OBJECTSPACE VOYAGER CORE PACKAGE VERSION 1.0 TECHNICAL OVERVIEW

THE AGENT ORB FOR JAVA

The ObjectSpace Voyager™ Core Technology (Voyager) contains the core features and architecture of the ObjectSpace Voyager platform, including a full-featured, intuitive ORB with support for mobile objects and autonomous agents. Also in the core package are services for persistence, scalable group communication, and basic directory services. Voyager can be downloaded for free commercial use from www.objectspace.com and is everything you need to get started building high-impact systems in Java™ today.

This text presents a high-level overview of Version 1.0 of the Voyager Core Technology. It first presents Voyager concepts, then an example of Voyager in action.

There are three models of distributed computing: client/server, peer-based, and agent-based. ObjectSpace's Voyager is the only product that offers the ability to build applications that mix all three models. This makes Voyager a leading tool in distributed application development.

—J.P. Morgenthal, President, New Horizon Computing Corp., leading analyst on Java

What Is Voyager?

ObjectSpace Voyager is the ObjectSpace product line designed to help developers produce high-impact distributed systems quickly. Voyager is 100% Java and is designed to use the Java language object model. Voyager allows you to use regular message syntax to construct remote objects, send them messages, and move them between programs. This reduces learning curves, minimizes maintenance, and, most importantly, speeds your time to market for new advanced systems. Voyager's architecture is designed to provide developers full flexibility and powerful expansion paths.

The root of the Voyager product line is the ObjectSpace Voyager Core Technology. This product contains the core features and architecture of the platform, including a full-featured, intuitive object request broker (ORB) with support for mobile objects and autonomous agents. Also in the core package are services for persistence, scalable group communication, and basic directory services. The ObjectSpace Voyager Core Technology is everything you need to get started building high-impact systems in Java today.

As the industry evolves, other companies providing distributed technologies struggle as they try to adapt to the new Java language. These companies are required to adapt older object models to fit Java. This results in a series of compromises that together have a dramatic impact on time to market and development costs. Voyager, on the other hand, is developed to use the Java language as its fundamental interface.

One of Java's primary differentiations is the ability to load classes into a virtual machine at run time. This capability enables infrastructures to use mobile objects and autonomous agents as another tool for building distributed systems. Adding this capability to older distributed technologies is often impractical and results in difficult-to-use infrastructures. Voyager provides seamless support for mobile objects and autonomous agents.

Future CORBA Integration

Complete bidirectional CORBA integration is scheduled for release as part of the Voyager Core Technology 1.1.0. This additional Java package allows Voyager to be used as a CORBA 2 client or server. You will be able to generate a Voyager remote interface from any IDL file. You will be able to use this interface to communicate with any Voyager or CORBA server. Without modifying the code, you will be able to export any Java class as a CORBA server in seconds, automatically generating IDL for use by CORBA implementations.

As part of the Voyager Core Technology, the CORBA integration will also be free for most commercial use. For more information, read the *ObjectSpace Voyager CORBA Integration Technical Overview* paper on www.objectspace.com.

Developing with Voyager

Voyager was designed from the ground up to solve problems encountered in the development of distributed systems in Java. As the premier Java distributed systems architecture, Voyager is a technology that enables developers to solve these problems quickly and efficiently.

Consider the following issues.

- **Problem**

  Time to market is crucial and development time is expensive. Extra months spent in development mean extra months for competitors to gain market share.

  Solution

  Voyager is the easiest way to build distributed systems in Java. Previous technologies require you to follow a tedious, clumsy, and error-prone multistep process to prepare a class for remote programming. A single Voyager command replaces this hassle and automatically enables any class for distributed computing and persistence in just seconds. Voyager does not require Java classes to be altered. You can remotely construct and communicate with any Java class, even third-party libraries, without accessing the source code. You can remotely persist any serializable object. Other technologies typically require the use of .idl files, interface definitions, and modifications to the original class, all of which consume development time and couple your domain classes tightly to a particular ORB or database technology.

- **Problem**

  Enterprise networks are often comprised of many hardware and operating system platforms. Systems built with legacy ORBs often require separate binaries for each platform. This increases developer load and system complexity, and complicates system maintenance.

  Solution

  Voyager is 100% Java. Voyager applications can be written once and run anywhere Java 1.1 is supported.

- **Problem**

  Resources in a distributed system need to be used wisely. When a machine is being overused, load should be shifted to other, less utilized machines. Existing ORBs do not help developers solve this problem.

  Solution

  Voyager is dynamic. Mobile objects and agents can be used to encapsulate processing, and can migrate through the network, carrying their work load with them. Instead of having only static processes occurring in a given virtual machine, developers now have the ability to exploit the natural connection between agents and processing. The result is effortless, dynamic load balancing.

- **Problem**

  Information access needs vary. Sometimes information needs to be broadcast across the enterprise; sometimes it should be filtered based on the needs of the user. Sometimes it is transient, while at others it is stored for future use.

  Solution

  Voyager is comprehensive. It allows development of high performance "push" systems using the built-in publish/subscribe technology. Voyager also supports distributed persistence, multicasting, and a rich set of message types.

- **Problem**

  Developers need to leverage their JavaBeans™ components in a distributed context, but cannot afford to modify their architecture with wrapper code or spend time developing complex glue logic.

  Solution

  Voyager is JavaBeans enabled. It provides support for distributed JavaBeans events without any modifications to the beans. No other ORB has such a seamless beans event distribution model.

- **Problem**

  Large systems and congested networks often result in sluggish software. Today's high performance needs require responsive software.

  Solution

  Voyager is fast. Remote messages with Voyager are as fast as the CORBA ORBs. Messages delivered by mobile agents can be up to 1,000,000 times quicker.

- **Problem**

  Today's embedded systems require small runtime footprints. Similarly, Web applets have to be small to minimize download times.

  Solution

  Voyager is compact. The entire Voyager system is less than 200KB, not including the JDK classes it uses. It is only 100K when compressed in a .jar file. Voyager is a fully functional, agent-enhanced object request broker and does not require any additional software beyond JDK 1.1.

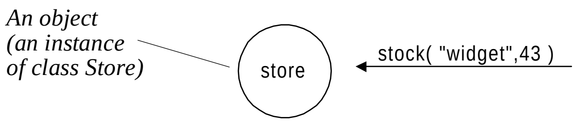
Concepts

This section describes the concepts behind the ObjectSpace Voyager Core Technology architecture, using a mix of text, example code, and drawings.

Objects

Objects are the building blocks of all Voyager programs. An object is a software component that has a well-defined set of public functions and encapsulates data. The following object is an instance of the class `Store` with a public function to accept new stock.

```
Store store = new Store();
store.stock( "widget", 43 );
```

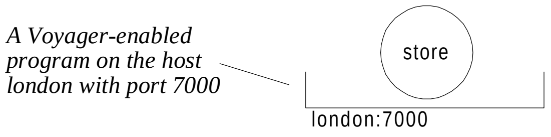


Voyager-Enabled Programs

When a Voyager-enabled program starts, it automatically spawns threads that provide timing services, perform distributed garbage collection, and accept network traffic. Each Voyager-enabled program has a network address consisting of its host name and a communications port number, which is an integer unique to the host.

Port numbers are usually randomly allocated to programs. This is sufficient for clients communicating with remote objects and for creating and launching agents into a network. However, if a program will be addressed by other programs, you can assign a well-known port number to the program at startup.

```
Voyager.startup( 7000 ); // assign port number 7000 to this program
Store store = new Store();
```

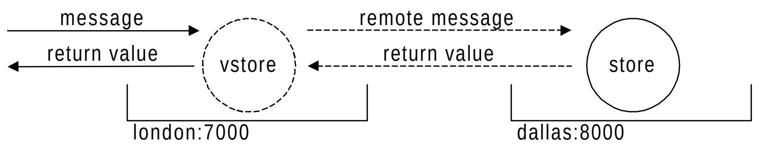


Agent technology will be as important for the Internet as the Internet has been for personal computing. Voyager is the most powerful and easy-to-use solution for agent-enabled distributed computing I have seen.

—John Nordstrom, Sabre Decision Technologies

Remote-Enabled Classes and Virtual References

A class is remote-enabled if its instances can be created outside the local address space of a program and if these instances can receive messages as if they were local. Voyager allows an object to communicate with an instance of a remote-enabled class via a special object called a *virtual reference*. When messages are sent to a virtual reference, the virtual reference forwards the messages to the instance of the remote-enabled class. If a message has a return value, the target object sends the return value to the virtual reference, which returns this message to the sender.



After remote-enabling a class, you can:

- Construct instances remotely, even if the class code does not exist on the remote machine.
- Send messages to remote instances using regular Java syntax.
- Connect to existing remote instances in other programs.
- Move objects to other programs, even if the class code is not already in the destination program.
- Persist the object.

Generating a Remote-Enabled Class

Use Voyager's `vcc` utility to generate a remote-enabled class from an existing class. The `vcc` utility reads a `.class` or `.java` file and generates a new *virtual class*. The virtual class contains a superset of the original class functions and allows function calls to occur even when objects are remote or moving.

The virtual class name is `V` plus the original class name. For example, if the file `Store.java` contains the source code for class `Store`, the compiled class file is `Store.class`. You can remote-enable the `Store` class by running `vcc` on either `Store.java` or `Store.class` to create a new, virtual class named `VStore`.

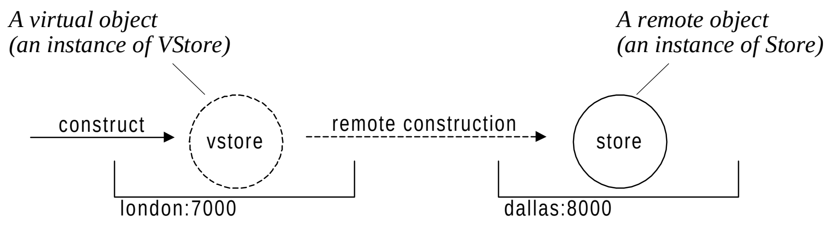
For more detailed information about remote enabling, refer to Chapter 5 of the *ObjectSpace Voyager Core Technology User Guide*.

Constructing a Remote Object

After remote-enabling a class, you can use the class constructors of the resulting virtual class to create a remote instance of the original class. The remote instance can reside in your current program or a different program, and a virtual reference to the remote instance is created in your current program.

To construct a remote instance of a class, give the virtual class constructor the address of the destination program where the remote instance will reside. If the original class code for the remote instance does not exist in the destination program, the Voyager network class loader automatically loads the original class code into the destination program.

```
Voyager.startup( 7000 );
VStore vstore = new VStore( "dallas:8000/Acme" ); // alias is Acme
```

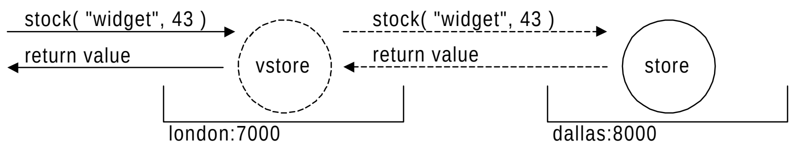


When a remote object is constructed, it is automatically assigned a 16-byte globally unique identifier (GUID), which uniquely identifies the object across all programs worldwide. Optionally, you can assign an alias to an object during construction. The GUID or the optional alias can be used to locate or connect to the object at a later point in time. This directory service is a basic Voyager feature. Voyager also includes an advanced federated directory service for more complex directory requirements. Refer to Chapter 16 of the *ObjectSpace Voyager Core Technology User Guide* for more information.

Sending a Message to a Remote Object

When a message is sent to a virtual reference, the virtual reference forwards the message to its associated remote object. If the message requires a return value, the remote object passes the return value to the virtual reference, which forwards it to the sender. Similarly, if the remote object throws an exception, the exception is caught and passed back to the virtual reference, which throws it to the caller.

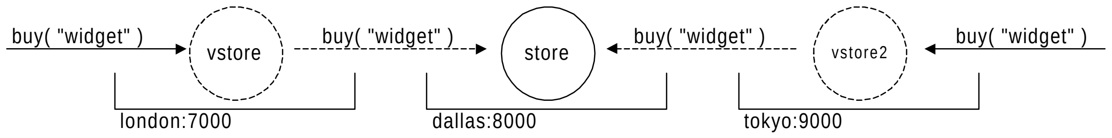
```
vstore.stock( "widget", 43 );
```



Connecting to an Existing Remote Object

A remote object can be referenced by any number of virtual references. To create a new virtual reference and associate it with an existing remote object, supply the address of the program where the existing remote object currently resides and the alias of the remote object to the static `VObject.forObjectAt()` method.

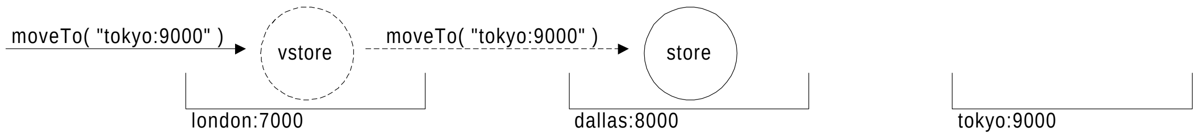
```
// connect using alias
Voyager.startup( 9000 );
VStore vstore2 = (VStore) VObject.forObjectAt( "dallas:8000/Acme" );
int price = vstore2.buy( "widget" );
```



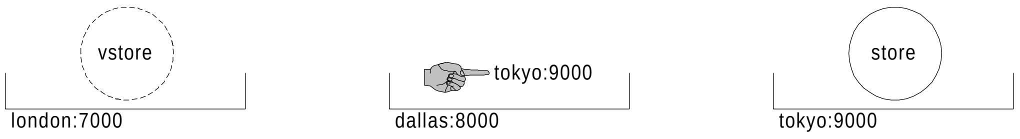
Mobility

You can move any serializable object from one program to another by sending the `moveTo()` message to the object via its virtual reference. Supply the address of the destination program as a parameter.

```
vstore.moveTo( "tokyo:9000" );
```

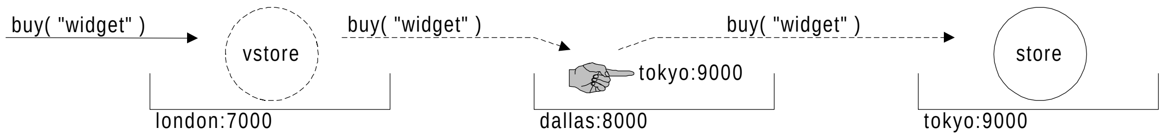


The object waits until all pending messages are processed and then moves to the specified program, leaving behind a forwarder to forward messages and future connection requests.

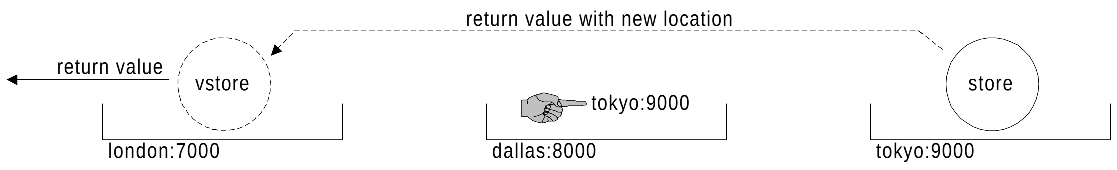


You can send a message to an object even if the object has moved from one program to another. Simply send the message to the object at its last known address. When the message cannot locate its target object, the message searches for a forwarder. If the message locates a forwarder representing the object, the forwarder sends the message to the object's new location.

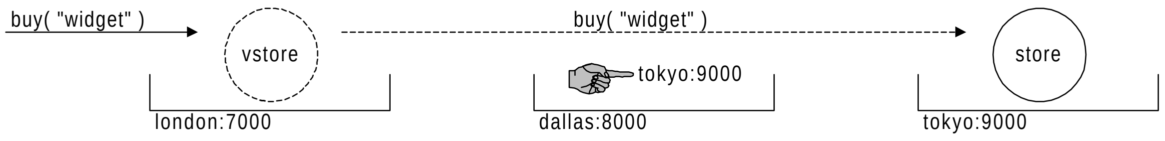
```
  int price = vstore.buy( "widget" );
```



The return value is tagged with the remote object's new location, so the virtual reference can update its knowledge of the remote object's location.



Subsequent messages are sent directly to the remote object at its new location, bypassing the forwarder.



Agents

An agent is a special object type. Although there is no single definition of an agent, all definitions agree that an agent has autonomy. An autonomous object can be programmed to satisfy one or more goals, even if the object moves and loses contact with its creator.

Some definitions state that an agent has mobility as well as autonomy. Mobility is the ability to move independently from one device to another on a network. Voyager agents are both autonomous and mobile. They have all the same features as simple objects—they can be assigned aliases, have virtual references, communicate with remote objects, and so on.

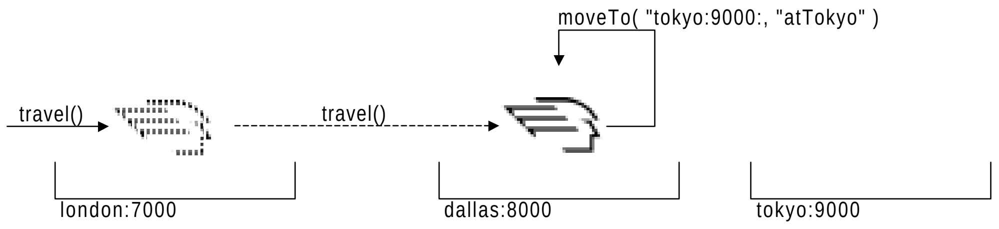
To create an agent, extend the base class `COM.objectspace.voyager.Agent`, and then use Voyager's `vcc` utility to remote-enable the agent's class. Use the resulting virtual class to instantiate an agent object and use virtual references to communicate with this object even if it moves.

Like all objects, an agent can be moved from one program to another. However, unlike simple objects, an agent can move itself autonomously. An agent can move to other programs, allowing the execution of distributed itineraries, or an agent can move to other objects, allowing communication using high-speed, local messaging.

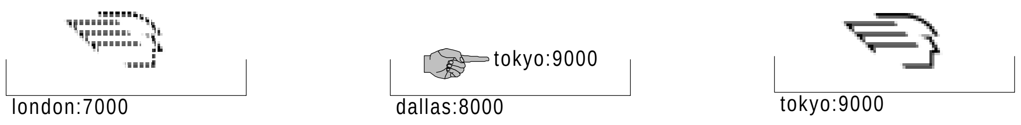
An agent can move to another program and continue to execute when it arrives by sending itself `moveTo()` with the address of the destination program and the name of the member function that should be executed on arrival.

For example, an agent in `dallas:8000` is told to travel. The agent sends itself a `moveTo()` message with two parameters: `dallas:9000`, the destination address, and `atTokyo`, the name of the callback function.

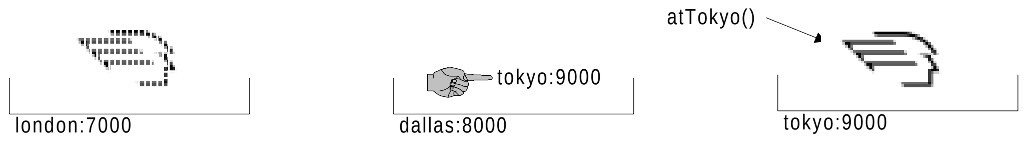
```
public void travel() // defined in Shopper
  {
  moveTo( "tokyo:9000", "atTokyo" );
  }
```



The agent then moves to `tokyo:9000`, leaving behind a forwarder to forward messages.



After arriving at its new location, the agent automatically receives the `atTokyo()` message.



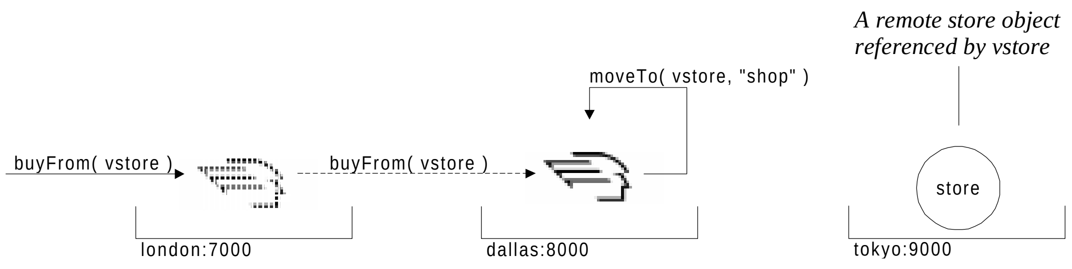
The following code in the agent is then executed.

```
public void atTokyo() // defined in Shopper
  {
  // this code is executed when I move successfully to tokyo:9000.
  }
```

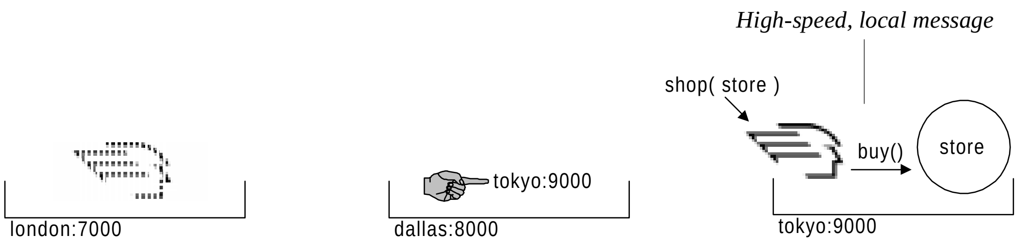
If an agent wants to have a high-speed conversation with a remote object, the agent can move to the object and then send it local Java messages. The easiest way for an agent to move to an object is by sending itself a variation of `moveTo()` that specifies both a virtual reference to the destination object and a callback parameter.

For example, an agent in `dallas:8000` is told to buy from a store object. The agent sends itself a `moveTo()` message with two parameters: `vstore`, a virtual reference to the remote store object, and `shop`, the name of a callback function.

```
public void buyFrom( VStore vstore ) // defined in Shopper
  {
  moveTo( vstore, "shop" );
  }
```



After leaving behind a forwarder and moving to `tokyo:9000`, the agent receives the callback message `shop()` with a local native Java reference to the object `store`.



The following code in the agent is then executed.

```
public void shop( Store store ) // defined in Shopper
  {
  // this code is executed when I successfully move to the store
  // note that store is a regular Java reference to the store
  int price = store.buy( "widget" );
  }
```

JavaBeans Integration

Voyager is designed to integrate with the JavaBeans™ component model. Existing JavaBeans can be used in a Voyager system. Voyager extends the beans delegation event model by allowing all events to be transmitted across the network. This is possible without modifying the bean or event classes in any way.

Voyager also uses the beans event model for object- and system-level monitoring. Every remote Voyager object is automatically a source of events. Objects can listen to remote objects and monitor every aspect of the remote object’s behavior. In particular, listeners are notified when the remote object receives messages, moves, is saved to or loaded from a database, and dies.

Voyager allows system-level monitoring with the beans event model as well. Listeners can monitor when the system garbage-collects remote objects, when classes are loaded into the system, when messages are sent and received, and when agents and mobile objects arrive and depart.

Voyager extends the beans event model further by introducing persistent listeners. Typically, developers use standard beans listeners for transient listening. However, more complex systems often require listeners that can move with objects or listeners that can automatically be stored to and retrieved from databases with the source objects. Voyager adds this critical piece of functionality to all listeners of Voyager events.

Voyager’s integration with the JavaBeans event model allows developers to leverage their bean knowledge and experience and apply it directly to their Voyager systems. The event system provides a wealth of information useful for monitoring, auditing, logging, and other higher-level, application-specific actions.

Dynamic Properties

Voyager allows developers to attach key value *properties* to remote objects. These properties are dynamically attached to any object without requiring modification to the object’s source. This property mechanism is used by the publish/subscribe system to allow objects to specify what subjects they are interested in and can also be used to attach application-specific information to an object at runtime.

```
myObject.addProperty( Subscription.SUBSCRIBE, "sports.basktball.*" );
```

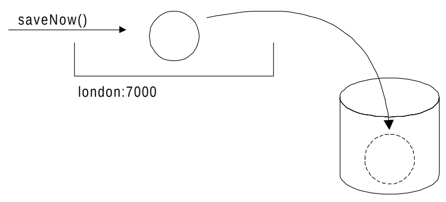
Database-Independent Persistence

A persistent object has a backup copy in a database. A persistent object is automatically recovered if its program is unexpectedly terminated or if it is flushed from memory to the database to make room for other objects. Voyager includes seamless support for object persistence. In many cases, you can persist an object without modifying its source.

Each Voyager program can be associated with a database. The type of database can vary from program to program and is transparent to a Voyager programmer. Voyager includes a high-performance object storage system called `VoyagerDb`, but provides an interface layer to allow developers to drop in their own custom bindings to other popular relational and object databases.

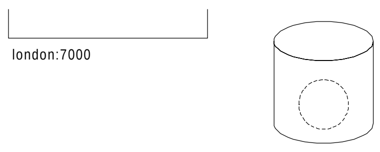
To save an object to the program's database, send `saveNow()` to the object. This method writes a copy of the object to the database, overwriting any previous copy. If the program is shut down and then restarted, the persistent objects are left in the database. Any attempt to communicate with a persistent object causes the object to be reloaded from the database.

See Chapter 14 of the *ObjectSpace Voyager Core Technology User Guide* for more details about Voyager persistence.

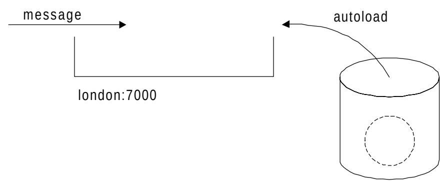


The saveNow() message writes a copy of the persistent object to the database.

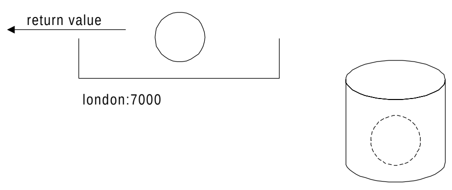
System is shut down temporarily.



When the system restarts, the copy of the object remains in the database, but the actual object is not immediately restored in its original location.

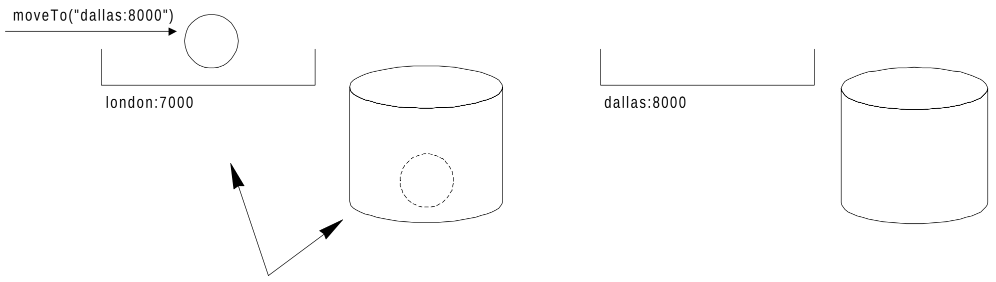


When a message arrives, a copy of the persistent object is autoloaded into memory.

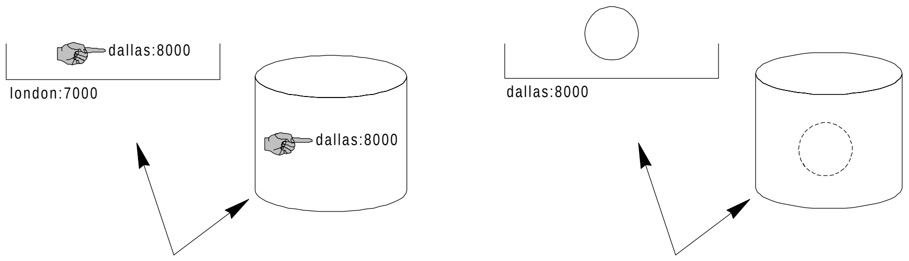


The object is restored, the message is delivered to the object, and a return value is sent.

If a persistent object is moved from one program to another, the copy of the object is automatically removed from the source program’s database and added to the destination program’s database.



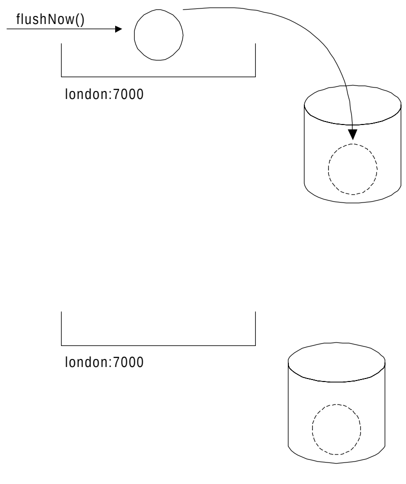
A persistent object in london:7000 and its copy in the database.



Forwarders are left behind in london:7000 and in the london:7000 database.

The persistent object is moved to dallas:8000, and a copy of the persistent object is entered into the dallas:8000 database.

You can conserve memory by using one of the `flush()` family of methods to remove a persistent object from memory and store it in a database. Any subsequent attempt to communicate with a flushed persistent object reloads the object from the database.



The flushNow() message writes a copy of the persistent object to the database and causes the actual object to be garbage-collected from its original location. The object is restored the first time a message is sent to it.

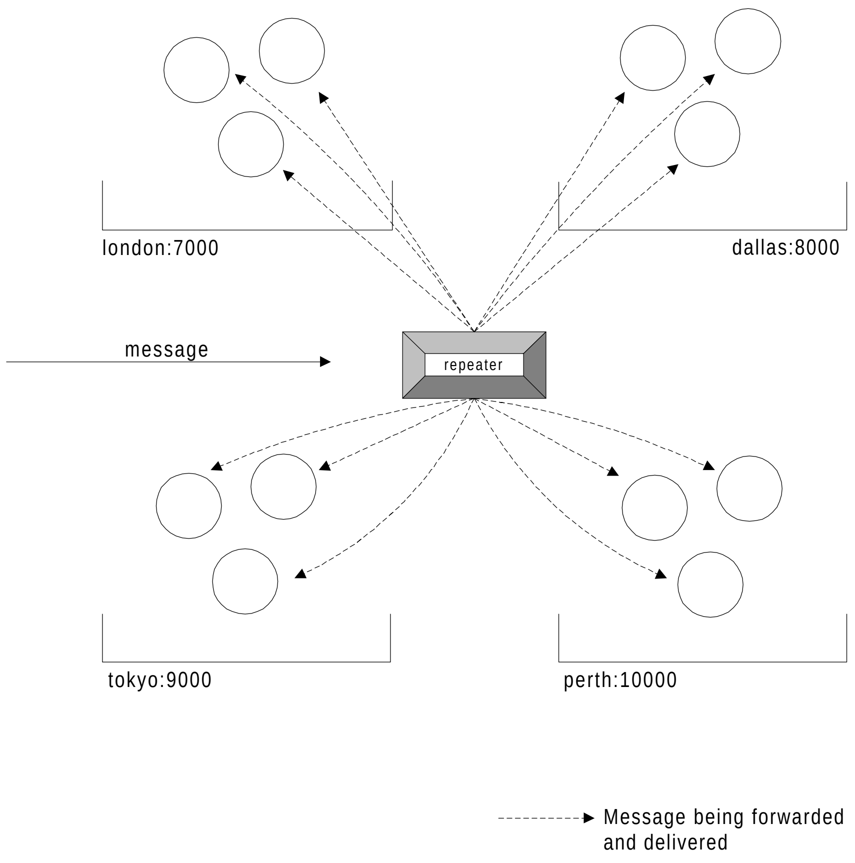
By default, Voyager’s database system automatically persists Java classes loaded into a program across a network, thus avoiding a reload of these classes when the program is restarted.

Space – Scalable Group Communication

Many distributed systems require features for communicating with groups of objects. For example:

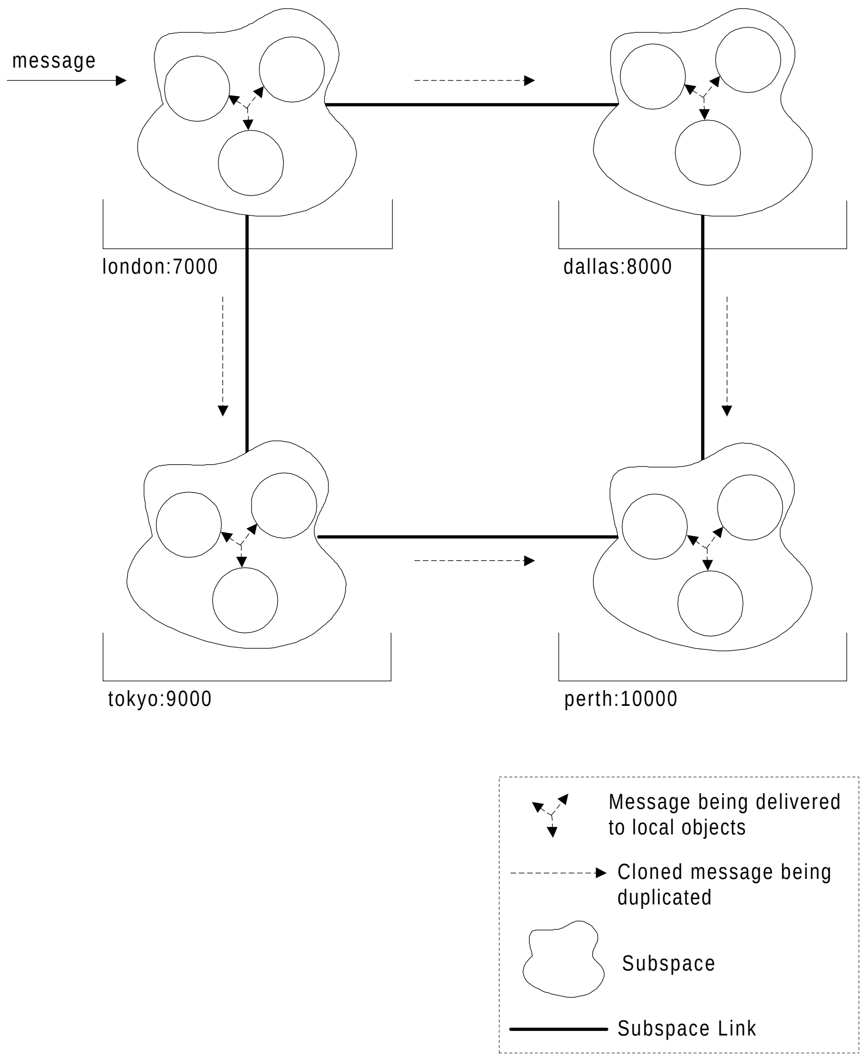
- Stock quote systems use a distributed event feature to send stock price events to customers around the world.
- A voting system uses a distributed messaging feature (multicast) to send messages around the world to voters, asking their views on a particular matter.
- News services use a distributed publish/subscribe feature so that broadcasts are received only by readers interested the broadcast topic.

Most traditional systems use a single *repeater* object to replicate the message or event to each object in the target group.



This traditional approach works well if the number of objects in the target group is small, but does not scale well when large numbers of objects are involved.

Voyager uses a different and innovative architecture for message/event replication called Space™ that can scale to global proportions. Clusters of objects in the target group are stored in local groups called *subspaces*. Subspaces are linked to form a larger logical group called a *Space*. When a message or event is sent into one of the subspaces, the message or event is cloned to each of the neighboring subspaces before being delivered to every object in the local subspace. This process results in a rapid parallel fanout of the message or event to every object in the Space. A special mechanism in each subspace ensures that no message or event is accidentally processed more than once, regardless of how the subspaces are linked.


message
london:7000
dallas:8000
tokyo:9000
perth:10000
Message being delivered to local objects
Cloned message being duplicated
Subspace
Subspace Link

Voyager's multicast, distributed events, and publish/subscribe features all use and benefit from the same underlying Space architecture.

Message Types

Unlike traditional object request brokers, which use a simple, on-the-wire message protocol, Voyager messages are delivered by lightweight agents called *messengers*. Voyager has four predefined message types.

Synchronous Messages

By default, Voyager messages are synchronous. When a caller sends a synchronous message, the caller blocks until the message completes and the return value is received. You can use regular Java syntax to send a synchronous message to an object. Arguments are automatically encoded on the sender side and decoded on the receiver side.

```
int price = vstore.buy( "Killer Rabbits" );
```

One-Way Messages

Although messages are synchronous by default, Voyager supports one-way messages as well. One-way messages do not return a value. When a caller sends a one-way message, the caller does not block while the message completes.

```
vstore.buy( "Killer Rabbits", new OneWay() ); // no return
  ...
```

Future Messages

Voyager also supports future messages. When a caller sends a future message, the caller does not block while the message completes. The caller receives a placeholder that can be used to retrieve the return value later by polling, blocking, or waiting for a callback.

```
Result result = vstore.buy( "Killer Rabbits", new Future() );
  ...
int price = result.readInt(); // Block for price.
```

One-Way Multicast Messages

One-way multicast messages can be used to send one-way messages to all objects in a Space using a single operation.

```
VStore stores = new VStore( space ); // gateway into space
stores.stock( "video", 25 ); // send stock() to all stores in space
```

To send a one-way message to only certain objects in a Space, use a one-way multicast message with a selector.

Selective Multicast Messages

Multicast messages can be selective broadcast to a subset of objects in a space. For instance, Voyager supports traditional publish/subscribe multicasting where objects are selected based on whether or not they are subscribed to given subjects (defined as heirarchical strings). However, Voyager also supports a more general selection mechanism in that messages can be multicast to objects that meet any user-defined criterion.

```
VAccount accounts = new VAccount( space ); // gateway into space
Selector selector = new DelinquentSelector(); // select if delinquent
accounts.close( selector ); // close account if delinquent in payment
```

Federated Directory Service

Voyager provides a directory service for remote object lookup. Using the directory service, an object can get a references to a remote or mobile object without advance knowledge of its location. Voyager's directory service avoids the single-server bottleneck/point-of-failure associated with monolithic directory services by allowing distributed directory services to be linked together to form a single, federated directory service.

All directories are completely integrated with Voyager's persistence mechanism and like any object, can be saved to a database with a single command.

Dynamic Messaging

Voyager supports dynamic message construction at run time. The following code creates a synchronous message at run time using the Java virtual machine syntax for signature definition.

```
// Dynamically create and execute a synchronous message.
Sync sync = new Sync();
sync.setSignature( "buy(Ljava.lang.String;)I" );
sync.writeObject( "Killer Rabbits" );
Result result = vstore.send( sync );
int price = result.readInt(); // price
```

Life Spans and Garbage Collection

Each instance of a remote-enabled class has a life span. When an object reaches the end of its life span, the object dies and is garbage-collected. Garbage collection destroys an object, freeing the object's memory for reclamation by the Java virtual machine.

Voyager includes a distributed garbage collector that supports a variety of life spans.

- An object can live forever.
- An object can live until there are no more local or virtual references to it. By default, an instance of any class that does not extend `Agent` has this kind of reference-based life span.
- An object can live for a specified amount of time. By default, an instance of any class that extends `Agent` lives for one day.
- An object can live until a particular point in time.

You can change an object's life span at any time.

Voyager leverages Java's run-anywhere code mobility to provide true agent-based computing as well as traditional distributed object communication. The ability of agents to move seamlessly about the new network provides a significant advantage for multitier, client/server, and peer-to-peer architectures.

—Michael Greenspon, Sequential Interface, Inc.

Guided Tour

This section guides you through an example project to demonstrate the power and simplicity of the ObjectSpace Voyager Core Technology. All steps necessary to build an agent-enhanced, persistent electronic shopping system are presented, complete with full, annotated source code from the directory `\voyager1.0.0\examples\shopper`.

This section is not a technical reference. For information about a particular aspect of Voyager, consult Parts 2 and 3 of the *ObjectSpace Voyager Core Technology User Guide*.

Introduction

One of the hot areas of computer technology is electronic commerce. As companies begin to allow customers to purchase goods and services electronically, an interesting opportunity arises for the consumer. Rather than scan the yellow pages for stores that sell the product you want, why not use a personal shopping agent that can do this for you automatically? Such an agent could learn your tastes and requirements over time and tirelessly scour the network to find you the best possible deal. Another advantage of such agents is their proactive abilities—they could be smart enough to locate items similar to those you purchased in the past and suggest them to you, rather than relying on you to continually prompt them into action.

This section demonstrates how to build a simple version of an agent-based shopping system in the following phases.

Phase 1: Building Stores. In the first phase, a Java class is defined to represent a store. Two persistent stores are constructed in different Voyager servers, and each store is added to a well-known registry. The Voyager servers are then shut down.

Phase 2: Launching a Shopping Agent. In the second phase, a Java class is defined to represent a shopping agent. The Voyager servers are restarted, and a persistent shopping agent named `Alfred` is constructed in one of the servers and told the location of the store registry and the name of the desired product. `Alfred` sets his itinerary to the contents of the registry, visits each store in turn, and then moves to the store with the best price to await further instructions.

Phase 3: Buying an Item. In the third and final phase, a program is written that contacts `Alfred` to request the store location offering the best price. The program then tells `Alfred` to die and contacts the recommended store to purchase the item directly.

The remainder of this section describes how to complete each phase of the shopping system using Voyager. The program in this section is text based. For an applet-based version of the same program, consult Chapter 10 of the *ObjectSpace Voyager Core Technology User Guide*.

Phase 1: Building Stores

The first phase of the shopping system project constructs two persistent stores and adds virtual references to each store into a registry by performing the following steps:

1. Define a Java class named `Store` that represents a store and generate a virtual version of `Store` so it can be constructed remotely.
2. Choose a class for the registry and generate a virtual version of this class.
3. Write a program named `Build.java` that creates two persistent stores and populates the remote registry.
4. Compile the Phase 1 programs.
5. Start Voyager servers to hold the persistent remote stores.
6. Run `Build.class` to create the stores and registry.
7. Shut down the Voyager servers.

The rest of this section discusses these steps in detail.

Step 1. Define a Java class named Store that represents a store and generate a virtual version of Store so it can be constructed remotely.

For the purposes of this example, a store has limited behavior. The `Store.java` program below gives the `Store` class a name and a hash table that maps the name of a product to its price. `Store` defines functions for adding, pricing, and purchasing a product. Several functions contain print statements used to track transactions as they occur. `Store` is defined to be serializable so it can be stored persistently in the default Voyager database.

Class voyager1.0.0\examples\shopper\Store.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import java.util.Hashtable; // utilize a JDK Hashtable

public class Store implements java.io.Serializable
  {
  String name;
  Hashtable products = new Hashtable(); // contains product->price pairs

  public Store( String name )
    {
    this.name = name;
    System.out.println( "Build " + this );
    }

  public String toString()
    {
    return "Store( " + name + " )";
    }

  public void stock( String product, int price )
    {
    System.out.println( "stock " + product + " @ $" + price );
    products.put( product, new Integer( price ) ); // add product to stock
    }

  public int getPrice( String product )
    {
    Integer integer = (Integer) products.get( product ); // get price
    return integer == null ? 0 : integer.intValue(); // zero if not in stock
    }

  public int buy( String product ) throws IllegalArgumentException
    {
    int price = getPrice( product );

    if( price == 0 )
      throw new IllegalArgumentException( "no " + product + " found" );

    System.out.println( "purchase " + product + " @ $" + price );
    return price;
    }
  }
```

Run the Voyager `vcc` utility on `Store`. The `vcc` utility uses the most recently modified version of the `Store.java` or `Store.class` file to create a virtual class named `VStore`. For the rest of this guided tour, assume commands are typed from a command line in the directory `voyager1.0.0\examples\shopper`.

```
>vcc Store
vcc 1.0.0, copyright objectspace 1997
note: VoyagerException not thrown by java.lang.Object:java.lang.String
toString()
>dir Vstore.*
VSTORE~1 JAV         4,658  08-22-97 10:17a VStore.java
>
```

Like all virtual classes, `VStore` directly or indirectly extends `VObject`, which contains the functionality common to all virtual objects.

Step 2. Choose a class for the registry and generate a virtual version of this class.

This guided tour uses the JDK class `java.util.Vector` as the registry class. Run `vcc` to create a virtual version of `Vector`. The following code creates the virtual class `VVector` and places it in the current directory.

```
>vcc java.util.Vector
vcc 1.0.0, copyright objectspace 1997
note: java.* virtual classes are not placed in a package
note: VoyagerException not thrown by java.lang.Object:java.lang.Object
clone()
note: VoyagerException not thrown by java.lang.Object:java.lang.String
toString()
>dir VVector.*
VVECTO~1 JAV        21,434  08-22-97 11:04a VVector.java
>
```

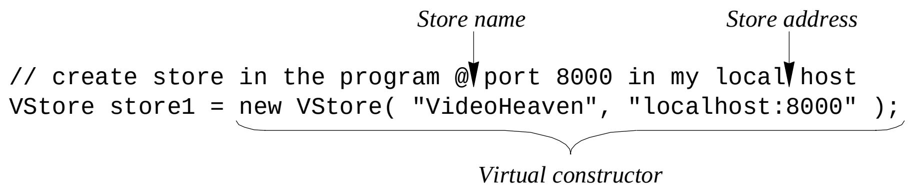
Step 3. Write a program named Build.java that creates two persistent stores and populates the remote registry.

The `Build.java` program below constructs two persistent instances of `Store` in local Voyager servers and populates the persistent registry.

Application voyager1.0.0\examples\shopper\Build.java

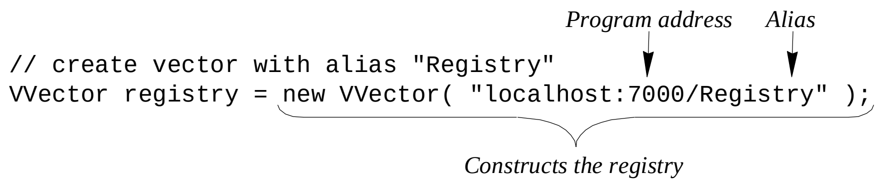
```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;
import VVector;

public class Build
  {
  public static void main( String args[] )
    {
    try
      {
      // create store in local server @ port 8000
      VStore store1 = new VStore( "VideoHeaven", "localhost:8000" );
      store1.liveForever(); // prevent garbage collection
      store1.stock( "Killer Rabbits", 25 ); // stock item
      store1.stock( "Jaws XXIII", 29 ); // stock item
      store1.saveNow(); // become persistent, save copy to database

      // create store in local server @ port 7000
      VStore store2 = new VStore( "MegaHits", "localhost:7000" );
      store2.liveForever(); // prevent garbage collection
      store2.stock( "Killer Rabbits", 35 ); // stock item
      store2.stock( "Jaws XXIII", 30 ); // stock item
      store2.saveNow(); // become persistent, save copy to database

      // create vector with alias "Registry" in local server @ port 8000
      VVector registry = new VVector( "localhost:8000/Registry" );
      registry.liveForever();
      registry.addElement( store1 );
      registry.addElement( store2 );
      registry.saveNow(); // store in database

      System.out.println( "Registry is " + registry );
      Voyager.shutdown(); // shutdown program
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

The `Build.java` program uses the `VStore` constructor to instantiate a `Store`. Virtual class constructors have the same arguments as their original classes plus an additional string that specifies the address of the destination program. In other words, the virtual counterpart of the constructor `Store( String` *name* `)` is `VStore( String` *name*, `String` *address* `)`. The format of an object's address resembles a URL and usually includes the host name and port number of the program in which the object is to be created. You can supply a specific host name or use the built-in host name `localhost` to denote your current local host.



```
// create store in the program @ port 8000 in my local host
VStore store1 = new VStore( "VideoHeaven", "localhost:8000" );
```

The `Build.java` program also creates a remote `Vector` and assigns it the alias `Registry` in a single step. Voyager allows you to assign an alias to a new object using standard URL syntax. A separate name-binding step is not required.



```
// create vector with alias "Registry"
VVector registry = new VVector( "localhost:7000/Registry" );
```

The following excerpt from `Build.java` demonstrates how to prevent an object from being garbage-collected. By default, a simple (non-agent) remote object is garbage-collected when there are no more local or virtual references to it. Sending the `liveForever()` message to an object prevents its garbage collection; that is, the object lives forever unless you explicitly send it the `dieNow()` message. Because the stores and the registry must survive beyond the lifetime of the program, they are made immortal.

```
store1.liveForever(); // prevent garbage collection
```

The `saveNow()` method instructs an object to become persistent and save itself into its program's database.

```
store1.saveNow(); // become persistent, save copy to database
```

Step 4. Compile the Phase 1 programs.

Use the `javac` command to compile the Phase 1 source code.

```
javac Store.java Build.java VStore.java VVector.java
```

Step 5. Start Voyager servers to hold the persistent remote stores.

Start a Voyager server on each of ports 7000 and 8000 by running the `voyager` command in two separate windows. As shown below, this command accepts the required port number as an argument. The `-d` option instructs Voyager to use the named database file for its persistent storage. The `-c` option clears the database file if one already exists. Note that a Voyager server runs until it is explicitly terminated, and two Voyager programs cannot share the same port.

Window 1

```
>voyager 7000 -cd 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 0 objects, 0 classes
```

Window 2

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
```

Step 6. Run Build.class to create the stores and registry.

Run `Build.class` in a third window. The following output is initially displayed.

Window 3

```
>java Build
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1187
```

Window 3 remains inactive while `Build.java` constructs the persistent stores and populates the registry. As `Build.java` executes, additional output displays in the first two windows:

Window 1

```
>voyager 7000 -cd 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 0 objects, 0 classes
Build Store( MegaHits )
stock Killer Rabbits @ $35
stock Jaws XXIII @ $30
```

Window 2

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
Build Store( VideoHeaven )
stock Killer Rabbits @ $25
stock Jaws XXIII @ $29
```

After the stores are constructed and the registry is populated, the `Build.java` program displays an additional line of output in Window 3, then terminates.

Window 3

```
>java Build
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1187
Registry is [Store( VideoHeaven ), Store( MegaHits )]
>
```

Step 7. Shut down the Voyager servers.

Phase 1 is now complete. Shut down the two Voyager servers by pressing Ctrl+C in Windows 1 and 2.

Phase 2: Launching a Shopping Agent

In this phase, a persistent shopping agent named `Alfred` is used to find the best price for a product. When launched, `Alfred` sets his itinerary to the contents of the registry, visits each store in the itinerary to find the best price, and then parks at the server that contains the best store to await further instructions. Phase 2 is comprised of the following steps:

1. Define a class named `Shopper` that represents a shopping agent.
2. Write a program named `Shop.java` to instantiate and launch an instance of `Shopper`.
3. Create a virtual version of `Shopper`.
4. Compile the Phase 2 programs.
5. Restart the Voyager servers.
6. Run `Shop.class` to launch the shopper.

The rest of this section discusses these steps in detail.

Step 1. Define a class named Shopper that represents a shopping agent.

The `Shopper` class defined below extends the `Agent` class and adds behavior specific for a shopping agent. The program does not need to override any special functions for the agent to operate correctly. Because `Agent` implements `Serializable`, all the nontransient, nonstatic fields in the agent are automatically maintained as the agent moves.

Class voyager1.0.0\examples\shopper\Shopper.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;
import java.util.Vector;
import VVector;

public class Shopper extends Agent
  {
  String product; // the product to locate
  Vector itinerary; // list of stores to visit
  int index; // index into itinerary
  VStore bestStore = null; // store with cheapest price
  int bestPrice = Integer.MAX_VALUE; // current best price
  boolean parked = false; // have I finished?

  public void findBestPriceFor( String product, VVector registry )
    {
    this.product = product;

    try
      {
      moveTo( registry, "atRegistry" );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
```

```
public void atRegistry( Vector registry )
  {
  itinerary = (Vector) registry.clone(); // get local copy of registry
  System.out.println( "shopping using itinerary: " + itinerary );

  try
    {
    moveTo( (VStore) itinerary.elementAt( index ), "shop" );
    }
  catch( VoyagerException exception )
    {
    System.err.println( exception );
    }
  }

public void shop( Store store )
  {
  int price = store.getPrice( product );

  if( price == 0 )
    {
    System.out.println( "at " + store + ", " + product + " not sold" );
    }
  else
    {
    System.out.println( "at " + store + ", " + product + " is $" + price );

    if( price < bestPrice ) // best store so far
      {
      // obtain virtual reference to store
      try
        {
        bestStore = (VStore) VObject.forObject( store );
        }
      catch( VoyagerException exception )
        {
        System.err.println( exception );
        }

      bestPrice = price;
      }
    }

  // delay to make execution easier to follow
  try{ Thread.sleep( 5000 ); } catch( InterruptedException exception ) {}

  try
    {
    if( ++index < itinerary.size() )
      moveTo( (VStore) itinerary.elementAt( index ), "shop" ); // next store
    else
      moveTo( bestStore.getProgramAddress(), "park" ); // best store
    }
  catch( VoyagerException exception )
    {
    System.err.println( exception );
    }
  }
  public void park()
    {
    parked = true;
    System.out.println( "at " + bestStore + ",  best price $" + bestPrice );
    System.out.println( "shopper parks at " + Voyager.getAddress() );

    if( getPersistent() ) // if i'm persistent save my final state
      {
      try
        {
        flushNow(); // save copy to database and flush from memory
        }
      catch( VoyagerException exception )
        {
        System.err.println( exception );
        }
      }
    }

  public VStore getBestStore()
    {
    if( !parked )
      throw new IllegalStateException( "not parked yet" );

    return bestStore;
    }
  }
```

A shopper uses two variations of `moveTo()` to move and continue execution on arrival:

- When `moveTo()` is passed a program address and a function name, the agent is moved to the specified program and is then resumed by executing the callback function with no arguments.
- When `moveTo()` is passed a virtual reference and a function name, the agent is moved to the referenced object and is then resumed by executing the callback function with a local reference to the target object as the single argument. The agent can then communicate with the target object using high-speed, local Java method calls.

This callback style of programming, familiar to any programmer who has created a graphical user interface, neatly avoids Java's inability to maintain an execution stack across virtual machine boundaries.

If an error occurs at any time during a move, an exception is thrown when the agent calls `moveTo()`. Because an agent is deactivated conceptually when it executes `moveTo()`, a programming error occurs if any code other than exception-handling code follows these methods.

Step 2. Write a program named Shop.java to instantiate and launch an instance of Shopper.

The `Shop.java` program below creates a persistent instance of `Shopper` with alias `Alfred` and tells him to find the best price for a video named *Killer Rabbits.*

Application voyager1.0.0\examples\shopper\Shop.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;
import VVector;

public class Shop
  {
  public static void main( String[] args )
    {
    try
      {
      // connect to vector with alias "Registry" in local server @ port 8000
      VVector registry =
        (VVector) VObject.forObjectAt( "localhost:8000/Registry" );

      // create a shopper with alias "Alfred" in local server @ port 7000
      VShopper shopper = new VShopper( "localhost:7000/Alfred" );
      shopper.saveNow(); // become persistent, save copy to database

      // ask the shopper to use the registry to find the best price of product
      shopper.findBestPriceFor( "Killer Rabbits", registry );

      // shutdown program
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

The first line of `Shop.java` uses the static method `VObject.forObjectAt()` to obtain a virtual reference to the existing remote `Vector` named `Registry`.

The second and third lines of `Shop.java` create a persistent `Shopper` with alias `Alfred`. Unlike a simple (non-agent) object, an agent lives for one day by default. This allows an agent to roam a network and perform duties without requiring any local or virtual references. The Voyager User Guide explains how you can specify an agent's life span so that the agent can either live forever or die when it has no local or virtual references.

The fourth line of `Shop.java` instructs `Alfred` to find the best price for the *Killer Rabbits* video. When `Alfred` receives the message `findBestPriceFor()`, he saves the name of the product for future use and then executes the following code:

```
    moveTo( registry, "atRegistry" );
```

This function deactivates `Alfred`, moves him from server 8000 to the registry in server 7000, and then reactivates him by sending him the message `atRegistry()` with the registry as its single argument. Because `Alfred` is persistent, his database backup copy is automatically moved between servers when `Alfred` moves. When `Alfred` arrives at server 7000, the original call to `findBestPriceFor()` returns, and the `Shop.java` program terminates. `Alfred`, however, continues to execute in the Voyager server. He stores a clone of the registry and then executes the following code:

```
moveTo( (VStore) registry.elementAt( index ), "shop" );
```

This causes `Alfred` to move to the first store (located in server 8000) and execute `shop()` with the store as its single argument. This function gets the price of the product from the store, updates the variable `bestStore` if appropriate, and then moves `Alfred` to the next store. This sequence continues until the itinerary is exhausted, at which point `Alfred` executes the following code:

```
moveTo( bestStore.getProgramAddress(), "park" );
```

This variation of `moveTo()` causes `Alfred` to move into the program that holds the best store and then execute `park()` with no arguments. The `park()` method displays a status message and flushes the final state of `Alfred` to the local database. When `park()` completes, `Alfred`'s thread of execution finishes, but `Alfred` does not die.

Step 3. Create a virtual version of Shopper.

Use `vcc` to create a virtual version of `Shopper`.

```
>vcc Shopper
vcc 1.0.0, copyright objectspace 1997
>dir VShopper.*
VSHOPP~1 JAV         5,926  08-25-97  9:37a VShopper.java
>
```

Step 4. Compile the Phase 2 programs.

Use the `javac` command to compile the Phase 2 source files.

```
javac Shopper.java VShopper.java Shop.java
```

Step 5. Restart the Voyager servers.

Restart a Voyager server on each of ports 7000 and 8000 in two different windows. As in Phase 1, use the `-d` option to load the handles of all persistent objects in a database.

Window 1

```
>voyager 7000 -d 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 1 object, 0 classes
```

Window 2

```
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 2 objects, 0 classes
```

Step 6. Run Shop.class to launch the shopper.

Run `Shop.class` in a third window. This program launches `Alfred` and then immediately terminates. The following output is displayed.

Window 3

```
>java Shop
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1191
>
```

As `Alfred` moves from server to server to find the best price, additional output displays in the first two windows.

Window 1

```
>voyager 7000 -d 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 1 object, 0 classes
at Store( MegaHits ), Killer Rabbits is $35
```

Window 2

```
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 2 objects, 0 classes
shopping using itinerary: [Store( VideoHeaven ), Store( MegaHits )]
at Store( VideoHeaven ), Killer Rabbits is $25
at Store( VideoHeaven ),  best price $25
shopper parks at 208.6.239.200:8000
```

After `Alfred` finds the best price for *Killer Rabbits,* he parks in server 8000 to await further instructions. Phase 2 is now complete. Because the stores, registry, and shopper are all persistent, the Voyager servers could be shut down and restarted at this point without causing problems.

Phase 3: Buying an Item

In the final phase of the shopping system project, a program is created that contacts `Alfred`, asks for the best store, tells `Alfred` to die, and then makes a remote purchase from the store. When `Alfred` dies, his resources are reclaimed by the local Java virtual machine. Phase 3 is comprised of the following steps:

1. Write a program called `Buy.java` that uses `Alfred`'s recommendation to purchase a product.
2. Compile `Buy.java`.
3. Run `Buy.class`.

Step 1. Write a program called Buy.java that uses Alfred's recommendation to purchase a product.

The following is the `Buy.java` source code:

Application voyager1.0.0\examples\shopper\Buy.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;

public class Buy
  {
  public static void main( String[] args )
    {
    try
      {
      // connect to Alfred, whose last known location was server @ port 7000
      VShopper shopper =
        (VShopper) VObject.forObjectAt( "localhost:7000/Alfred" );

      // ask the shopper for the best store, waiting if not ready yet
      VStore bestStore = getBestStore( shopper );

      // tell the shopper to die
      System.out.println( "sorry Alfred, but i have to kill you now" );
      shopper.dieNow(); // kill and remove from database

      // buy the product
      int price = bestStore.buy( "Killer Rabbits" );
      System.out.println( "bought video for $" + price + " @ " + bestStore );

      // shutdown program
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }

  /**
   * Get the best store from the agent, waiting until the store is available
   */
  static VStore getBestStore( VShopper shopper ) throws VoyagerException
```

```
    {
    while( true )
      {
      try
        {
        return shopper.getBestStore();
        }
      catch( IllegalStateException exception )
        {
        System.out.println( "Shopper not parked yet" );
        try { Thread.sleep( 2000 ); } catch( InterruptedException ie ) {}
        }
      }
    }
  }
```

Step 2. Compile Buy.java.

Use the `javac` command to compile the Phase 3 program.

```
  javac Buy.java
```

Step 3. Run Buy.class.

The `Buy.java` program connects to `Alfred`, using his last known location at server 7000, even though by this time he has moved to the best store in server 8000. This is possible due to the trail of forwarders `Alfred` leaves behind as he moves. `Buy.java` then attempts to obtain a virtual reference to the best store. If `Alfred` has not yet parked, an exception is thrown, which the `Buy.java` program catches. `Buy.java` continues to attempt to get a virtual reference to the best store. When successful, the program kills `Alfred` (removing him from the local database) and makes the purchase.

Running `Buy.java` generates the output below.

```
>java Buy
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1195
sorry Alfred, but i have to kill you now
bought video for $25 @ Store( VideoHeaven )
>
```

This concludes the guided tour.

Product Evolution

Many people ask us, “If Voyager is free, then how will you make money?” We believe that years from now, companies will not make much money selling basic middleware. DCOM will be embedded and distributed everywhere. CORBA price points are already plummeting. In the not-too-distant future, the bulk of the features currently in the Voyager Core Technology will be freely available in several forms and locations. Your cost is in development time, and your revenues are increasingly dependent on time to market. ObjectSpace believes Voyager’s Java-centric binding, advanced mobile object features and innovative services provide the best basis for rapid development of distributed systems in Java.

As the industry changes, ObjectSpace will continue to develop and sell partnerships, support, and other services, but will also begin to unveil more and more next-generation add-on features for the ObjectSpace Voyager platform. These add-ons will progress in areas of security, group communication, and persistence concurrency and will deliver the same time-to-market and rapid development advantages found in the Voyager Core Technology today. Unlike the Voyager Core Technology, these add-ons will not be free.

ObjectSpace is also pursuing several partnerships for the creation of technology integrations and enhanced development tools. Our relationships, based on the deployment of JGL, will enable the rapid distribution, adoption, and integration of the ObjectSpace Voyager platform.

As you look further into the future, you will see ObjectSpace using the Voyager technology base as the platform for its own next-generation product lines. As definite product release dates approach, we will announce these longer-term projects. Be assured that we will use the advantages of Voyager, such as agent technology, to deliver products that, until now, you have only speculated about.

For additional information on Voyager, visit the ObjectSpace Web site at www.objectspace.com. There you will find several additional white papers, user stories, and of course, the complete Voyager Core Technology download. This download includes a comprehensive user guide that covers additional details on the Voyager 1.0 feature set.

ObjectSpace also offers several packaged services to help you in the evaluation, adoption, and use of ObjectSpace Voyager.

- ObjectSpace Voyager Core Technology Support
- ObjectSpace Voyager Core Technology Training
- ObjectSpace Voyager Platinum Partners Program
- ObjectSpace Voyager QuickStart Adoption Program

For additional information or to purchase any of these packaged services, contact us at our North American corporate offices.

OBJECT SPACE

14881 Quorum Drive, Suite 400
Dallas TX 75240

972.726.4500
1.800.OBJECT.1
Fax: 972.715.9099

E-mail: sales@objectspace.com
Web: www.objectspace.com