PROGRAMMING PARADIGMS

MODULE-5

TOPICS:

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

1. Object-Oriented Programming

- Three fundamental concepts of Object-oriented programming – Encapsulation, Inheritance and dynamic method binding
- Control or process abstraction is a very old concept
- Data abstraction is somewhat newer, though its roots can be found in ***simula67***
- Simula was weak in data hiding part of encapsulation
- Inheritance and dynamic method binding were adopted and refined in Smalltalk

Daa Abstraction

- An **Abstract Data Type** is one that is defined in terms of the operations that it supports (i.e., that can be performed upon it) rather than in terms of its structure or implementation
- *Statics* allow a subroutine to retain values from one invocation to the next, while hiding the name in-between
- *Modules* allow a collection of subroutines to share some statics, still with hiding – If you want to build an abstract data type, though, you have to make the module a manager

Need of Data Abstraction

- easier to think about - hide what doesn't matter
- protection - prevent access to things you shouldn't see
- plug compatibility
    - replacement of pieces, often without recompilation, definitely without rewriting libraries
    - division of labor in software projects

Features of Object-Oriented Programming

• Concept of ***objects*** & ***classes***

- Objects add inheritance and dynamic method binding
- ***Simula 67*** introduced these, but *didn't have data hiding*
- The 3 key factors in OO programming
    – Encapsulation (data hiding)

    – Inheritance

    – Dynamic method binding

- Enables code reuse by making it easy to define new abstractions as *extensions* or *refinements* of existing abstractions
- Classes have **data members** and **subroutine members(Member functions)**
- Constructors – creation of objects – have same name of class (C++,C#,Java)
- Destructors – Destruction of objects – have same name of class but with a leading tilde(~) – *used for storage management & error checking.*
- Access specifiers – Public, Private, Protected
- Inheritance
- Constructor overloading etc.

Encapsulation and Inheritance

Encapsulation–grouping of data and its subroutines that operate on them together in one place, and to hide irrelevant details from the users.

•Examples include procedures, packages, and classes.

•By utilizing this mechanism, the program can limit the scope and visibility of the data values and encapsulated functions for this newly defined data type. Thus, we have the notion of an *abstract data type.*

•**Inheritance**–Helps in code reuse.

•A class can be declared as a ***subclass*** of another class, which is called the ***parent class*** or ***superclass***. Within this hierarchy, each subclass is able to inherit variables and methods from its parent simply by virtue of the fact that it is a subclass.

Modules

Clu,Euclid,Modula,& other module-based languages have scope rules for data hiding

•Module declaration –Header

•Module definition –Body

•**M** is a Euclid module which exports a type **T,** then by default the remainder of the program can do nothing with objects of type **T** other than pass them to subroutines exported from **M**. Then **T** is an **opaque type**

- Ada allows the headers & bodies of modules to be separate. That is by dividing the header of a package into public & private parts.

• A type is exported ***opaquely*** by putting its definition in the private part of the header and by just 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;
```

• Change to module body –no need of recompiling the module's users

• Change to private part of header –need of recompiling the module's users. But no need to change their code.

•Change to public part of header -need for changing the user's code.

•Visibility rules

•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 { ...

•Here public members of the base class act like protected members of the derived class

Visibility rules of C++

•Any class can limit the visibility of its members.

•A derived class can restrict the visibility of members of a base class, but can never increase it.

Base class members	Derived class members
Private	Not visible
Protected & Public members of public base class	Protected & Public members
Protected & Public members of protected base class	Protected members
Protected & Public members of private base class	Private members

•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 by inserting a "**using"** declaration in the **protected** or **public** portion of the derived class declaration.

Case of other programming languages

•**Eiffel** is more flexible than C++, but not adheres to the first principle of C++

•Derived class can restrict and increase the visibility of the members of base classes

•Each method has its own *export status.*

•Status –**{NONE}** –member is effectively private(secret in Eiffel)

•Status –**{ANY}** –member is effectively public(generally available in Eiffel)

•Java & C# follow C++ in public, protected and private

•Does not provide protected and private designations for base classes.

•**Protected in Java** -a protected member of a Java class is visible not only within derived classes, but also within the entire package (namespace) in which the class is declared.

•**C# defines protected** as C++ does, but provides an additional "**internal"** keyword that makes a member visible throughout the "*assembly"* in which the class appears.

•Members of a C# class are private by default.

Nesting (Inner Classes)

Issue with nesting or inner classes:

if Inner is a member of Outer, can Inner's methods see Outer's members?

• In c++ and c# allow access to only the static members of the outer class,

•In effect, nesting serves simply as a means of information hiding.

•In Java 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() {
        n = 0;
        System.out.println(n);      // prints 0
        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.

•Inner and local classes in Java are widely used to create *object closures*

Extending without Inheritance

Case: The class one wants to extend may not permit inheritance
•for instance: in Java, it may be labeled **final**; in C#, it may be **sealed**.

•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);
    }
}
```

Rules:
•An extension method **must be static**, and **must be declared in a static class.**

•Its **first parameter must be prefixed with the keyword "this"**.

•The method can then be invoked as if it were a member of the class of which this is an instance:

```
int n = myString.toInt();
```

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

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

Constructors and Destructors - Issues

1. choosing a constructor
2. references and values
   –If variables are references, then every object must be created explicitly - appropriate constructor is called
   –If variables are values, then object creation can happen implicitly as a result of elaboration
3. execution order
   –When an object of a derived class is created in C++, the constructors for any base classes will be executed before the constructor for the derived class
4. garbage collection

choosing a constructor

•Smalltalk, Eiffel, C++, Java, and C# -can have more than one constructor for a given class.

•Smalltalk, Eiffel –
- •different constructors have different names.
- •code that creates an object must name a constructor explicitly

•Modula-3 and Oberon provide no constructors at all

•Ada 95 supports constructors and destructors (called Initialize and Finalize routines)

References and Values

•Simula, Smalltalk, Python, Ruby, and Java -variables refer to objects.

•C++, Modula-3, Ada 95, and Oberon, allow a variable to have a value that *is* an object.

•C# uses **struct** to define types whose variables are values, and **class** to define types whose variables are references.

References and Values –Examples

```
foo b;                  // calls foo::foo()


foo b(10, 'x');         // calls foo::foo(int, char)

foo a;
bar b;
...
foo c(a);               // calls foo::foo(foo&)
foo d(b);               // calls foo::foo(bar&)
```

- Usually the programmer's intent is to declare a new object whose initial value is "the same" as that of the existing object
- Another Example :

```
foo a;                  // calls foo::foo()
bar b;                  // calls bar::bar()
...
foo c = a;              // calls foo::foo(foo&)
foo d = b;              // calls foo::foo(bar&)
```

  - 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 –C++

• if the object's class (call it *B*) is derived from some other class (call it *A*), C++ insists on calling an *A* constructor before calling a *B* constructor,

- **Issue** -where does the compiler obtain arguments for the *A* constructor?
- **Solution** - C++ is to allow the header of the constructor of a derived class to specify base class constructor arguments:

```
foo::foo( foo_params ) : bar( bar_args ) {
    ...
```

Execution Order –Java

•Java insists that a constructor for a base class be called before the constructor for a derived class.

•the initial line of the code for the derived class constructor may consist of a "call" to the base class constructor:

```
super( args );
```

•super is a Java keyword that refers to the base class of the class in whose code it appears.

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.

```
class name_list_node : public gp_list_node {
    char *name;             // pointer to the data in a node
public:
    name_list_node() {
        name = 0;           // empty string
    }
    name_list_node(char *n) {
        name = new char[strlen(n)+1];
        strcpy(name, n);    // copy argument into member
    }
    ~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.
- Java and C# allow the programmer to declare a ***finalize*** method that will be called immediately before the garbage collector reclaims the space for an object.

Aliasing

•Two or more names that refer to the same object at the same point in the program are said to be *aliases*.

•Examples: -

•common blocks and equivalence statements of Fortran

•variant records and unions of languages like Pascal and C

•also arise naturally in programs that make use of pointer-based data structures.

How to create aliases?

Solution: pass a variable by reference to a subroutine that also accesses that variable directly.

- Example –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.

Drawbacks:

- Aliases tend to make programs more confusing
- Aliases make it much more difficult for a compiler to perform certain important code improvements

Overloading

•A name that can refer to more than one object at a given point in the program is said to be *overloaded*.

•Examples:

- •In C –the plus sign (+) is used to name several different functions, including signed and unsigned integer and floating-point addition.
- •enumeration constants of Ada.

```
declare
    type month is (jan, feb, mar, apr, may, jun,
                   jul, aug, sep, 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
```

- The constants **oct** and **dec** refer either to months or to numeric bases, depending on the context in which they appear.
- Most languages that allow overloaded enumeration constants allow the programmer to provide appropriate context explicitly.
- Example –Ada

```
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 C, C++, and standard Pascal, one cannot overload enumeration constants at all; every constant visible in a given scope must be distinct.

Redefining Built-in Operators

•Ada, C++, C#, Fortran 90, and Haskellalso allow the built-in arithmetic operators (+, -, *, etc) to be overloaded with user-defined functions.

•Ada, C++, and C# do this by defining alternative *prefix* forms of each operator

•In Ada, A + B is short for "+"(A, B). If "+" is overloaded, it must be possible to determine the intended meaning from the types of A and B.

Polymorphism

•Example:

•Ada –

```
function min(a, b : integer) return integer is ...
function min(x, y : real) return real is ...
```

•C –

```
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

•it allows a single subroutine to accept *unconverted* arguments of multiple types.

•It is applied to code—both data structures and subroutines—that can work with values 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*. (Ada, C++, Clu, Eiffel, Modula-3, Java, and C#,etc.)

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

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*

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


student s;
professor p;
...
person *x = &s;
person *y = &p;
```

•A polymorphic subroutine:

```
void person::print_mailing_label() { ...
s.print_mailing_label();    // i.e., print_mailing_label(s)
p.print_mailing_label();    // i.e., print_mailing_label(p)
```

Here it use the type of the reference ⟶static method binding

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

Here it use the class of the object⟶ dynamic method binding

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

•Java and Eiffel use dynamic method binding by default, but allow individual methods and (in Java) classes to be labeled final (Java) or frozen (Eiffel), in which case they cannot be overridden by derived classes, and can therefore employ an optimized implementation

•Simula, C++, C#, and Ada 95 use static method binding by default, but allow the programmer to specify dynamic binding when desired.

Virtual and Non-virtual Methods

•Simula, C++, C#, and Ada 95 use static method binding by default, but allow the programmer to specify dynamic binding when desired 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

•Ex: C++ & C# -the keyword virtual prefixes the subroutine declaration

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

•Ex: In Simula, virtual methods are listed at the beginning of the class declaration:

```
CLASS Person;
    VIRTUAL: PROCEDURE PrintMailing
BEGIN
    ...
    PROCEDURE PrintMailingLabel...
        COMMENT body of subroutine
    ...
END Person;
```

•Ex: Ada 95 associates it with certain references

- •a formal parameter or an access variable (pointer) can be declared to be of the *class-wide* type in which case 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();
    ...
```

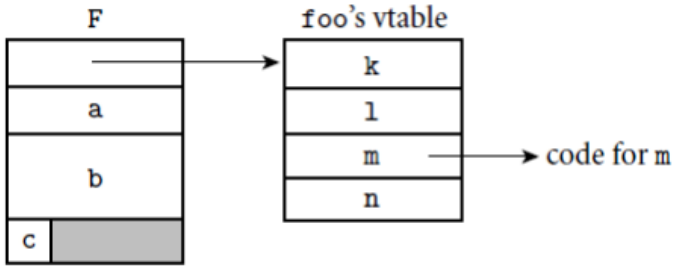
in C++ one follows the subroutine declaration with an “assignment” to zero

```
class person {
    ...
public:
    virtual void print_mailing_label() = 0;
    ...
```

Member Lookup

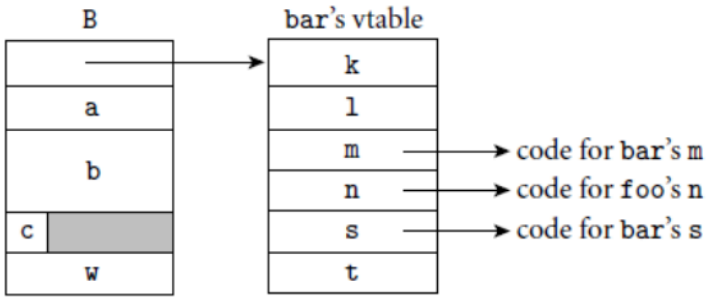
•The most common implementation represents each object with a record whose first field contains the address of a *virtual method table* (*vtable*) for the object's class. The vtable is an array whose *i*th entry indicates the address of the code for the object's *i*th virtual method. All objects of a given class share the same vtable.

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



•If bar is derived from foo, we place its additional fields at the end of the "record" that represents it. We create a vtable for bar by copying the vtable for foo, replacing the entries of any virtual methods overridden by bar, and appending entries for any virtual methods declared in bar

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



Ex:

•C++ uses dynamic_castoperator for backward assignments

```
s = dynamic_cast<bar*>(q);      // performs a run-time check
```

•C++ also supports traditional C-style casts of object pointers and references

```
s = (bar*) q;                   // permitted, but risky
```

•Eiffel has a *reverse assignment* operator, ?=, which (like the C++ dynamic_cast) assigns an object reference into a variable if and only if the type at run time is acceptable:

```
class foo ...
class bar inherit foo ...
...
f : foo
b : bar
...
f := b    -- always ok
b ?= f    -- reverse assignment: b gets f if f refers to a bar object
          -- at run time; otherwise b gets void
```

Multiple Inheritance

•derived class to inherit features from more than one base class.

•Ex: administrative computing system to keep all students of the same year (freshmen, sophomores, juniors, seniors, non matriculated) on some list. It may then be desirable to derive class student from both person and gp_list_node.

In C++

```
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.

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, and Oberon 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

Innovative Features of Scripting Languages

- Conventional languages tend to stress efficiency, maintainability, portability, and the static detection of errors.
- Their type systems tend to be built around such hardware-level concepts as fixed size integers, floating-point numbers, characters, and arrays.
- *Scripting languages* tend to stress flexibility, rapid development, local customization, and dynamic (run-time) checking.

- Their type systems, likewise, tend to embrace such high-level concepts as tables, patterns, lists, and files.
- They were originally designed to **—glue‖** existing programs together to build a larger system.
- Ex: general-purpose scripting languages like **Perl** and **Python**
- Modern scripting languages have two principal sets of ancestors.
    - command interpreters or " shells" of traditional batch and " terminal" (command-line) computing
        - IBM's JCL, MS-DOS command interpreter, Unix sh and csh
    - various tools for text processing and report generation
        - IBM's RPG, and Unix's sed and awk.

Common Characteristics:

- Both batch and interactive use
    - While a few languages (e.g. Perl) have a compiler that requires the entire source program, almost all scripting languages either compile or interpret line by line
    - Many "compiled" versions are actually completely equivalent to the interpreter running behind the scenes (like in Python).
- Economy of expression
    - Two variants: some make heavy use of punctuation and short identifiers (like Perl), while others emphasize "English-like" functionality

Either way, things get shorter. Java versus Python (or Ruby or Perl):

```
class Hello {
        public static void main(String[] args) {
        System.out.println("Hello, world!");
}
```

Output `print "Hello, world!"`

- Lack of declarations; simple scoping rules.
    - While the rules vary, they are generally fairly simple and additional syntax is necessary to alter them.
    - In **Perl**, everything is of global scope by default, but optional parameters can limit the scope to local
    - In **PHP**, everything is local by default, and any global variables must be explicitly imported.

    - In **Python**, everything is local to the block in which the assignment appears, and special syntax is required to assign a variable in a surrounding scope.
- Flexible dynamic typing
    - In PHP, Python and Ruby, the type of a variable is only checked right before use
    - In Perl, Rexx, or Tcl, things are even more dynamic:

```
$a = "4"
print $a . 3 . "\n"
print $a + 3 . "\n"
```

      } Perl

      Outputs the following:
      43
      7
- Easy access to system facilities
    - While all languages provide support for OS functionality, scripting languages generally provide amazing and much more fundamental built-in support.
    - Examples include directory and file manipulation, I/O modules, sockets, database access, password and authentication support, and network communications.
- Sophisticated pattern matching and string manipulation
    - Perl is perhaps the master of this, but it traces back to the text processing sed/awk ancestry.
    - These are generally based on *extended regular expression*
- High level data types
    - In general, scripting languages provide support for sets, dictionaries, lists and tuples (at a minimum).
    - While languages like C++ and Java have these, they usually need to be imported separately.
    - Behind the scenes, optimizations like arrays indexed using hash tables are quite common.
    - Garbage collection is always automatic, so user never has to deal with heap/stack issues.

Scope and Names

- Most scripting languages (Scheme is the obvious exception) do not require variables to be declared

- Perl and JavaScript permit optional declarations - sort of compiler-checked documentation
- Perl can be run in a mode (use strict 'vars') that requires declarations

- With or without declarations, most scripting languages use dynamic typing
  - The interpreter can perform type checking at run time, or coerce values when appropriate
  - Tcl is unusual in that all values—even lists—are represented internally as strings
- Nesting and scoping conventions vary quite a bit
  - Scheme, Python, JavaScript provide the classic combination of nested subroutines and static (lexical) scope
  - Tcl allows subroutines to nest, but uses dynamic scope
  - Named subroutines (methods) do not nest in PHP or Ruby
    - Perl and Ruby join Scheme, Python, and JavaScript in providing first class anonymous local subroutines
  - Nested blocks are statically scoped in Perl
  - In Ruby, they are part of the named scope in which they appear
  - Scheme, Perl, Python provide for variables captured in closures
  - PHP and the major glue languages (Perl, Tcl, Python, Ruby) all have sophisticated namespace rules
    - mechanisms for information hiding and the selective import of names from separate modules
- What is the scope of undeclared variable?
  - In Perl, all variables are global unless otherwise specified.
  - In PHP, local unless explicitly imported.
  - Ruby has only two levels: $foo is global, foo is local; @foo is instance of current object, and @@foo is instance variable of current object's class
  - In Python, all variables are local by default, unless explicitly imported:
    i=1; j=3
    def outer( )
- Scope in Python
  - In Python, all variables are local by default, unless explicitly imported:

```
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
    return middle(j)        #old (global) j
print outer( )
print i,j
```

- This prints: (2,3,3)
  4 3

- Scope in Python
  - By default, there is no way for a nested scope to write to a non-local or non-global scope - so in previous example, inner could not modify outer's i variable.
- R has an interesting convention:
  - **Normal assignment** puts value into the local variable:
    i <- 4
  - **Super assignment** puts value into whatever variable would be found under normal (static) scoping rules:
    i <<- 4
- Tcl uses dynamic scoping, but in an odd way - the programmer must request other scopes explicitly:

```
upvar i j ;                    #j is the local name for caller‟s I
uplevel 2     {puts [expr $a + $b] }
        #executes 'puts' two scopes up on dynamic chain
```

Pattern matching

- Regular expressions are present in many scripting languages and related tools employ extended versions of the notation
  - extended regular expressions – in sed and awk, Perl, Tcl, Python, and Ruby
  - grep, the stand-alone Unix is a pattern-matching tool, is another useful program that you might be familiar with
- In general, two main groups.
  - The first group includes awk, egrep, the regex routines of the C standard library, 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**"

Pattern matching: POSIX Res

- Basic operations are familiar:
  /ab(cd|ef)g*/ - Matches abcd, abcdg, abefg, abefgg, etc.
- Other quantifiers:
  - **? :** 0 or 1 repetitions
  - **+ :** 1 or more repetitions
  - **{n} :** exactly n repetitions
  - **{n,} :** at least n repetitions
  - **{n,m} :** between n and m repetitions
  - ^ and **$** force the match to be at the beginning or end of the line
  - Brackets can indicate a character class: **[**aeiou**]** - any vowel
  - Ranges: [0-9]
  - A dot **.** matches any single character
  - ^ before a character class is negation

Pattern matching: extended REs

- Perl adds on to this extensively.
  - Example:
    $_ = "albatross";
    if (/ba.*s+/) … #true
    if (/^ba.*s+/) … #false - no match at start

  - **=~** tests if it matches, **!~** tests if it does not (or defaults to checking against $_, if not specified)
  - Substitution is done by **s///**:
    $foo = "albatross";
    $foo =~ s/lbat/c; #now across
- Variations on normal REs:
  - Trailing i makes the match case insensitive.
    $foo = "Albatross";
    if ($foo =~ /^al/i) … #true
  - Trailing g will replace all occurances.
    $foo = "albatross";
    $foo =~ s/[aeiou]/-/g … # "-lb-tr-ss"
- Trailing **x** causes Perl to ignore all comments and embedded white space in the pattern, so that you can break up long patterns into multiple lines.

Pattern matching: greedy matches

- If multiple matches are possible, it will take the ―left-most longest‖ possible one. For example, in the string abcbcbcde, the pattern /(bc)+/ will match abcbcbcde.
- This is known as the greedy match.
- Other options:
    - ***?** matches the smallest number of instances of the preceding subexpression that will allow it to succeed.
    - **+?** matches at least one instance, but no more than necessary
    - **??** matches either 0 or 1 instance, with a preference for 0

Data Types

- As we have seen, scripting languages don't generally require (or even permit) the declaration of types for variables
- Most perform extensive run-time checks to make sure that values are never used in inappropriate ways
- Some languages (e.g., Scheme, Python, and Ruby) are relatively strict about this checking
    - When the programmer wants to convert from one type to another, it must say so explicitly
- Perl (and likewise Rexx and Tcl) takes the position that programmers should check for the errors they care about
    - in the absence of such checks the program should do something reasonable
- Numeric types have a bit more variation across languages, but emphasis is universally that the programmer shouldn't worry about the issue unless necessary.
- Won't say too much here, except be cautious about arithmetic if it matters to your program.
- Some of these even store numbers as strings, so calculations may not always be what you expect, although most do a good job of auto-converting if needed.
- For composite types, a heavy emphasis is on mappings (also called dictionaries, hashes, or associated arrays).
    - Generally these are similar to arrays, but access time depends upon a hash function.
    - Example:

      director = {}
      director["Star Wars"] = "George Lucas"

director[“The Princess Bride”] = “Rob Reiner”
print director[“Star Wars”]
print “Buffy” in director

- Behind the scenes, this is actually using a hash function. Still O(1) access time (mostly), but the constant is not nearly as fast as normal array access.

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
    - both allow the programmer to use a more traditional imperative style
- Python and Ruby are explicitly and uniformly object-oriented
- 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.
    - In contrast to Perl, however, these languages provide no way to speak of the reference itself, only the object to which it refers
- Python and Ruby use a uniform reference model
- Classes are themselves objects in Python and Ruby, much as they are in Smalltalk
- They are types in PHP, much as they are in C++, Java, or C#
- Classes in Perl are simply an alternative way of looking at packages (namespaces)
- JavaScript, remarkably, has objects but no classes
    - its inheritance is based on a concept known as *prototypes*
- Both PHP and JavaScript are more explicitly object oriented