

# D: A Language Framework for Distributed Programming

Cristina Videira Lopes

PhD Thesis, College of Computer Science, Northeastern University. November 1997.

© Copyright 1997 Xerox Corporation. All rights reserved.

# DJ: Primer

---

Version 0.3  
June 1997

by Cristina Videira Lopes

XEROX  
Palo Alto Research Center  
3333 Coyote Hill Road  
Palo Alto, CA 94304

© Copyright 1997 Xerox Corporation

## I Introduction

---

DJ is an object-oriented language framework designed for facilitating the development and maintenance of concurrent and distributed applications. It uses the aspect oriented programming approach [37] to allow the code for the basic functionality of a distributed application to be written without having to explicitly deal with remote interactions and synchronization. Separate code deals with those issues.

This guide describes how to program in DJ. While using this guide, be sure to have two other documents at hand: “DJ: Framework Specification” and your favorite Java manual.

### 1 *Overview of DJ*

DJ consists of three relatively independent languages:

- 1) one full object-oriented language, JCore, for programming functional components;
- 2) one small language, COOL, for programming the aspect of thread synchronization over the execution of the components; and
- 3) one small language, RIDL, for programming the aspect of remote interactions between components.

JCore is Java™ 1.0 minus the following:

- 1) the keyword `synchronized`
- 2) the `Object` methods `wait`, `notify` and `notifyAll`
- 3) method overloading (method overriding is allowed!)

Everything else of Java, including the libraries, is available to DJ programs. However, the aspect modules of DJ (coordinators and portals) can only be applied to user programmed JCore classes, not to Java library classes.

### 2 *Developing Programs in DJ*

While concurrency and distribution should be taken into account right from the early stages of the *design* of a DJ application, the *implementation* of the application usually includes a number of iterations on the coding of the components and the aspects, alternately. First, the programmer should concentrate on getting the *functionality* right, independently of concurrency or distribution. That is, what the JCore classes do, the operations they provide, the composition between classes,

etc. Much of the testing of the functionality can be done at this stage, without concurrency or distribution.

Then the programmer introduces the necessary concurrency (i.e. creation of threads) and distribution (i.e. many virtual machines), and with them the aspect programs in COOL and RIDL. In most cases, the issues related to concurrency should be introduced and tested before distributing the application over the network. After the aspect programs are introduced, it may be necessary or desirable to make small modifications in the implementation of the classes. (Beware of premature optimizations, though!).

As the functional and operational requirements of the application evolve, the process repeats: 1) think about the implementation of the classes, as independent as possible from concurrency and distribution issues; 2) introduce those issues and modify/extend the aspect programs; and 3) tune the classes, if necessary.

## II Concurrency: Programming in COOL

---

COOL provides the means for dealing with mutual exclusion of threads, synchronization state, guarded suspension and notification, in relative separation from the method code. Coordination programs consist of a set of coordinator modules which are associated with the classes on a name basis. A coordinator may coordinate more than one class at the same time. The smallest blocks for synchronization are the methods.

### 1 Basic Example: The Bounded Buffer

This is the classical example of synchronization in concurrent systems. It models the situation in which a number of concurrent clients tries to access a limited shared resource. A bounded buffer maintains a fixed array of elements, and its clients concurrently put and take elements from it. So, a basic interface to these objects is:

---

```
public interface BoundedBuffer {
    public int capacity(); // invariant: capacity >= 0
    public int count();   // invariant: 0 <= count <= capacity
    public void put(Object x) throws Full;
    public Object take() throws Empty;
}
```

---

There are many ways of implementing bounded buffers. Let's start by the simplest one, which doesn't even consider issues of synchronization:

---

```
public class BoundedBufferV1 {
    private Object array[];
    private int takePtr = 0, putPtr = 0;
    protected int usedSlots = 0, size;

    BoundedBufferV1(int capacity) throws IllegalArgumentException {
        if (capacity <= 0) throw new IllegalArgumentException();
        array = new Object[capacity];
        size = capacity ;
    }

    public int count() { return usedSlots; }
    public int capacity() { return size; }

    public void put(Object x) throws Full {
        if (usedSlots == size) throw new Full();
        array[putPtr] = x;
        putPtr = (putPtr + 1) % size;
        usedSlots++;
    }
}
```

```

public Object take() throws Empty {
    if (usedSlots == 0) throw new Empty();
    Object old = array[takePtr];
    takePtr = (takePtr + 1) % size;
    usedSlots--;
    return old;
}
}

```

---

A client of the bounded buffer may be something like:

```

public class Client implements Runnable {
    BoundedBufferV1 buf;
    Random sleeptime=new Random ();
    boolean done=false;
    boolean shouldInput=false; // Indicates client is a producer

    public Client(BoundedBufferV1 b, boolean putp){
        buf=b;
        shouldInput=putp;
    }

    public void run(){
        while (!done){
            int x=sleeptime.nextInt();
            try {
                if (shouldInput) buf.put(new Integer(x)); // put if producer
                else buf.take(); // else take
            } catch (BufferException e) {};
            try{
                Thread.sleep(Math.abs(sleeptime.nextInt() % 500));
            }catch(InterruptedException e){};
        }
    }
    public void finish(){
        done=true;
    }
}

```

---

If there is only client, then it sequentially executes the loop of random puts and takes, and everything works fine, except for occasional exceptions for when the buffer is full or empty.

When there are multiple concurrently clients inserting and removing objects from the same buffer, however, those requests need to be synchronized, because the internal variables of the bounded buffer are modified in the implementation of `put` and `take`; therefore there may be temporary inconsistencies within the buffer while these methods are running. To synchronize the access to the execution of the methods, we use COOL. For this particular implementation of the bounded buffer, we can define a coordinator as follows:

```

coordinator BoundedBufferV1 {
    selfex {put, take};
    mutex {put, take};
}

```

---

This declaration establishes an association between class `BoundedBufferV1` and this coordinator module. The body of this coordinator states that

- both methods `put` and `take` are self-exclusive, that is, if two threads try to execute `put` at the same time, one of them waits until the other is finished; the same for `take`.
- methods `put` and `take` are also mutually exclusive, that is, if one thread tries to execute `put` and another thread tries to execute `take` at the same time, one of them must wait until the other is finished.
- since `count` and `capacity` are not mentioned, their execution is not synchronized, and they always get executed, no matter what other executions there may be on the object.

The above coordinator, then, takes care of synchronizing the threads that try to execute the methods of the bounded buffer, guaranteeing the proper exclusion constraints.

But in the concurrent environment, if the buffer is full or empty we can make the threads wait until the buffer is not full or not empty, respectively — since other threads may eventually remove or insert, respectively, elements from the buffer. In COOL, this is done through the use of method managers that establish pre-conditions and modify special state that belongs to the coordinator. We can enhance the above coordinator as follows:

---

```

coordinator BoundedBufferV1 {
  selfex {put, take};
  mutex {put, take};
  condition empty = true, full = false; // coordination state with
                                         // initial values
  put: requires !full; // pre-condition; wait if false
      on_exit {
        // as soon as put finishes, change state accordingly
        if (empty) empty = false;
        if (usedSlots == size) full = true;
      }
  take: requires !empty; // pre-condition; wait if false
      on_exit {
        // as soon as take finishes, change state accordingly
        if (full) full = false;
        if (usedSlots == 0) empty = true;
      }
}

```

---

Note that coordinators have access to the variables defined in the classes they coordinate (in this case, `usedSlots` and `size`). But that's as much as they can do directly on the classes. They cannot assign values to those variables; and they cannot invoke methods.

Note also that this particular coordination strategy guarantees that no thread will ever remove objects from an empty buffer or insert objects on a full buffer. Therefore, assuming that the class will always be used along with its coordinator, on a later phase we can go back to the implementation of the class and remove the guard tests. In general, coordinators *add* constraints to the execution of the methods; hence, some failure states in a class implementation may be eliminated when a coordinator is associated with that class. The rule, however, is that you shouldn't do these optimizations unless you are 100% sure that the coordinator will always be there.

## 2 Subclassing and COOL

Coordinators are inherited by subclasses. Consider the following subclass of `BoundedBufferV1` defined in §1, which stores the objects in a linked list, instead of an array:

---

```

public class BoundedBuffer2 extends BoundedBufferV1 {
    private ObjectList olist; // the elements, implemented as a list
    private Object putPtr, takePtr; // "pointers"
    public BoundedBuffer2 (int size) {
        olist = new ObjectList(); size = capacity;
        takePtr = putPtr = olist;
    }
    public void put(Object o) { // override the implementation
        putPtr.insert(o);
        putPtr = putPtr.next;
        ++usedSlots;
    }
    public Object take() { // override the implementation
        Object old;
        old = takePtr.remove();
        takePtr = takePtr.next;
        --usedSlots;
        return old;
    }
}
public class ObjectList {
    Object o;
    public ObjectList next;
    public void insert (Object x) {
        this.o = x; this.next = new ObjectList();
    }
    public Object remove() {
        return this.o;
    }
}

```

---

The coordinator of the `BoundedBufferV1` applies to all of its subclasses, namely to `BoundedBufferV2`. For methods that are simply inherited but not redefined, it is relatively easy to understand the reason why: when instances of `BoundedBufferV2` are invoked for `capacity` or `count`, the actual method invoked is the one defined in the superclass. However, `put` and `take` are overridden here. Nevertheless, their execution is affected by the coordination strat-

egy defined in the coordinator of the superclass, because they match the *name*. That is, coordinators manage method names, not the methods themselves.

But class inheritance does not necessarily imply that the coordination of a superclass is appropriate for the subclass. Coordination is relatively dependent on the implementation of the classes. Consider this other implementation:

---

```

public class BoundedBufferV3 extends BoundedBufferV1 {
    private Object array[];
    private int takePtr = 0, putPtr = 0;
    protected emptySlots;

    BoundedBufferV1(int capacity) throws IllegalArgumentException {
        if (capacity <= 0) throw new IllegalArgumentException();
        array = new Object[capacity];
        emptySlots = size = capacity ;
    }

    public int count() { return usedSlots; }
    public capacity() { return size; }

    public void put(Object x) {
        do_put(x);
        increment_usedSlots();
    }
    public Object take() {
        Object x = do_take();
        increment_emptySlots();
        return x;
    }
    private void do_put(Object x) throws Full {
        if (emptySlots == 0) throw new Full();
        array[putPtr] = x;
        putPtr = (putPtr + 1) % size;
        emptySlots--;
    }
    private Object do_take() throws Empty {
        if (usedSlots == 0) throw new Empty();
        Object old = array[takePtr];
        takePtr = (takePtr + 1) % size;
        usedSlots--;
        return old;
    }
    private void increment_usedSlots() {usedSlots++;}
    private void increment_emptySlots() {emptySlots++;}
}

```

---

The inherited coordination is still valid for this implementation (i.e. we can use it, and nothing bad will happen). However, a careful analysis of this code leads to the conclusion that the granularity of the synchronization can be finer for this class. There is no need to synchronize on the top put and take, but rather we can synchronize the private methods in a more efficient way:

---

```

coordinator BoundedBufferV3 {
  selfex {do_take, do_put, increment_emptySlots, increment_usedSlots};
  mutex {do_take, increment_usedSlots};
  mutex {do_put, increment_emptySlots};

  do_take: requires !empty;
             on_exit { if (full) full = false; }
  do_put:   requires !full;
             on_exit { if (empty) empty = false; }
  increment_emptySlots:
             on_exit { if (emptySlots == size) empty = true; }
  increment_usedSlots:
             on_exit { if (usedSlots == size) full = true; }
}

```

---

Associating this coordinator with class `BoundedBufferV3` overrides the inherited coordinator. Therefore in this class hierarchy, `BoundedBufferV1` and `BoundedBufferV2` are affected by `BoundedBufferV1`'s coordinator, but `BoundedBufferV3` is affected by its own coordinator.

### 3 *The Dining Philosophers: the Classical Monitor Solution*

This is the other classical example of synchronization. It models the situation in which a number of concurrent clients tries to access a number of shared resources, and can only proceed if *all* the necessary resources are available. A number of philosophers are sitting around a table, eating spaghetti and thinking, alternately. Between each pair of neighbor philosophers there is exactly one fork. Before eating, each philosopher picks both left and right forks, and after eating he puts down the forks. The synchronization, then, is that each philosopher can only start eating if both left and right forks are free (that is, if their neighbors are not eating). So, the functionality can be implemented by the following class:

---

```

public class Philosopher implements Runnable {
  // the global set up
  static final int max = 5;
  static protected Fork forks[max];
  static protected int count = 0;
  // for each philosopher
  protected int      mynumber;
  protected Fork     left, right;
  protected Random   sleeptime = new Random ();
  protected boolean done = false;
  Philosopher() throws MaxPhilosophers {
    if (count == max) throw new MaxPhilosophers();
    left = forks[count];
    right = forks[(count + 1) % max];
    mynumber = count++;
  }
}

```

---

```

}

public void run() {
    while (!done) {
        think();
        eat();
    }
}

public void finish() {done = true;}

private void think() {
    int x=sleeptime.nextInt();
    try{
        Thread.sleep(Math.abs(sleeptime.nextInt() % 500));
    } catch(InterruptedException e){};
}

private void eat() {
    // do something with the forks
    int x=sleeptime.nextInt();
    try{
        Thread.sleep(Math.abs(sleeptime.nextInt() % 500));
    } catch(InterruptedException e){};
    // do something else with the forks
}
}

```

---

We need to synchronize the access to method eat. Any book on concurrent systems presents at least the monitor solution. In COOL, that solution looks like:

```

per_class coordinator Philosopher {
    condition OKToEat[] = {true,true,true,true,true};
    boolean eating[] = {false,false,false,false,false};

    eat: requires OKToEat[mynumber];
    on_entry {
        OKToEat[(mynumber+1) % max] = false;
        OKToEat[(mynumber-1) % max] = false;
        eating[mynumber] = true;
    }
    on_exit {
        if (eating[(mynumber+2) % max] == false)
            OKToEat[(mynumber+1) % max] = true;
        if (eating[(mynumber-2) % max] == false)
            OKToEat[(mynumber-1) % max] = true;
        eating[mynumber] = false;
    }
}

```

---

#### 4 The Dining Philosophers: another COOL Solution

In DJ we can do the synchronization in yet a different way, by taking advantage of inheritance. In the previous set up, there was only one class, which was instantiated exactly five times. In this set

up, we define five subclasses of `Philosopher`, each one instantiated exactly once, and then we coordinate those five instances distinguishing them by their class names.

---

```
public class Philosopher0 extends Philosopher {
    static boolean exactlyOne = false;
    Philosopher0() throws MaxInstances {
        if (exactlyOne) throw new MaxInstances();
        exactlyOne = true;
        super();
    }
}
// ...
public class Philosopher4 extends Philosopher {
    static boolean exactlyOne = false;
    Philosopher4() throws MaxInstances {
        if (exactlyOne) throw new MaxInstances();
        exactlyOne = true;
        super();
    }
}
}
```

---

In this set up, we use a multiple-class coordinator that coordinates the five philosopher classes (and, consequently, the five philosopher instances), and we can do the synchronization simply with mutual exclusion:

---

```
per_class coordinator Philosopher0, Philosopher1, Philosopher2,
    Philosopher3, Philosopher4 {
    mutex {Philosopher0.eat, Philosopher1.eat};
    mutex {Philosopher1.eat, Philosopher2.eat};
    mutex {Philosopher2.eat, Philosopher3.eat};
    mutex {Philosopher3.eat, Philosopher4.eat};
    mutex {Philosopher4.eat, Philosopher0.eat}
}
```

---

## 5 An Assembly Line

Consider an assembly line for making candy packets, as depicted in Figure 41. Candy makers make candy, one piece at a time, and there can be many of them. They pass each candy they make to a packer (*newCandy*), which accumulates a number of candy for producing a packet. When the packet is ready, the packer passes it to the finalizer (*newPack*), which takes also a label (*newLabel*) from the label maker, glues the label on the packet and outputs the final packet. All this is done concurrently, i.e. each participant works on its own, and they only synchronize at certain points.

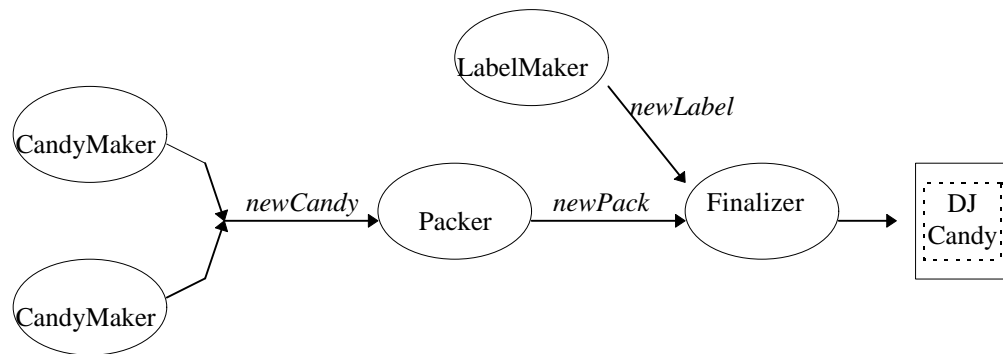


Figure 41. An assembly line for making candy packets.

The points of synchronization can be described as follows. When the packet in the packer is full, and while the packer is processing the packet, the candy makers must wait before passing any more new candy to the packer. As soon as the finalizer gets the packet from the packer, the packer restarts collecting candy from the candy makers into a new packet. The finalizer must wait for one packet and one label, and while it is gluing the label in the packet, both the packer and the label maker must wait before passing new items to the finalizer.

The remainder of this section shows the highlights of an implementation of this assembly line in DJ. First, the most important classes are presented. The coordinator is shown at the end.

Class CandyMaker:

---

```

public class CandyMaker implements Runnable {
    protected Random worktime = new Random ();
    protected Packer thePacker = null;
    CandyMaker(Packer p) {
        thePacker = p;
    }

    public void run() {
        Candy aCandy = null;
        while (!done) {
            aCandy = makeCandy();
            thePacker.newCandy(aCandy);
        }
    }

    public void finish() {done = true;}

    private Candy makeCandy() {
        Candy aCandy = new Candy();
        try {
            Thread.sleep(Math.abs(worktime.nextInt() % 500));
        } catch (InterruptedException e) {};
        return aCandy;
    }
}
  
```

---

Class Packer:

---

```

public class Packer implements Runnable {
    static final int maxPackers = 1;
    static int count = 0;
    static int nCandyPerPack = 50;

    protected int myNumber = 0;
    protected Random worktime = new Random ();
    protected Finalizer theFinalizer = null;
    private Pack candyPack = null;
    private int nCandy = 0;

    Packer(Finalizer f) throws MaxInstances {
        if (count == maxPackers) throw new MaxInstances();
        myNumber = count++;
        theFinalizer = f;
    }

    public void run() {
        Pack candyPack = null;
        while (!done) {
            candyPack = new Pack(nCandyPerPack);
            nCandy = 0;
            processPack(candyPack);
            theFinalizer.newPack(candyPack);
        }
    }

    public void finish() {done = true;}

    public void newCandy(Candy aCandy) throws Full {
        if (nCandy == nCandyPerPack) throw new Full();
        candyPack.put(aCandy);
        nCandy++;
    }

    private void processPack() {
        try {
            Thread.sleep(Math.abs(worktime.nextInt() % 500));
        } catch (InterruptedException e) {};
    }
}

```

---

Class LabelMaker:

---

```

public class LabelMaker implements Runnable {

    protected int myNumber = 0;
    protected Random worktime = new Random ();
    protected Finalizer theFinalizer = null;

    LabelMacker(Finalizer f) {
        theFinalizer = f;
    }

    public void run() {
        Label aLabel = null;
        while (!done) {
            aLabel = makeLabel();
            theFinalizer.newLabel(aLabel);
        }
    }
}

```

---

```

    }
}

public void finish() {done = true;}

private Label makeLabel() {
    Label aLabel = new Label();
    try {
        Thread.sleep(Math.abs(worktime.nextInt() % 500));
    } catch (InterruptedException e) {};
    return aLabel;
}
}
}

```

---

### Class Finalizer:

```

public class Finalizer implements Runnable {
    static final int maxFinalizers = 1;
    static int count = 0;

    protected int myNumber = 0;
    protected Random worktime = new Random ();
    private Pack thePack = null;
    private Label theLabel = null;

    Finalizer() throws MaxInstances {
        if (count == maxFinalizers) throw new MaxInstances();
        myNumber = count++;
    }

    public void run() {
        while (!done) {
            glueLabelToPack();
            newDJCandyPack();
        }
    }

    public void finish() {done = true;}

    public void newPack(Pack aPack) {
        thePack = aPack;
    }

    public void newLabel(Label aLabel) {
        theLabel = aLabel;
    }

    private void glueLabelToPack() {
        try {
            Thread.sleep(Math.abs(worktime.nextInt() % 500));
        } catch (InterruptedException e) {};
    }

    private void newDJCandyPack() {
        System.out.println ("New DJ Candy Pack!");
    }
}
}

```

---

Finally, the coordinator:

---

```

coordinator Packer, Finalizer {

    selfex {Packer.newCandy};

    condition packFull = false, gotPack = false, gotLabel = false;

    Packer.newCandy: requires !packFull;
        on_exit { if (nCandy == nCandyPerPack) packFull = true; }

    Packer.processPack: requires packFull;

    Finalizer.newPack: requires !gotPack;
        on_entry { gotPack = true; }
        on_exit { packFull = false; }

    Finalizer.newLabel: requires !gotLabel;
        on_entry { gotLabel = true; }

    Finalizer.glueLabelToPack: requires (gotPack && gotLabel);

    Finalizer.newDJCandyPack:
        on_exit {gotPack = false; gotLabel = false;}
}

```

---

Note that this coordinator assumes that there is only one instance of the packer and only one instance of the finalizer. Otherwise, the coordination is incorrect — since the condition variables are simple variables, and different instances of packers and finalizers would conflict when modifying the coordination state. If more instances exist, the condition variables should be arrays of condition variables (see Philosophers).

### III Distribution: Programming in RIDL

---

RIDL provides the means for dealing with data transfers between different execution spaces in relative separation from the classes. RIDL programs consist of a set of portal definitions which are associated with the classes on a name basis. Portals are helpers with respect to the implementation of the classes: they take care of data transfers across space boundaries.

#### 1 *Remote Objects*

In DJ, some objects may be promoted to being “remote objects.” Remote objects are ordinary objects that can be invoked from other execution spaces. That is, a remote object can be invoked both locally and remotely.

For an object to be also a remote object, the programmer must define, along with the class, a *remote interface* to instances of that class. A portal identifies the subset of methods of the class that can be invoked remotely (the remote operations), and the parameters and return values that those remote operations take. For each of the parameters and return values, an optional mode may be supplied that describes how the data transfers are to be made between space where the remote object lives and the spaces where its clients live. A portal is, therefore, a contract to which communicating execution spaces agree.

From a client’s perspective, invocation to remote objects has exactly the same form as invocation to local objects: *obj.method(args)*; that is invocation is location-transparent. *obj* may be bound either to a local or to a remote object. When *obj* is bound to a local object, then an ordinary local invocation occurs. When *obj* is bound to a remote object, then a remote method invocation occurs. In this case, the method invocation must comply with the object’s portal. This implies two things:

- 1) The data transfer is done according to the remote object’s portal.
- 2) The run-time exception `DJInvalidRemoteOp` may occur.

#### 2 *The Name Server*

A noticeable difference between a non-distributed and a distributed environment, is that in the latter there is no automatic way of binding objects that live in different execution spaces. Therefore an additional bootstrap is necessary. The most common mechanism for doing this is through a Name Service that is implemented by one or more name server objects. A name server object is a remote

object that maps names (i.e. strings) to remote object references. The name service in DJ is the one provided by Java RMI [27], but with a special DJ-specific portal to it:

---

```
portal DJNaming {
    // Associate the given URL with the given DJ object
    void bind(String url, Object obj) {
        obj: gref;
    };

    // Same as bind, but if the an association to the same url exists,
    // associate the url with the new object
    void rebind(String url, Object obj) {
        obj: gref;
    };

    // Lookup a DJ remote object that is associated with the given name
    Object lookup(String url) {
        return: gref;
    };

    String[] list();
}

```

---

DJNaming is the DJ-specific wrapper class that interacts with Java RMI's Naming class. The functionality of the DJNaming class is the same as Java RMI's Naming class. From the Java RMI reference manual: "The java.rmi.Naming class allows remote objects to be retrieved and defined using the familiar Uniform Resource Locator (URL) syntax. The URL consists of protocol, host, port, and name fields. [...] The protocol should be specified as rmi, as in rmi://java.sun.com:2001/root." Not all the fields need to be present.

Here's an example, of how to bind and look up remote objects:

---

```
BoundedBuffer1 bb = new BoundedBuffer(100);
String url = "rmi://parc.xerox.com/BoundedBuffer";
// bind url to remote object
DJNaming.bind(url, bb);
...
// lookup bounded buffer
bb = (BoundedBuffer)DJNaming.lookup(url);

```

---

### 3 Basic Example: The Distributed Bounded Buffer

The most simple distributed application is the one that consists of one "server" object and multiple clients that access it from other execution spaces. Let's reuse the bounded buffer example of the previous section to demonstrate how this is done in DJ.

### *Functionality*

The distributed bounded buffer has exactly the same functionality as the non-distributed bounded buffer, i.e. it maintains a fixed array of elements, and its clients, local or remote, concurrently put and take elements from it. So, we can reuse the exact same class. But to make certain points more clear, let's make the internal buffer to be an array of particular objects, say books. We need to change the signatures of the methods:

---

```

public class BoundedBufferV1 {
    private Book array[];
    private int takePtr = 0, putPtr = 0;
    protected int usedSlots = 0, size;

    BoundedBufferV1(int capacity) throws IllegalArgumentException {
        if (capacity <= 0) throw new IllegalArgumentException();
        array = new Object[capacity];
        size = capacity ;
    }

    public int count() { return usedSlots; }
    public int capacity() { return size; }

    public void put(Book x) throws Full {
        if (usedSlots == array.length) throw new Full();
        array[putPtr] = x;
        putPtr = (putPtr + 1) % size;
        usedSlots++;
        System.out.println("BB got book:");
        b.print();
    }

    public Book take() throws Empty {
        if (usedSlots == 0) throw new Empty();
        Book old = array[takePtr];
        takePtr = (takePtr + 1) % size;
        usedSlots--;
        return old;
    }
}

public class Book {
    private int isbn = 0;
    private String title = null;
    Book(int n, String t) {isbn = n; title = t;}
    public void print() {
        System.out.println ("Book: " + isbn + title);
    }
}

```

---

### *Portal*

But for bounded buffers to be remote objects, we need to define their portal. Let's define it as follows:

---

```
portal BoundedBufferV1 {
    int capacity();
    void put(Book x);
    Book take();
}
```

---

This declaration establishes an association between class `BoundedBufferV1` and this portal module. The body of this portal states that

- methods `capacity`, `put` and `take` are remote operations.
- method `count` defined in the class is not a remote operation.
- the `Book` argument to `put` and the return `Book` object of `take` are to be passed according to the default transfer strategy, which is by deep copy. This means that when a client invokes `put` with a book object as an argument, the bounded buffer gets a replica of that book, and not the book object itself. And when a client invokes `take`, it gets a replica of the book that lives in the bounded buffer space.

The above coordinator, then, takes care of limiting the access that remote clients have to bounded buffers, and of establishing the data transfer strategies between the communicating execution spaces.

### *Exporting the object reference*

The application that instantiates a bounded buffer should export its reference to the name server. For example:

```
public class StartBuffer {
    public static void main(String args[]) {
        BoundedBufferV1 bb = new BoundedBufferV1(100);
        try {
            DJNaming.bind("rmi://goblin/BB", bb);
        } catch (Exception e) {
            System.out.println("StartBuffer err: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

---

From this point on, clients all over the network may get the reference to the bounded buffer object that was instantiated in this applet.

*Client*

A client of the remote bounded buffer is exactly like the client of a local bounded buffer:

---

```

public class Client implements Runnable {
    BoundedBufferV1 buf;
    Random sleeptime = new Random ();
    boolean done = false;
    boolean shouldInput = false; // Indicates client is a producer

    public Client(BoundedBufferV1 b, boolean putp){
        buf = b;
        shouldInput = putp;
    }

    public void run(){
        Book b;
        while (!done){
            int x=sleeptime.nextInt();
            try {
                if (shouldInput) {
                    b = new Book(x, "aBook" + new Integer(x).toString());
                    buf.put(b); // if producer, put
                }
                else { // else take
                    b = buf.take();
                    System.out.println("Client got book:");
                    b.print();
                }
            } catch (BufferException e) {};
            try{
                Thread.sleep(Math.abs(sleeptime.nextInt() % 500));
            }catch(InterruptedExpection e){};
        }
    }
    public void finish(){
        done=true;
    }
}

```

---

But whoever instantiates clients must first fetch the reference to the remote bounded buffer:

---

```

public class StartClient {
    public static void main (String args[]) {
        BoundedBufferV1 bb;
        bb = (BoundedBufferV1)DJNaming.lookup("rmi://goblin/BB");
        new Thread(new Client(bb)).start();
    }
}

```

---

*Running the App*

To run this distributed application, we need to first run the `StartBuffer` class and then, in a separate shell, we can run a `StartClient` classes. We should see the output messages displaying the books, both in the bounded buffer terminal and in the client terminals.

#### 4 *Multiple clients*

We can run multiple clients in different execution spaces. In that case, we may end up with concurrency on the bounded buffer space. Hence, we need to synchronize the method invocations by using, for example, the coordinator shown in page 213.

#### 5 *Passing Remote References*

In the previous example, the book objects are passed by copy. Let's change the portal of the BoundedBufferV1 class, so that the books are passed by global reference:

---

```
portal BoundedBufferV1 {
    int capacity();
    void put(Book x) {
        x: gref;
    }
    Book take() {
        return: gref;
    }
}
```

---

But now, books may also be remote objects. Therefore, they need a portal:

---

```
portal Book {
    void print();
}
```

---

After recompiling the bounded buffer and the clients of the bounded buffer, we can run the same set up. In this case, the book arguments and return objects are not replicated, and only their global references are passed. Therefore invocations to the print method will be displayed in within the space where the books were instantiated.