Object Oriented Thinking:

Everywhere you look in the real world you see objects—people, animals, plants, cars, planes, buildings, computers and so on. Humans think in terms of objects. Telephones, houses, traffic lights, microwave ovens and water coolers are just a few more objects. Computer programs, such as the Java programs you'll read in this book and the ones you'll write, are composed of lots of interacting software objects.

We sometimes divide objects into two categories: animate and inanimate. Animate objects are "alive" in some sense—they move around and do things. Inanimate objects, on the other hand, do not move on their own .Objects of both types, however, have some things in common. They all have attributes (e.g., size, shape, color and weight), and they all exhibit behaviors (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleep crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water). We will study the kinds of attributes and behaviors that software objects have. Humans learn about existing objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults and between humans and chimpanzees. Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes and behaviors just as we describe real-world objects. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages. A bank account object may receive a message to decrease its balance by a certain amount because the customer has withdrawn that amount of money.

Object-Oriented:

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance nonobjects.

OOD encapsulates (i.e., wraps) attributes and operations (behaviors) into objects, an object's attributes and operations are intimately tied together. Objects have the property of information hiding. This means that objects may know how to communicate with one another across well-defined interfaces, but normally they are not allowed to know how other objects are implemented ,implementation details are hidden within the objects themselves. We can drive a car effectively, for instance, without knowing the details of how engines, transmissions, brakes and exhaust systems work internally—as long as we know how to use the accelerator pedal, the brake pedal, the wheel and so on. Information hiding, as we will see, is crucial to good software engineering.

Languages like Java are object oriented. Programming in such a language is called object-oriented programming (OOP), and it allows computer programmers to implement an object-oriented design as a working system. Languages like C, on the other hand, are procedural, so programming tends to be action oriented. In C, the unit of programming is the function. Groups of actions that perform some common task are formed into functions, and functions are grouped to form programs. In Java, the unit of programming is the class from which objects are eventually instantiated (created). Java classes contain methods (which implement operations and are similar to functions in C) as well as fields (which implement attributes).

Java programmers concentrate on creating classes. Each class contains fields, and the set of methods that manipulate the fields and provide services to clients (i.e., other classes that use the class). The programmer uses existing classes as the building blocks for constructing new classes. Classes are to objects as blueprints are to houses. Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class.

Classes can have relationships with other classes. For example, in an object-oriented design of a bank, the "bank teller" class needs to relate to the "customer" class, the "cash drawer" class, the "safe" class, and so on. These relationships are called associations.

Packaging software as classes makes it possible for future software systems to reuse the classes. Groups of related classes are often packaged as reusable components. Just as realtors often say that the three most important factors affecting the price of real estate are "location, location and location," people in the software community often say that the three most important factors affecting the future of software development are "reuse, reuse and reuse." Reuse of existing classes when building new classes and programs saves time and effort. Reuse also helps programmers build more reliable and effective systems, because existing classes and components often have gone through extensive testing, debugging and performance tuning.

Indeed, with object technology, you can build much of the software you will need by combining classes, just as automobile manufacturers combine interchangeable parts. Each new class you create will have the potential to become a valuable software asset that you and other programmers can use to speed and enhance the quality of future software development efforts.

NEED FOR OOP PARADIGM:

Object-Oriented Programming:

Object-oriented programming is at the core of Java. In fact, all Java programs are object-oriented—this isn't an option the way that it is in C++, for example. OOP is so integral to Java. Therefore, this chapter begins with a discussion of the theoretical aspects of OOP.

Two Paradigms of Programming:

As you know, all computer programs consist of two elements: code and data. Furthermore,a program can be conceptually organized around its code or around its data. That is, some programs are written around "what is happening" and others are written around "who is being affected." These are the two paradigms that govern how a program is constructed.

The first way is called the process-oriented model. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success. Problems with this approach appear as programs grow larger and more complex. To manage increasing complexity, the second approach, called object-oriented programming, was conceived.

Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

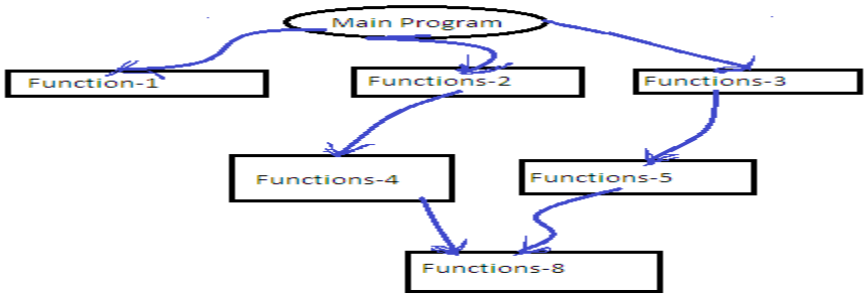
Procedure oriented Programming:

In this approach, the problem is always considered as a sequence of tasks to be done. A number of functions are written to accomplish these tasks. Here primary focus on "Functions" and little attention on data.

There are many high level languages like COBOL, FORTRAN, PASCAL, C used for conventional programming commonly known as POP.

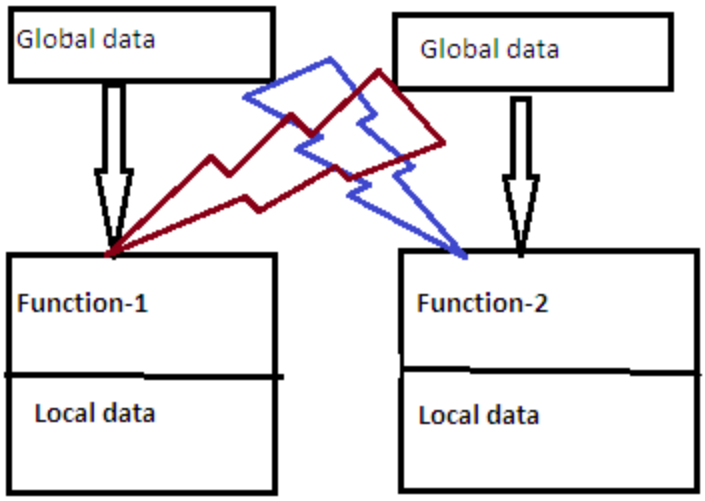
POP basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as functions.

A typical POP structure is shown in below:



Normally a flowchart is used to organize these actions and represent the flow of control logically sequential flow from one to another. In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Global data are more vulnerable to an in advent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we should also revise all the functions that access the data. This provides an opportunity for bugs to creep in.

Drawback: It does not model real world problems very well, because functions are action oriented and do not really corresponding to the elements of the problem.



Chars of POP:

- Emphasis is on doing actions.
- Large programs are divided into smaller programs known as functions.
- Most of the functions shared global data.
- Data move openly around the program from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

OOP:

OOP allows us to decompose a problem into a number of entities called objects and then builds data and methods around these entities.

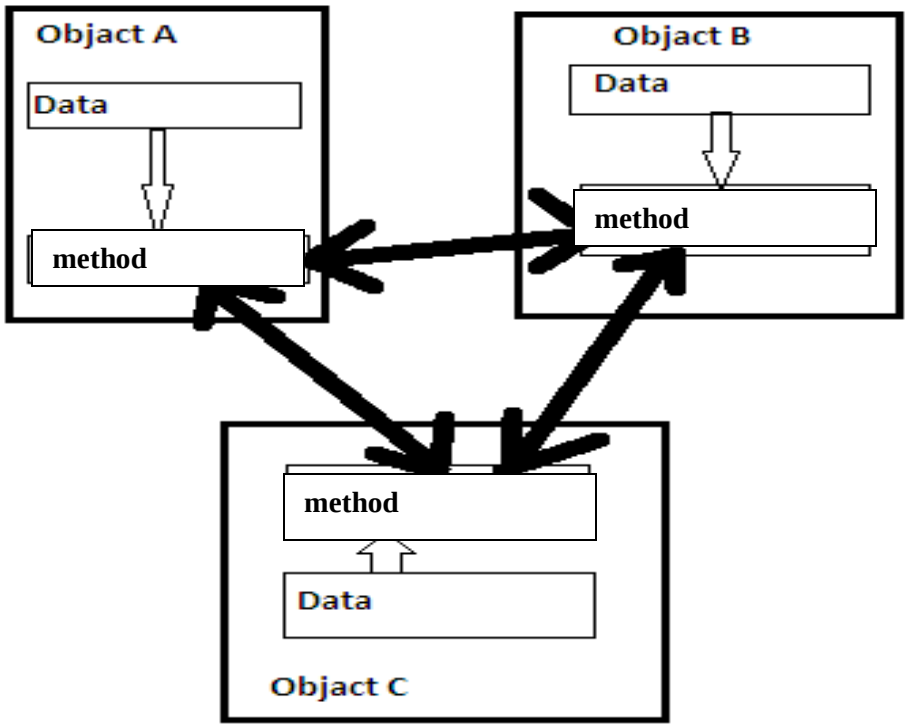
DEF: OOP is an approach that provides a way of modularizing programs by creating portioned memory area for both data and methods that can used as templates for creating copies of such modules on demand.

That is ,an object a considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

OOP Chars:

- Emphasis on data .
- Programs are divided into what are known as methods.
- Data structures are designed such that they characterize the objects.
- Methods that operate on the data of an object are tied together .
- Data is hidden.
- Objects can communicate with each other through methods.
- Reusability.
- Follows bottom-up approach in program design.

Organization of OOP:



Evolution of Computing and Programming: Computer use is increasing in almost every field of endeavor. Computing costs have been decreasing dramatically due to rapid developments in both hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars decades ago can now be inscribed on silicon chips smaller than a fingernail, costing perhaps a few dollars each. Fortunately, silicon is one of the most abundant materials on earth it is an ingredient in common sand. Silicon chip technology has made computing so economical that about a billion general-purpose computers are in use worldwide, helping people in business, industry and government, and in their personal lives. The number could easily double in the next few years. Over the years, many programmers learned the programming methodology called structured programming.

You will learn structured programming and an exciting newer methodology, object-oriented programming. Why do we teach both? Object orientation is the key programming methodology used by programmers today. You will create and work with many software objects in this text. But you will discover that their internal structure is often built using structured-programming techniques. Also, the logic of manipulating objects is occasionally expressed with structured programming.

Language of Choice for Networked Applications: Java has become the language of choice for implementing Internet-based applications and software for devices that communicate over a network. Stereos and other devices in homes are now being networked together by Java technology. At the May 2006 JavaOne conference, Sun announced that there were one billion java-enabled mobile phones and hand held devices! Java has evolved rapidly into the large-scale applications arena. It's the preferred language for meeting many organizations' enterprise-wide programming needs. Java has evolved so rapidly that this seventh edition of Java How to Program was published just 10 years after the first edition was published. Java has grown so large that it has two other editions. The Java Enterprise Edition (Java EE) is geared toward developing large-scale, distributed networking applications and web-based applications. The Java Micro Edition (Java ME) is geared toward developing applications for small, memory constrained devices, such as cell phones, pagers and PDAs.

Security:

As you are likely aware, every time that you download a "normal" program, you are risking a viral infection. Prior to Java, most users did not download executable programs frequently, and those who did scanned them for viruses prior to execution. Even so, most users still worried about the possibility of infecting their systems with a virus. In addition to viruses, another type of malicious program exists that must be guarded against. This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. Java answers both of these concerns by providing a "firewall" between a networked application and your computer. When you use a Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer. (You will see how this is accomplished shortly.).The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most important aspect of Java.

Portability

As discussed earlier, many types of computers and operating systems are in use throughout the world—and many are connected to the Internet. For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. As you will soon see, the same mechanism that helps ensure security also helps create portability. Indeed, Java's solution to these two problems is both elegant and efficient.

A way of viewing World-Agents and Responsibility:

Object-oriented languages use objects to encapsulate the details. Each object simulates an object in real life, carrying state data as well as behaviors. State data are represented as ***instance data***. Behaviors are represented as ***methods***.

> Java programmers concentrate on creating classes. Each class contains fields, and the set of methods that manipulate the fields and provide services to clients (i.e., other classes that use the class). The programmer uses existing classes as the building blocks for constructing new classes.

OOP establish the communication between Distributed Objects. It is possible with the help of clients and servers in the form of agents. It helps for Distributed Applications.

OOP Chars with Distributed Applications:

Interprocess Communication: A distributed application require the participation of two or more independent entities. To do so, the processes must have the ability to exchange data among themselves.

Event Synchronization: In a distributed application, the sending and receiving of data among the participants of a distributed application must be synchronized.

- A *distributed system* is a collection of independent computers, interconnected via a network, capable of collaborating on a task.
- *Distributed computing* is computing performed in a distributed system.

Peer-to-Peer distributed computing: The peer-to-peer paradigm can be implemented with facilities using any tool that provide message-passing, or with a higher-level tool such as one that supports the point-to-point model of the Message System paradigm.

For web applications, the ***web agent*** is a protocol promoted by the XNSORG (the XNS Public Trust Organization) for peer-to-peer interprocess communication

"Project JXTA is a set of open, generalized peer-to-peer protocols that allow any connected device (cell phone, to PDA, PC to server) on the network to communicate and collaborate. JXTA is short for Juxtapose, as in side by side. It is a recognition that peer to peer is juxtapose to client server or Web based computing -- what is considered today's traditional computing model. "

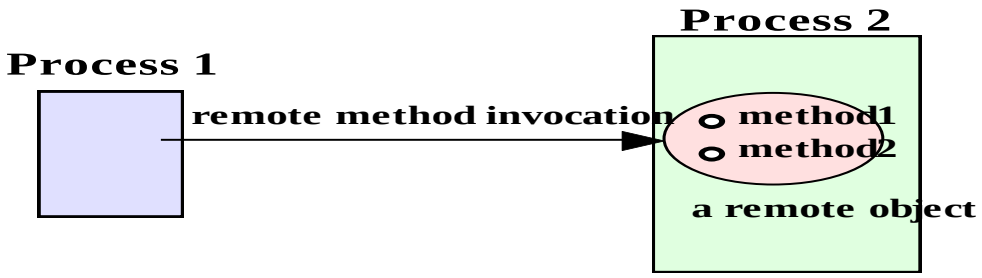
The Distributed Objects Paradigms

- The idea of applying object orientation to distributed applications is a natural extension of object-oriented software development.
- Applications access objects distributed over a network.
- Objects provide methods, through the invocation of which an application obtains access to services.

- Object-oriented paradigms include:
  - Remote method invocation (RMI)
  - Network services
  - Object request broker
  - Object spaces

Remote Method Invocation (RMI):

- Remote method invocation is the object-oriented equivalent of remote method calls.
- In this model, a process invokes the methods in an object, which may reside in a remote host.
- As with RPC, arguments may be passed with the invocation.



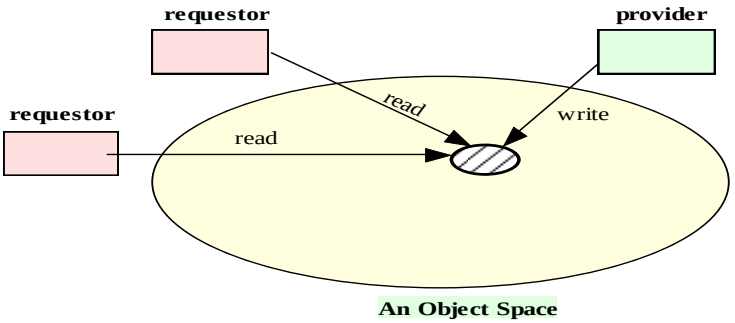
The Remote Method Call Paradigm

The Object Request broker Paradigm:

- In the object broker paradigm , an application issues requests to an *object request broker* (ORB), which directs the request to an appropriate object that provides the desired service.
- The paradigm closely resembles the remote method invocation model in its support for remote object access. The difference is that the object request broker in this paradigm functions as a middleware which allows an application, as an object requestor, to potentially access multiple remote (or local) objects.
- The request broker may also function as an mediator for heterogeneous objects, allowing interactions among objects implemented using different APIs and /or running on different platforms.

The Object Space Paradigm:

- Perhaps the most abstract of the object-oriented paradigms, the object space paradigm assumes the existence of logical entities known as *object spaces*.
- The participants of an application converge in a common object space.
- A provider places objects as entries into an object space, and requesters who subscribe to the space access the entries.



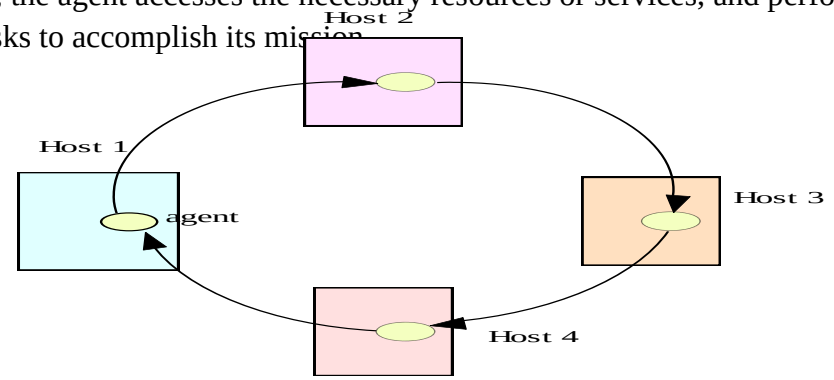
In addition to the abstractions provided by other paradigms, the object space paradigm provides a virtual space or meeting room among provides and requesters of network resources or objects. This abstraction hides the detail involved in resource or object lookup needed in paradigms such as remote method invocation, object request broker, or network services.

- Current facilities based on this paradigm include JavaSpaces

  http://java.sun.com/products/javaspaces/.

The Mobile Agent Paradigm

- A mobile agent is a transportable program or object.
- In this model, an agent is launched from an originating host.
- The agent travels from host to host according to an itinerary that it carries.
- At each stop, the agent accesses the necessary resources or services, and performs the necessary tasks to accomplish its mission.

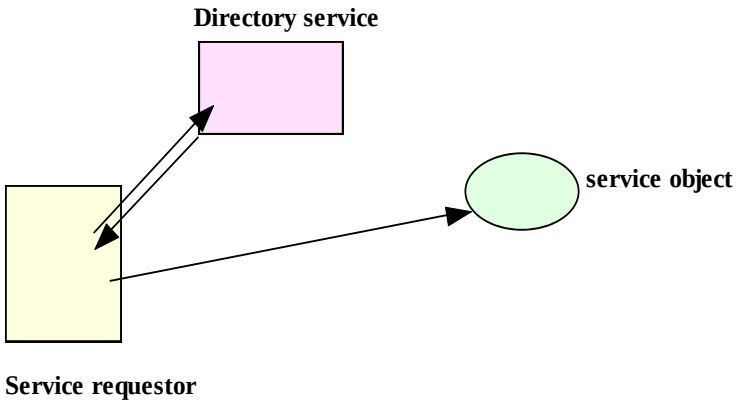


- The paradigm offers the abstraction for a transportable program or object.
- In lieu of message exchanges, data is carried by the program/object as the program is transported among the participants.

The Network Services Paradigm:

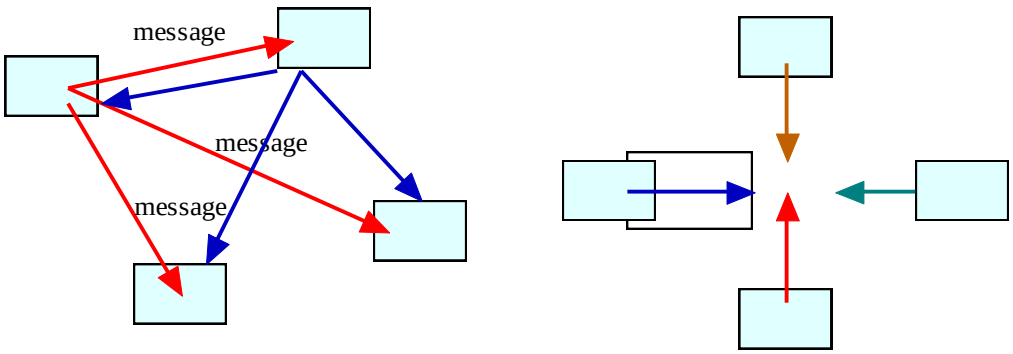
- In this paradigm, service providers register themselves with directory servers on a network. A process desiring a particular service contacts the directory server at run time, and, if the service is available, will be provided a reference to the service. Using the reference, the process interacts with the service.
- This paradigm is essentially an extension of the remote method call paradigm. The difference is that service objects are registered with a global directory service, allowing them to be look up and accessed by service requestors on a federated network.

Java's Jini technology is based on this paradigm



The Collaborative Application (Groupware) Paradigm:

- In this model, processes participate in a collaborative session as a group. Each participating process may contribute input to part or all of the group.
- Processes may do so using:
  - multicasting to send data to all or part of the group, or they may use a
  - virtual *sketchpads or whiteboards* which allows each participant to read and write data to a shared display.



The Architecture of Distributed Applications:

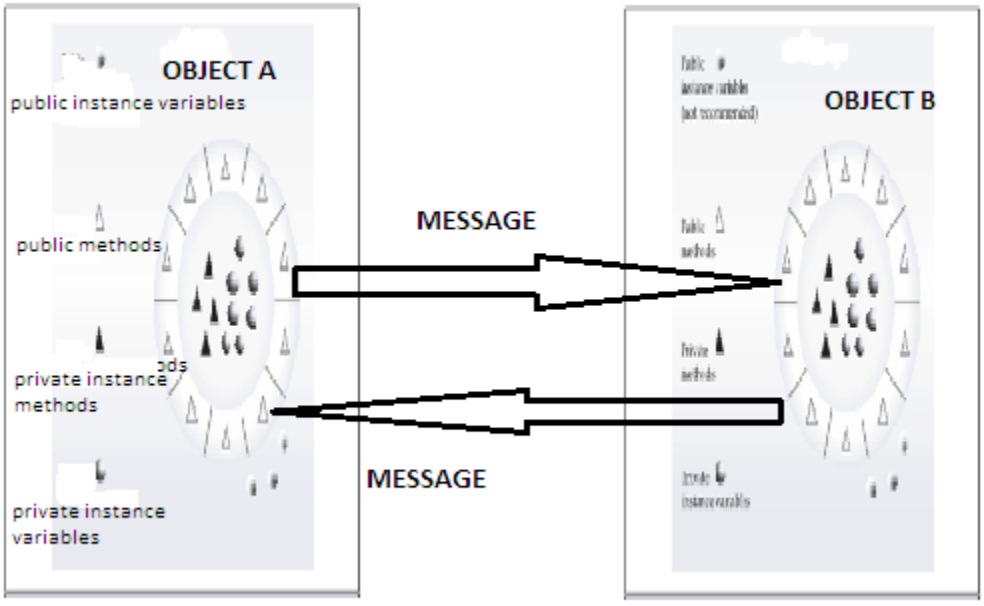
Presentation
Application (Business) logic
Services

MESSAGES:

An OOP consists of objects that communicate with each other. The process of programming in an object-Oriented language, therefore, involves the basic concepts:

1. Creating classes that define objects and their behavior.
2. Creating objects from class definitions.
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another.



The concept of message passing makes it easier to talk about building systems that directy model or simulate their real-world counter parts.

A message for an object is a request for execution of a procedure, and therefore will invoke a procedure in the receiving object that generates the desired results.

Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

Message passing involves specifying the name of the object, the name of the method and the information to be sent.

Example:

1. The object to which the message is addressed.(YourBicycles)
2. The name of the method to perform. (changeGears)
3. Any parameters needed by the method.(lowerGear)

Example 2: System.out.println("hi");

Just as you cannot drive an engineering drawing of a car, you cannot "drive" a class. Just as someone has to build a car from its engineering drawings before you can actually drive a car, you must build an object of a class before you can get a program to perform the tasks the class describes how to do. That is one reason Java is known as an object-oriented programming language.

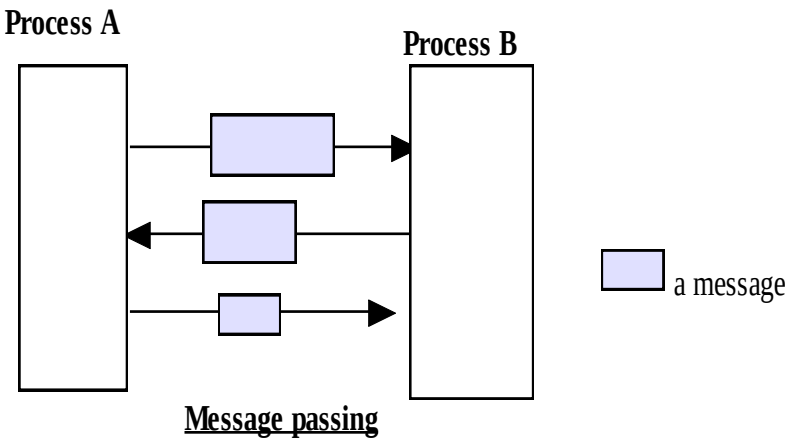
When you drive a car, pressing its gas pedal sends a message to the car to perform a task—that is, make the car go faster. Similarly, you send messages to an object—each message is known as a method call and tells a method of the object to perform its task.

The Message Passing Paradigm:

Message passing is the most fundamental paradigm for distributed applications.

- A process sends a message representing a request.
- The message is delivered to a receiver, which processes the request, and sends a message in response.

In turn, the reply may trigger a further request, which leads to a subsequent reply, and so forth.



- The basic operations required to support the basic message passing paradigm are *send*, and *receive*.
- For connection-oriented communication, the operations *connect* and *disconnect* are also required.
- With the abstraction provided by this model, the interconnected processes perform input and output to each other, in a manner similar to file I/O. The I/O operations encapsulate the detail of network communication at the operating-system level.
- The socket application programming interface is based on this paradigm.

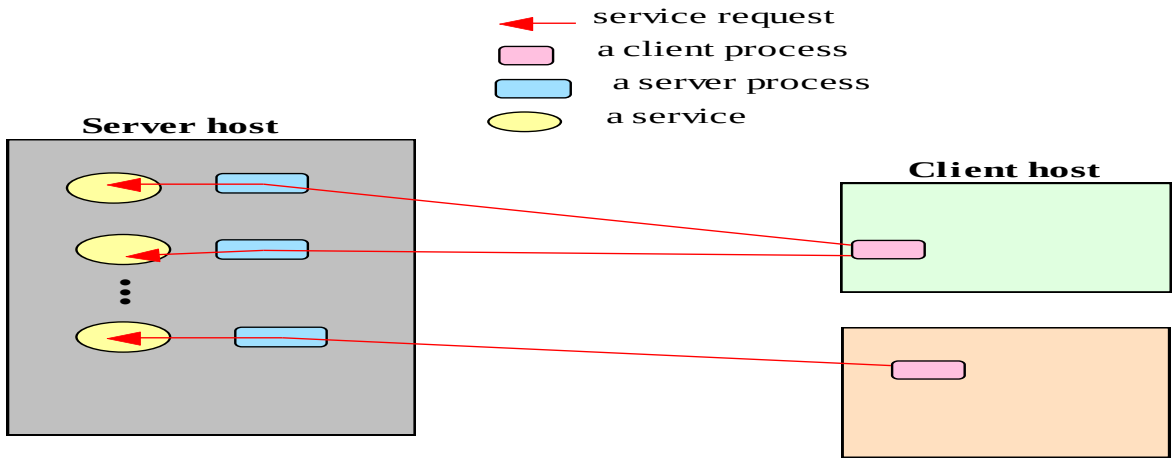
The Client-Server Paradigm:

Perhaps the best known paradigm for network applications, the client-server model assigns asymmetric roles to two collaborating processes.

One process, the server, plays the role of a service provider which waits passively for the arrival of requests. The other, the client, issues specific requests to the server and awaits its response.

- Simple in concept, the client-server model provides an efficient abstraction for the delivery of network services.
- Operations required include those for a server process to listen and to accept requests, and for a client process to issue requests and accept responses.
- By assigning asymmetric roles to the two sides, event synchronization is simplified: the server process waits for requests, and the client in turn waits for responses.

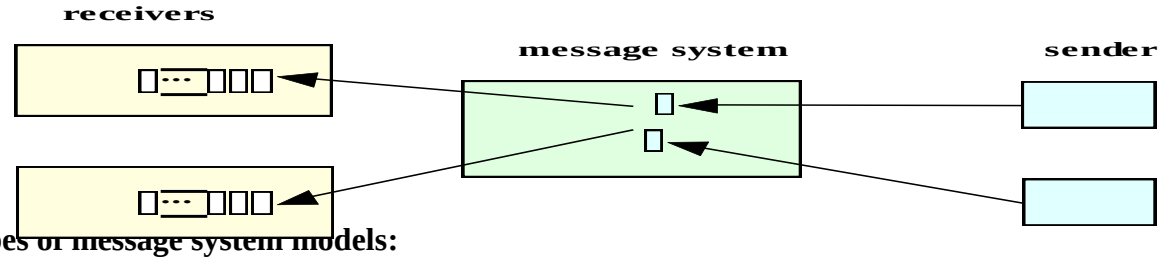
Many Internet services are client-server applications. These services are often known by the protocol that the application implements. Well known Internet services include HTTP, FTP, DNS, finger, gopher, etc.



The Client-Server Paradigm, conceptual

The Message System Paradigm:

- The Message System or Message-Oriented Middleware (MOM) paradigm is an elaboration of the basic message-passing paradigm.
- In this paradigm, a message system serves as an intermediary among separate, independent processes.
- The message system acts as a switch for messages, through which processes exchange messages asynchronously, in a decoupled manner.
- A sender deposits a message with the message system, which forwards it to a message queue associated with each receiver. Once a message is sent, the sender is free to move on to other tasks.



Two subtypes of message system models:

The Point-To-Point Message Model:

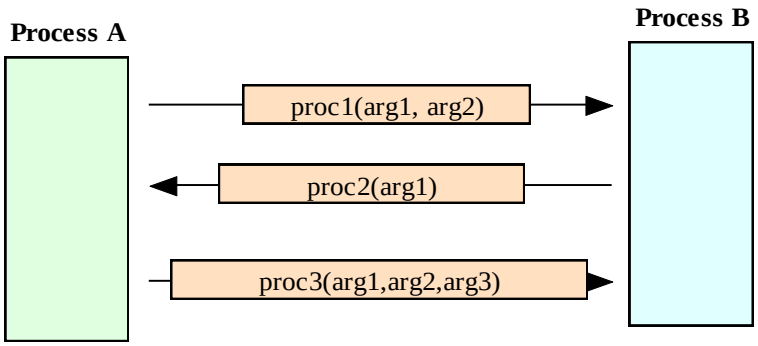
- In this model, a message system forwards a message from the sender to the receiver's message queue. Unlike the basic message passing model, the middleware provides a message depository, and allows the sending and the receiving to be decoupled. Via the middleware, a sender deposits a message in the message queue of the receiving process. A receiving process extracts the messages from its message queue, and handles each one accordingly.
- Compared to the basic message-passing model, this paradigm provides the additional abstraction for asynchronous operations. To achieve the same effect with basic message-passing, a developer will have to make use of threads or child processes.

The Publish/Subscribe Message Model:

- In this model, each message is associated with a specific topic or event. Applications interested in he occurrence of a specific event may subscribe to messages for that event. When the awaited event occurs, the process publishes a message announcing the event or topic. The middleware message system distributes the message to all its subscribers.
- The publish/subscribe message model offers a powerful abstraction for multicasting or group communication. The *publish* operation allows a process to multicast to a group of processes, and the *subscribe* operation allows a process to listen for such multicast.

Remote Procedure Call:

- As applications grew increasingly complex, it became desirable to have a paradigm which allows distributed software to be programmed in a manner similar to conventional applications which run on a single processor.
- The Remote Procedure Call (RPC) model provides such an abstraction. Using this model, interprocess communications proceed as procedure, or function, calls, which are familiar to application programmers.
- A remote procedure call involves two independent processes, which may reside on separate machines. A process, *A*, wishing to make a request to another process, *B*, issues a procedure call to *B*, passing with the call a list of argument values. As in the case of local procedure calls, a remote procedure call triggers a predefined action in a procedure provided by process *B*. At the completion of the procedure, process *B* returns a value to process *A*.
- RPC allows programmers to build network applications using a programming construct similar to the local procedure call, providing a convenient abstraction for both interprocess communication and event synchronization.
- Since its introduction in the early 1980s, the Remote Procedure Call model has been widely in use in network applications.
- There are two prevalent APIs for Remote Procedure Calls.
  - The *Open Network Computing Remote Procedure Call*, evolved from the RPC API originated from Sun Microsystems in the early 1980s.
  - The *Open Group Distributed Computing Environment* (DCE) RPC.
- Both APIs provide a tool, *rpcgen*, for transforming remote procedure calls to local procedure calls to the stub.



MESSAGE: SOFTWARE OBJECTS INTERACT AND COMMUNICATE WITH EACH OTHER USING MESSAGES.

METHODS:

Method is an action required by an object. Methods allow the programmer to modularize the program. All variables declared in method definitions are local variables. That means they are known only in the method in which they are defined. Most methods have a list of parameters that provide the means for communicating information between the methods. A methods parameters are also local variables.

There are several motivations for modularizing a program with methods. The divide and conquer approach makes program development more manageable. Another motivation is reusability. That means using existing methods as building blocks to create new programs. With good method naming and definition, programs can be created from standardized methods, rather than being built by using customized code. A third motivation is to avoid repeating code in a program. Packaging code as a method allows that code to be executed from several locations in a program simply by calling the method.

General format of method:

modifier returnvaluetype methodname(list of parameters)

{

//method body;

}

If an object wants another object to do some work on its behalf, then in the parlance of OOP, the first object sends a message to the second object. In response, the second object selects the appropriate method to invoke. Java method invocations look similar to functions in C.

Using the message passing paradigm of OOP, you can build entire networks and webs of objects that pass messages between them to change state. This programming technique is one of the best ways to create models and simulations of complex real-world systems.

Example:

MODIFIER RETURNVALUETYPE

Public static int max(int num1, int num2)

NAME PARA-LIST

```
{
  if(num1>num2)
 {
   return num1;
else
    return num2;
}
```

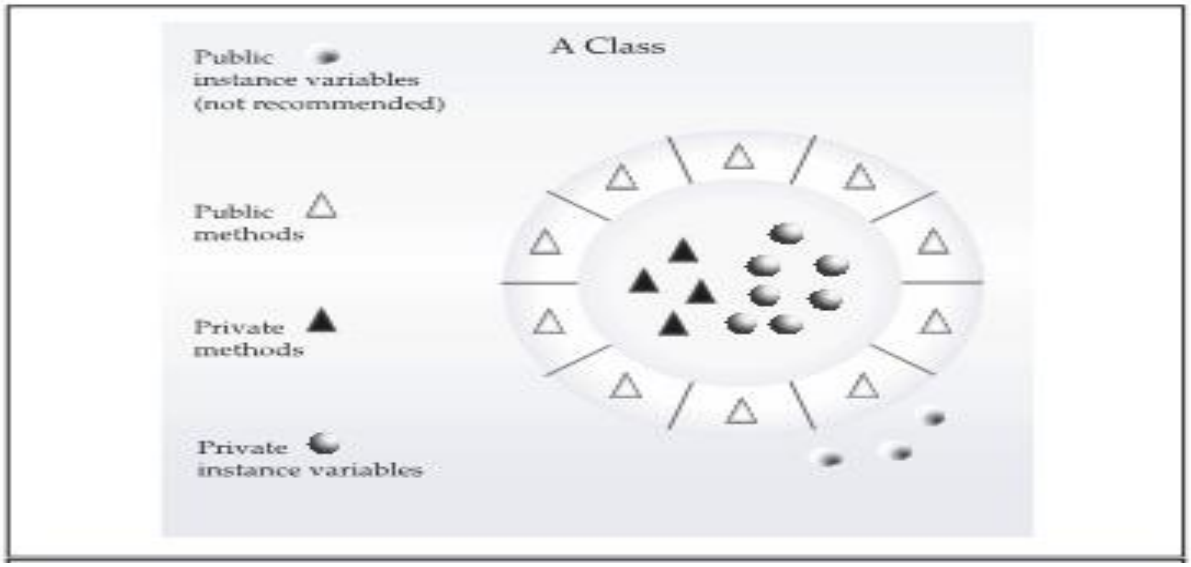
RETURN VALUE

CLASSES AND INSTANCES:

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such,the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class. Because the class is so fundamental to Java.

Class Fundamentals

The classes created in the preceding description primarily exist simply to encapsulate the main( ) method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far. Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, ***a class is a template for an object, and an object is an instance of a class***. Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.



The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex.

The general form of a class definition is shown here:

```
class classname {

type instance-variable1;

type instance-variable2;

// ...

type instance-variableN;

type methodname1(parameter-list) {

// body of method

}
```

type methodname2(parameter-list) {

// body of method

}

// ...

type methodnameN(parameter-list) {

// body of method

}

}

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.

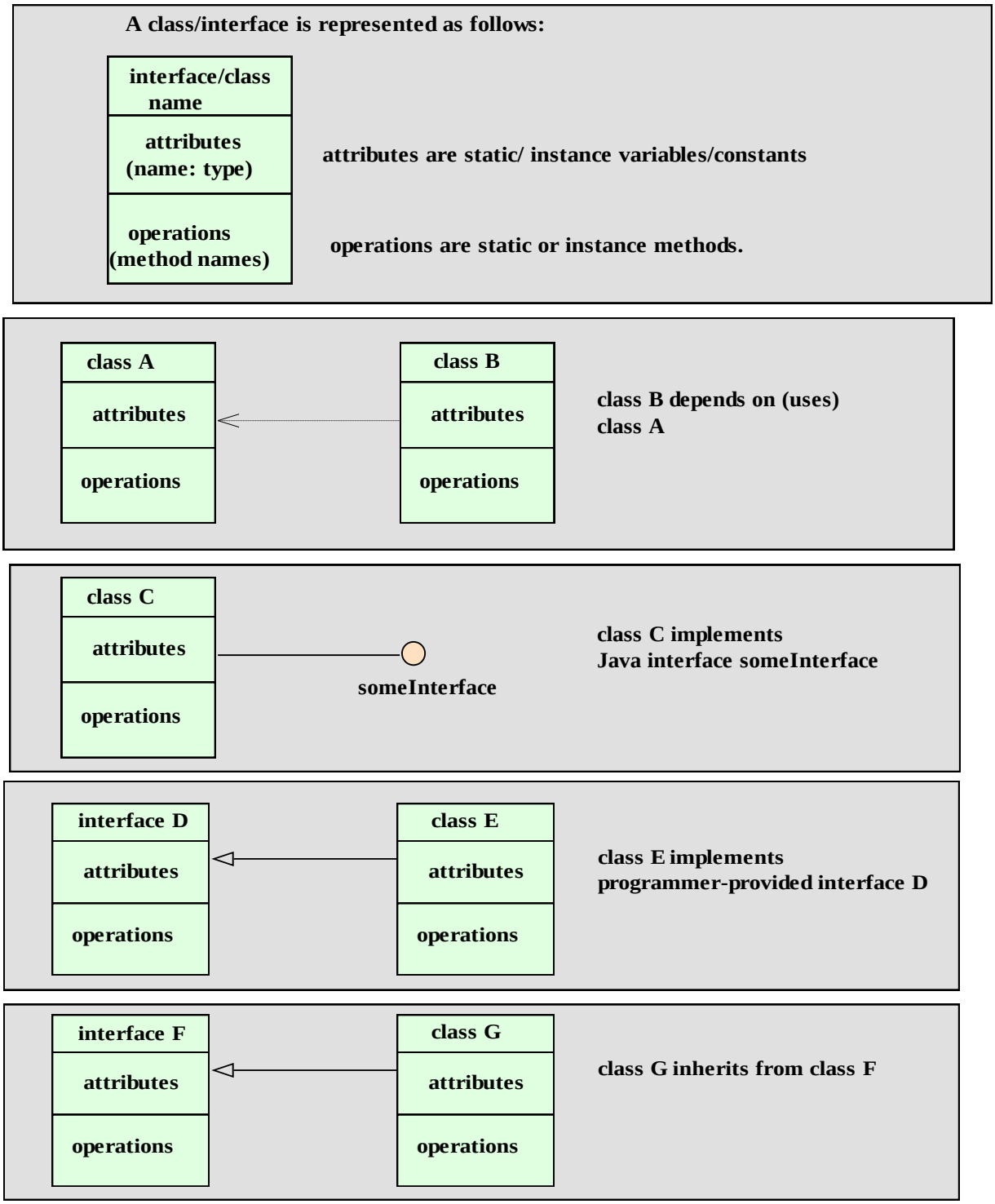
Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. All methods have the same general form as main( ), which we have been using thus far. However, most methods will not be specified as static or public. ***Notice that the general form of a class does not specify a main( ) method. Java classes do not need to have a main( ) method. You only specify one if that class is the starting point for your program. Further, applets don't require a main( ) method at all.***

A class is a software construct that defines the data(states) and methods(behavior) of the specific concrete objects that are subsequently constructed from that class. In Java terminology, a class is built out of members, which are either fields or methods. Fields are the data for the class. Methods are the sequences of statements that operate on the data. Fields are normally specific to an object-that is, every object constructed from the class definition will have its own copy of the field. Such fields are known as instance variables. Similarly methods are also normally declared to operate on the instance variables of the class, and are thus known as instance methods.

A class in and of itself is not an object. A class is like a blueprint that defines how an object will look and behave when the object is created or instantiated from the specification declared by the class. You obtain concrete objects by instantiating a previously defined class. You can instantiate many objects from one class definition, just as you can construct many houses all the same from a single architect's drawing.

CLASS HIERARCHIES(INHERITANCE):

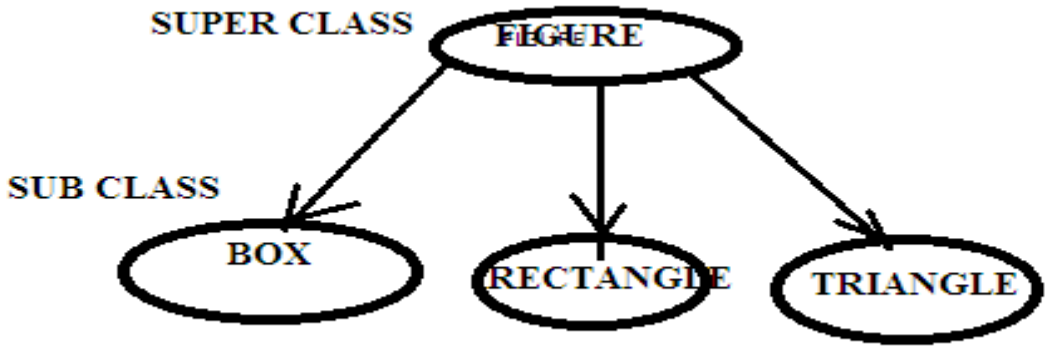
Basic UML Class Diagram Notations



Inheritance is the process by which objects of one class acquire the properties of objects of another class. Inheritance supports the concept of hierarchical classification. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.

Most people naturally view the world as made up of ojects that are related to each other in a hierarchical way.

Inheritance: A new class (subclass, child class) is derived from the existing class(base class, parent class).



Main uses of Inheritance:

1. Reusability

2. Abstraction

Method Overriding:

In a class hierarchy, when a method in a subclass has the same name and the type signature as a method in its super class, then the method in the subclass is said to be override the method in the super class.

Example for Method Overriding and Inheritance:

```
class Figure

{

  double dim1;

  double dim2;
```

```
  Figure(double a,double b)
 {
   dim1=a;
   dim2=b;
 }
 double area()
  {
    System.out.println("Area for figure is undefined");
    return 0;
  }


}
class Rectangle extends Figure
 {
   Rectangle(double a,double b)
   {
     super(a,b);
    }
    double area()
  {
    System.out.println("Inside area for rectangle");
    return dim1*dim2;
  }
 }
class Triangle extends Figure
```

```
 {
  Triangle(double a,double b)
  {
    super(a,b);
   }
   double area()
 {
   System.out.println("Inside area for triangle");
   return dim1*dim2/2;
 }
}
class FindAreas
{
  public static void main(String args[])
  {
    Figure f=new Figure(10,10);
    Rectangle r=new Rectangle(9,5);
    Triangle t=new Triangle(10,8);
     Figure figref;
     figref=r;
    System.out.println("area is"+figref.area());
     figref=t;
    System.out.println("area is"+figref.area());
     figref=f;
    System.out.println("area is"+figref.area());
```

```
  }
}
```

OUTPUT:

```
Inside area for rectangle
area is 45
Inside area for triangle
area is 40
Inside area for figure is undefined
area is 0
```

Use of *super* Keyword:

Whenever a subclass needs to refer to its immediate super class, it can do so by the use of the keyword *super*.

Super has the two general forms.

1. super (args-list) : calls the Super class's constructor.
2. Super . member: To access a member of the super class that has been hidden by a member of a subclass. Member may be variable or method.

Use: Overriden methods allow Java to support Run-time polymorphism. This leads to Robustness by Reusability.

Method Binding:

If we use a super class reference to a subclass object and invoke the (common) method , the program will choose the correct subclass's (coommon) method . The program will find out a class to which the reference is actually pointing and that class method will be binded.

```
class Figure
{
  double dim1;
  double dim2;
  Figure(double a,double b)
 {
   dim1=a;
   dim2=b;
 }
 double area()
  {
    System.out.println("Area for figure is undefined");
    return 0;
  }


}
class Rectangle extends Figure
 {
   Rectangle(double a,double b)
   {
     super(a,b);
    }
```

```
    double area()
  {
    System.out.println("Inside area for rectangle");
    return dim1*dim2;
  }
 }
class Triangle extends Figure
 {
   Triangle(double a,double b)
   {
     super(a,b);
    }
    double area()
  {
    System.out.println("Inside area for triangle");
    return dim1*dim2/2;
  }
 }
class FindAreas
{
   public static void main(String args[])
   {
     Figure f=new Figure(10,10);
     Rectangle r=new Rectangle(9,5);
     Triangle t=new Triangle(10,8);
```

```
    Figure figref;
    figref=r;
   System.out.println("area is"+figref.area());
      figref=t;
     System.out.println("area is"+figref.area());
      figref=f;
     System.out.println("area is"+figref.area());
  }
}
```

OUTPUT:

Inside area for rectangle

area is 45

Inside area for triangle

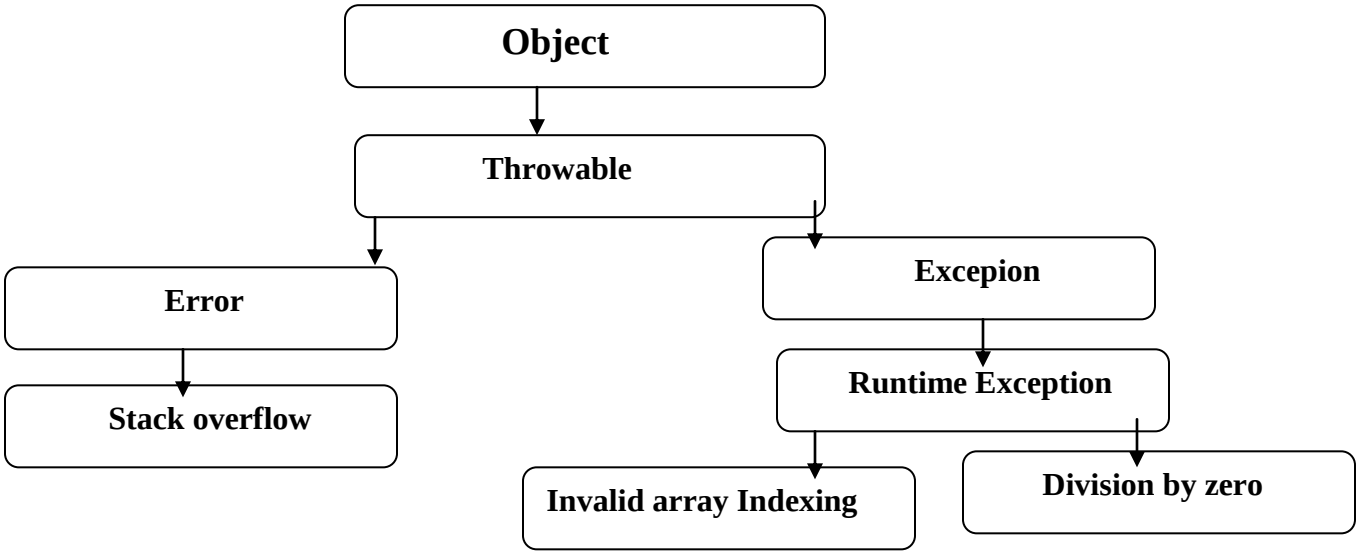
area is 40

Inside area for figure is undefined

area is 0

Exceptions:

An exception is an abnormal condition that arises during the execution of a program that disrupts the normal flow of execution.



Error: When a dynamic linking failure or some other “hard” failure in the virtual machine occurs, the virtual machine throws an Error.

Java exception handling is managed via by five keywords: try, catch, throw, throws, finally.

Try: The try block is said to govern the statements enclosed within it and defines the scope of any exception associated with it. It detects the exceptions.

Catch: The catch block contains a series of legal Java statements. These statements are executed if and when the exception handler is invoked. It holds an exception.

Throw: To manually throw an exception ,use the keyword throw.

Throws: Any exception that is thrown out of a method must be specified as such by a throws clause.

Finally: Any code that absolutely must be executed after a try block completes is put in a finally block. After the exception handler has run, the runtime system passes control to the finally block.

General form of an exception handling:

```
try
{
  //block of code to monitor for errors
}
catch(ExceptionType exOb)
{
  //exception handler for ExceptionType
}
//...
finally
{
  //block of code to be executed after try block ends
}
```

Example:

```
public class ExceptionDemo
{
   public static void main(String args[])throws IOException
   {
      int subject[]={12,23,34,21};
      try
      {
        System.out.println(subject[2]);
        System.out.println("not okay");
```

```
      }
      catch(ArrayIndexOutOfBoundException e)
     {
       System.out.println("i caught the exception:"+e);
        throw e;
      }
    finally
    {
      System.out.println("okay");
     }
  }
}
```

Output:

34

Not Okay okay

Coping with complexity:

The size and complexity of a small program is small and simple. Whereas, the size and complexity of a large application program is large and hard. The complexity in dealing with the problems to build a large application depends on the ‘composition’ and ‘abstraction’ mechanisms.

Composition mechanisms:

We can create objects of another (super) class as instance variable in the current (sub) class. This capability is called composition. Composition is one form of software reuse in which a class has references to objects of other classes as members. It is called a has a relationship.

```
class Figure

{

  double dim1;

  double dim2;

  Figure(double a,double b)

 {

   dim1=a;

   dim2=b;

 }

 double area()

  {

    System.out.println("Area for figure is undefined");

    return 0;

  }


}

class Rectangle extends Figure

 {
```

```
  Rectangle(double a,double b)
  {
    super(a,b);
   }
   double area()
 {
   System.out.println("Inside area for rectangle");
   return dim1*dim2;
 }
}
class Triangle extends Figure
{
  Triangle(double a,double b)
  {
    super(a,b);
   }
   double area()
 {
   System.out.println("Inside area for triangle");
   return dim1*dim2/2;
 }
}
class FindAreas
{
  public static void main(String args[])
```

```
  {
    Figure f=new Figure(10,10);
    Rectangle r=new Rectangle(9,5);
    Triangle t=new Triangle(10,8);
  //creating objects of one(super) class in another(sub) class:Composition
    Figure figref;
        figref=f;
       System.out.println("area is"+figref.area());
    figref=r;
    System.out.println("area is"+figref.area());
     figref=t;
    System.out.println("area is"+figref.area());

  }
}
```

OUTPUT:

```
Inside area for rectangle
area is 45
Inside area for triangle
area is 40
Inside area for figure is undefined
area is 0
```

Abstraction Mechanisms:

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of attributes and methods to operate on these attributes. They encapsulate all the essential features of the objects that are to be created since the classes use the concept of data abstraction they are known as Abstract Data Types.

An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units.

For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior. You can treat these objects as concrete entities that respond to messages telling them to do something. This is the essence of object-oriented programming.

Object-oriented concepts form the heart of Java just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging. For example, once you have well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear.

Abstract class: Any class that contains one or more abstract methods must also be declared abstract.

To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract

constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the super class, or be itself declared abstract.

There are situations in which you will want to define a super class that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a super class that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a super class is unable to create a meaningful implementation for a method. This is the case with the class Figure used in the preceding example. The definition of area( ) is simply a placeholder. It will not compute and display the area of any type of object. As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its super class. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods which must be overridden by the subclass in order for the subclass to have any meaning. Consider the class Triangle. It has no meaning if area( ) is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the abstract method.

Abstract method: A method that is declared but not implemented (no body). Abstract methods are used to ensure that subclasses implement the method.

You can require that certain methods be overridden by subclasses by specifying the abstract type modifier. These methods are sometimes referred to as subclasses responsibility because they have no implementation specified in the super class. Thus, a subclass must override them—it cannot simply use the version defined in the super class. To declare an abstract method, use this general form:

abstract type name(parameter-list);

As you can see, no method body is present.

An abstract class can be sub classed and can't be instantiated.

Example program for Abstract class and method:

```
abstract class Figure
 {
        double dim1;
        double dim2;
        Figure(double a, double b)
        {
```

```
        dim1 = a;
        dim2 = b;
    }
// area is now an abstract method
    abstract double area();
}
class Rectangle extends Figure
 {
    Rectangle(double a, double b)
     {
        super(a, b);
    }


// override area for rectangle
    double area()
    {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
class Triangle extends Figure
{
    Triangle(double a, double b)
     {
      super(a, b);
    }
```

```
// override area for right triangle
        double area()
         {
                System.out.println("Inside Area for Triangle.");
                return dim1 * dim2 / 2;
        }
}
class AbstractAreas
 {
        public static void main(String args[])
        {
                // Figure f = new Figure(10, 10); // illegal now
                Rectangle r = new Rectangle(9, 5);
                Triangle t = new Triangle(10, 8);
                Figure figref; // this is OK, no object is created
                figref = r;
                System.out.println("Area is " + figref.area());
                figref = t;
                System.out.println("Area is " + figref.area());
        }
}
```

output:

Inside area for Rectangle.

Area is 45.0

Inside are for Triangle.

Area is 40.0

As the comment inside main( ) indicates, it is no longer possible to declare objects of type Figure, since it is now abstract. And, all subclasses of Figure must override area( ). To prove this to yourself, try creating a subclass that does not override area( ). You will receive a compile-time error.

Although it is not possible to create an object of type Figure, you can create a reference variable of type Figure. The variable figref is declared as a reference to Figure, which means that it can be used to refer to an object of any class derived from Figure. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

Summary of OOP concepts:

OOP concepts:

If you have never used an object-oriented language before, you need to understand the underlying concepts before you begin writing code. You need to understand what an object is, what a class is, how objects and classes are related, how objects communicate by using messages.

Object: An object is a software bundle of related variables and methods. Software objects are often used to model real-world objects you find in everyday life. It is real time entity distinguished from others by it's name and behavior.

Messages: Information for interaction or data for communication. Software objects interact and communicate with each other using messages.

Class: A class is a blueprint or prototype that defines the variables and the methods common to all objects of a certain kind.

Inheritance: A class inherits state and behavior from its superclass. Inheritance provides a powerful and natural mechanism for organizing and structuring software programs.

Interface: An interface is a contract in the form of a collection of method and constant declarations. When a class implements an interface, it promises to implement all of the methods declared in those methods.

General concepts of OOP:

1.class

2.objects

3. data abstraction

4. data encapsulation

5. inheritance

6. polymorphism

7. dynamic binding

8. message passing

The three principles of OOPs:

1. Encapsulation
2. Inheritance
3. Polymorphism

Encapsulation: Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission.

Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming.

The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

In Java the basis of encapsulation is the class. A class defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class. Thus, a class is a logical construct; an object has physical reality.

Polymorphism: Polymorphism (from the Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler's job to select the specific action (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

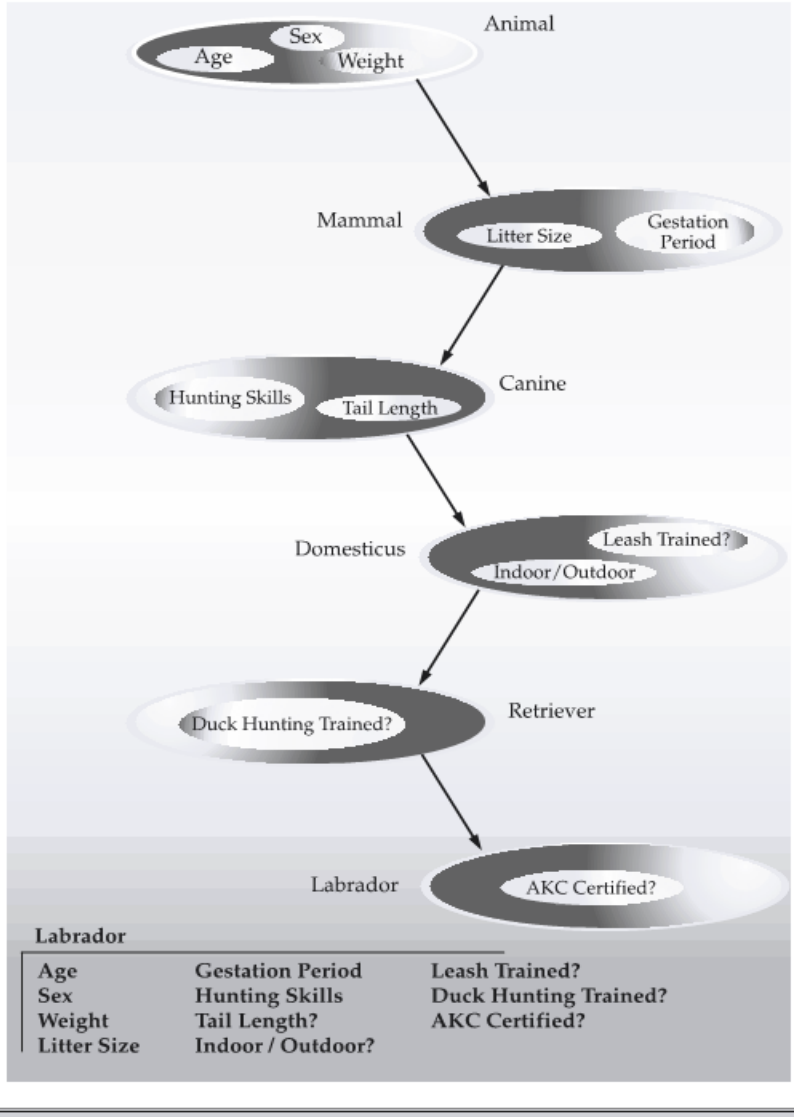


Figure 2-2. *Labrador inherits the encapsulation of all of its superclasses*

Extending the dog analogy, a dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

Advantages of OOPs:

1. We can eliminate redundant code and extend the use of classes with the concept of inheritance.
2. We can build the programs from the standard working modules that communicate with one another, rather than having to start writing the code from beginning. This leads to saving of development time and higher productivity.
3. The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
4. It is possible to have multiple instance of an object to exist without any interference.
5. Software complexity can be managed.
6. OO systems can be easily upgraded from small to large systems.

Applications of OOPs:

1. Real time systems
2. Simulation and modeling
3. OO database
4. Hypertext, hypermedia and expert-text
5. AI and expert systems
6. Neural networks and parallel programming
7. Decision support and office automation systems
8. CAM/CAD systems

Conclusion:

- Everything is an object.
- Methods are used to pass the messages between objects.
- Objects are packaged by class.
- Methods and variables are considered as the members of class.
- One class can inherit the many classes
- One method can do many actions.
- Complexity is managed by abstraction and composition.
- We can hide and bind the data.
- OOP follows the bottom-up approach.