.

String and Pattern Manipulation

There are many different implementations of extended regular expressions ("REs" for short), with slightly different syntax, most fall into two main groups.
The first group includes awk, egrep -the most widely used of several different versions of grep, the regex library for C, and older versions of Tcl.
These implement REs as defined in the POSIX standard.
Languages in the second group follow the lead of Perl, which provides a large set of extensions, sometimes referred to as "advanced REs."
Few tools, including sed, classic grep, and older Unix editors, provide so-called "basic" REs,
less capable than those of egrep.
In certain languages and tools—notably sed, awk, Perl, PHP, Ruby, and JavaScript—regular expressions are tightly integrated into the rest of the language, with special syntax and built-in operators.
In these languages an RE is typically delimited with slash characters, though other delimiters may be accepted in some cases.
In most other languages, REs are expressed as ordinary character strings, and are manipulated by passing them to library routines.

POSIX Regular Expressions

Like the "true" regular expressions of formal language theory, extended REs support concatenation, alternation, and Kleene closure.
Parentheses are used for grouping.
/ab(cd|ef)g*/ matches abcd, abcdg, abefg, abefgg, abcdggg, etc.
Several other *quantifiers* (generalizations of Kleene closure) are also available:
? indicates zero or one repetitions, + indicates one or more repetitions, {*n*} indicates exactly *n* repetitions.
{*n*,} indicates at least *n* repetitions, and {*n,m*} indicates *n*–*m* repetitions:
/a(bc)*/ matches a, abc, abcbc, abcbcbc, etc.
/a(bc)?/ matches a or abc
/a(bc)+/ matches abc, abcbc, abcbcbc, etc.
Two *zero-length assertions*, ^ and $, match only at the beginning and end, respectively, of a target string.
As an abbreviation for /a|b|c|d/, extended REs permit *character classes* to be specified with square brackets:
/b[aeiou]d/ matches bad, bed, bid, bod, and bud
Outside a character class, a dot (.) matches any character other than a newline.
The expression /b.d/, for example, matches not only bad, bbd, bcd, and so on, but also b:d, b7d, and many others.
A caret (^) at the beginning of a character class indicates negation: the class expression matches anything *other* than the characters inside.
A backslash will similarly protect any of the special characters | ( ) [ ] { } $. * + ? outside a character class.
Several character classes expressions are predefined in the POSIX standard.
As we saw in Example, the expression [:space:] can be used to capture white space.
For punctuation there is [:punct:].
These expressions must be used *inside* a built-up character class; they aren't classes by themselves.

Extended REs are a central part of Perl.
The built-in =~ operator is used to test for matching:
$foo = "albatross";
if ($foo =~ /ba.*s+/) ... # true
$_ is set automatically when iterating over the lines of a file.
It is also the default index variable in for loops. _
The !~ operator returns true when a pattern *does not* match:

if ("albatross" !~ /^ba.*s+/) ... # true

Modifiers and Escape Sequences

Both matches and substitutions can be *modified* by adding one or more characters after the closing delimiter.
A trailing i, for example, makes the match case-insensitive:
$foo = "Albatross";
if ($foo =~ /^al/i) ... # true

Escape sequence Meaning(In Perl)
\0 NUL character
\a alarm (BEL) character
\b backspace (within character class)
\e escape (ESC) character
\z end of string
\Z prior to final newline, or end of string if none
\d digit (decimal)
\D not a digit
.....
Perl allows nonprinting characters to be specified in REs using backslash *escape sequences*.
Perl also provides several zero-width assertions, in addition to the standard ^ and $.
The \A and \Z escapes differ from ^ and $ in that they continue to match only at the beginning and end of the string.

Greedy and Minimal Matches

The usual rule for matching in REs is sometimes called “left-most longest”: when a pattern can match at more than one place within a string- the chosen match will be the one that starts at the earliest possible position within the string.
In the string abcbcbcde, for example, the pattern /(bc)+/ can match in six different ways:
abcbcbcde
abcbcbcde
abcbcbcde
abcbcbcde
abcbcbcde
abcbcbcde
The third of these is “left-most longest,” also known as *greedy*.
In some cases, however, it may be desirable to obtain a “left-most shortest” or *minimal* match.
*? matches the smallest number of instances of the preceding subexpression that will allow the overall match to succeed.

Variable Interpolation and Capture

Like double-quoted strings, regular expressions in Perl support variable interpolation.
Any dollar sign that does not immediately proceed a vertical bar, closing parenthesis, or end of string is assumed to introduce the name of a Perl variable.
Its value as a string is expanded prior to passing the pattern to the regular expression evaluator. This allows us to write code that generates patterns at run time:

$prefix = ...
$suffix = ...
if ($foo =~ /^$prefix.*$suffix$/) ...
Every parenthesized fragment of a Perl RE is said to *capture* the text that it matches.
The captured strings may be referenced in the right-hand side of the substitution as \1, \2, and so on.
One can even use a captured string later in the RE itself.
Such a string is called a *backreference*.
For simple matches, Perl also provides pseudovariables named $‘, $&, and $’.
These name the portions of the string before, in, and after the most recent match, respectively.
$line = <>;
chop $line; # delete trailing newline
$line =~ /is/;
print "prefix($‘) match($&) suffix($’)\n";
With input “now is the time”, this code prints
prefix(now) match(is) suffix(the time)

Data Types

Scripting languages don’t generally require; or even permit the declaration of types for variables.
In Ruby, the programmer who wants to convert from one type to another must say so explicitly.
If we type the following in Ruby,
a = "4"
print a + 3, "\n"
we get the following message at run time: “In ‘+’: failed to convert Fixnum into String (TypeError).”
Perl is willing, for example, to accept the following -though it prints a warning if run with the -w compile-time switch:
$a[3] = "1"; # (array @a was previously undefined)
print $a[3] + $a[4], "\n";
Here $a[4] is uninitialized and hence has value undef.
In a numeric context as an operand of +, the string "1" evaluates to 1, and undef evaluates to 0.
A comparable code fragment in Ruby requires a bit more care.
Before we can subscript a, we must make sure that it refers to an array:
a = [] # empty array assignment
a[3] = "1"
If the first line were not present ,and a had not been initialized in any other way, the second line would have generated an “undefined local variable” error.

NumericTypes

Scripting languages generally provide a very rich set of mechanisms for string and pattern manipulation.
Internally, numbers in JavaScript are always double-precision floating point.
In Tcl they are strings, converted to integers or floating-point numbers when arithmetic is needed.
PHP uses integers, plus double-precision floating point.
To these Perl and Ruby add arbitrary precision (multiword) integers, sometimes known as *bignum*s.
Python has bignums, too, plus support for complex numbers.
Ruby is perhaps the most explicit about the existence of different representations: classes Fixnum, Bignum, and Float have overlapping but not identical sets of built-in methods.

CompositeTypes

Most scripting languages place a heavy emphasis on *mappings*, sometimes called *dictionaries*, *hashes*, or *associative arrays*.
A mapping is typically implemented with a hash table.

Perl, the oldest of the widely used scripting languages, inherits its principal composite types—the array and the hash—from awk.
It also uses prefix characters on variable names as an indication of type: $foo is a scalar -a number, Boolean, string, or pointer.
@foo is an array; %foo is a hash.
Ordinary arrays in Perl are indexed using square brackets and integers starting with 0.

```
@colors = ("red", "green", blue");          # initializer syntax
print $colors[1];                           # green
```

Hashes are indexed using curly braces and character string names.
Python and Ruby, like Perl, provide both conventional arrays and hashes.
They use square brackets for indexing in both cases, and distinguish between array and hash initializers or aggregates using bracket and brace delimiters, respectively:

```
colors = ["red", "green", "blue"]
complements = {"red" => "cyan",
"green" => "magenta", "blue" => "yellow"}
print colors[2], complements["blue"]
```

(This is Ruby syntax; Python uses : in place of =>.)
In addition to arrays (*lists*) and hashes (*dictionaries*), Python provides two other composite types: tuples and sets.
A tuple is an immutable list (array).
The initializer syntax uses parentheses rather than brackets:

```
crimson = (0xdc, 0x14, 0x3c)        # R,G,B components
```

Python sets are like dictionaries that don't map to anything of interest, but simply serve to indicate whether elements are present or absent.
Unlike dictionaries, they also support union, intersection, and difference operations:

```
X = set(['a', 'b', 'c', 'd'])          # set constructor
Y = set(['c', 'd', 'e', 'f'])          # takes array as parameter
U = X | Y                       # (['a', 'b', 'c', 'd', 'e', 'f'])
```

Tuples in Python work particularly well:

```
matrix = {}             # empty dictionary (hash)
matrix[2, 3] = 4        # key is (2, 3)
```

Context
Type compatibility, in a statically typed language determines, which types can be used in which *contexts*.
The term"context" refers to information about how a value will be used.
In C, for example, one might say that in the declaration

```
double d = 3;
```

the 3 on the right-hand side occurs in a context that expects a floating-point number.
The C compiler *coerces* the 3 to make it a double instead of an int.
Type inference, allows a compiler to determine the type of an expression based on the types of its constituent parts and, in some cases, the context in which it appears.
The assignment operator (=) provides a scalar or list context to its right-hand side based on the type of its left-hand side.
If we write,

```
$time = gmtime();
```

Perl's standard gmtime() library function will return the time as a character string, along the lines of "Sun Aug 17 15:10:32 2008".
Functions typically indicate an error by returning the empty array when called in a list context, and

the undefined value (undef) when called in a scalar context:
if (*something went wrong*) {
return wantarray ? () : undef;
}

Object Orientation

Perl 5 has features that allow one to program in an object-oriented style.
PHP and JavaScript have cleaner, more conventional-looking object-oriented features.
Perl uses a value model for variables; objects are always accessed via pointers.
In PHP and JavaScript, a variable can hold either a value of a primitive type or a reference to an object of composite type.
Classes are themselves objects in Python and Ruby, much as they are in Smalltalk.
JavaScript, has objects but no classes; its inheritance is based on a concept known as *prototypes*.

Perl 5

Object support in Perl 5 boils down to two main things:
(1) a *blessing* mechanism that associates a reference with a package.
(2) special syntax for method calls that automatically passes an object reference or package name as the initial argument to a function.
In Perl we define a package, Integer, that plays the role of a class.
It has three functions, one of which (new) is intended to be used as a constructor.
Two of which (set and get) are intended to be used as accessors.
Both Integer->new and new Integer are syntactic sugar for calls to Integer::new with an additional first argument that contains the name of the package (class) as a character string.
In the first line of function new we assign this string into the variable $class.
Once a reference has been blessed, Perl allows it to be used with method invocation syntax:
c1->get() and get c1() are syntactic sugar for Integer::get($c1).
As usual in Perl, if an argument list is empty, the parentheses can be omitted.
Inheritance in Perl is obtained by means of the @ISA array, initialized at the global level of a package.
Most often packages (and thus classes) in Perl are declared in separate modules (files).
In this case, one must import the module corresponding to a superclass in addition to modifying @ISA.
The standard base module provides convenient syntax for this combined operation, and is the preferred way to specify inheritance relationships:

```
{   package Tally;
    use base ("Integer");
    ...
```

PHP and JavaScript

Class declarations in PHP must include declarations of all members -fields and methods, and the set of members in a given class cannot subsequently change.
JavaScript takes the unusual approach of providing objects—with inheritance and dynamic method dispatch—without providing classes.
Such a language is said to be *object-based*, as opposed to object-oriented.
Functions are first-class entities in JavaScript—objects, in fact.
A method is simply a function that is referred to by a *property* (member) of an object.
When we call o.m, the keyword this will refer to o during the execution of the function referred to by m.
Associated with every constructor f is an object f.prototype.
If object o was constructed by f, then JavaScript will look in f.prototype whenever we attempt to use a property of o that o itself does not provide.

Use of Prototypes:-

Function Integer serves as a constructor.

Assignments to properties of Integer.prototype serve to establish methods for objects constructed by Integer.
We can write,

```
c2 = new Integer(3);
c3 = new Integer;
document.write(c2.get() + "  " + c3.get() + "<BR>");
c2.set(4); c3.set(5);
document.write(c2.get() + "  " + c3.get() + "<BR>");
```

This code will print

```
3 0
4 5
```

We can override methods and fields on an object-by-object basis.

```
function Integer(n) {
this.val = n || 0;                    // use 0 if n is missing (undefined)
}
function Integer_set(n) {
this.val = n;
}
function Integer_get() {
.....................
}
Integer.prototype.set = Integer_set;
Integer.prototype.get = Integer_get;
```

Figure :Object-oriented programming in JavaScript.
The Integer function is used as a constructor.
Assignments to members of its prototype object serve to establish methods.
These will be available to any object created by Integer that doesn't have corresponding members of its own.
To obtain the effect of inheritance, we can write

```
function Tally(n) {
this.base(n);                 // call to base constructor
}
function Tally_inc() {
this.val++;
}
Tally.prototype = new Integer;          // inherit methods
Tally.prototype.base = Integer;        // make base constructor available
.............................
...
t1 = new Tally(3);
t1.inc();          t1.inc();
.....................................
```

<u>Python and Ruby</u>
Python and Ruby incorporate an object hierarchy in which classes themselves are represented by objects.
The root class in Python is called object; in Ruby it is Object.

In both Python and Ruby, each class has a single distinguished constructor, which cannot be overloaded.
In Python it is __init__; in Ruby it is initialize.
To create a new object in Python one says my_object = My_class(*args*).
In Ruby one says my_object = My_class.new(*args*).
Both Python and Ruby are more flexible than PHP or more traditional object oriented languages regarding the contents (members) of a class.
New fields can be added to a Python object simply by assigning to them: my_object.new_field= value.
Python and Ruby differ in many other ways.
The initial parameter to methods is explicit in Python; by convention it is usually named self.
In Ruby self is a keyword, and the parameter it represents is invisible.
Ruby methods may be public, protected, or private.
Access control in Python is purely a matter of convention; both methods and fields are universally accessible.