

declared as **final**, which means that the variable can be assigned exactly once. Final variables must be assigned before they are used.

Java introduces its own terminology for some concepts. Classes contain fields (attributes, in our terminology), methods, other classes, or interfaces, which are known collectively as class members. Variables are either fields, local variables (declared in code blocks local to methods), or method parameters. Variables are declared by giving their type, identifier, and an optional set of modifiers (e.g., **final**). The **self** concept is called **this**.

Interfaces

Java has an elegant solution to the problems of multiple inheritance (Sections 7.4.4 and 7.4.5). Java introduces the concept of *interface*, which syntactically looks like a class with only method declarations. An interface has no implementation. A class can implement an interface, which simply means that it defines all the methods in the interface. Java supports single inheritance for classes, thus avoiding the problems of multiple inheritance. But, to preserve the advantages of multiple inheritance, Java supports multiple inheritance for interfaces.

Java supports higher-order programming in a trivial way by means of the encoding given in Section 7.5.2. In addition to this, Java has more direct support for higher-order programming through inner classes. An *inner class* is a class definition that is nested inside another class or inside a code block (such as a method body). An instance of an inner class can be passed outside of the method body or code block. An inner class can have external references, but there is a restriction if it is nested in a code block: in that case it cannot reference non-final variables. We could say that an instance of an inner class is *almost* a procedure value. The restriction likely exists because the language designers wanted non-final variables in code blocks to be implementable on a stack, which would be popped when exiting the method. Without the restriction, this might create dangling references.

7.7.2 Introduction to Java programming

We give a brief introduction to programming in Java. We explain how to write a simple program, how to define classes, how to use inheritance, and how to write concurrent programs with locks and monitors. We situate the Java style with respect to the computation models of this book.

This section only scratches the surface of what is possible in Java. For more information, we refer the reader to one of the many good books on Java programming. We especially recommend [10] (on the language) and [111] (on concurrent programming).

A simple program

We would like to calculate the factorial function. In Java, functions are defined as methods that return a result:

```
class Factorial {
    public long fact(long n) {
        long f=1;
        for (int i=1; i<=n; i++) f=f*i;
        return f;
    }
}
```

Statements are terminated with a semicolon “;” unless they are compound statements, which are delimited by braces { ...}. Variable identifiers are declared by preceding them with their type, as in `long f`. Assignment is denoted by the equals sign `=`. In the object system of Chapter 7 this becomes:

```
class Factorial
  meth fact(N ?X)
    F={NewCell 1} in
      for I in 1..N do F:=@F*I end
      X=@F
    end
end
```

Note that `i` is an assignable variable (a cell) that is updated on each iteration, whereas `I` is a value that is declared anew on each iteration. Factorial can also be defined recursively:

```
class Factorial {
    public long fact(long n) {
        if (n==0) return 1;
        else return n*this.fact(n-1);
    }
}
```

In our object system this becomes:

```
class Factorial
  meth fact(N ?F)
    if N==0 then F=1
    else F=N*{self fact(N-1 $)} end
  end
end
```

There are a few differences with the object system of Chapter 7. The Java keyword `this` is the same as `self` in our object system. Java is statically typed. The type of all variables is declared at compile time. Our model is dynamically typed. A variable can be bound to an entity of any type. In Java, the visibility

of `fact` is declared to be public. In our model, `fact` is public by default; to get another visibility we would have to declare it as a name.

Input/output

Any realistic Java program has to do I/O. Java has an elaborate I/O subsystem based on the notion of *stream*, which is an ordered sequence of data that has a source (for an input stream) or a destination (for an output stream). Do not confuse this with the concept of stream as used in the rest of this book: a list with unbound tail. The Java stream concept generalizes the Unix concept of standard I/O, i.e., the standard input (`stdin`) and standard output (`stdout`) files.

Streams can encode many types, including primitive types, objects, and object graphs. (An object graph is an object together with the other objects it references, directly or indirectly.) Streams can be byte streams or character streams. Characters are not the same as bytes since Java supports Unicode. A *byte* in Java is an 8-bit unsigned integer. A *character* in Java is a Unicode 2.0 character, which has a 16-bit code. We do not treat I/O further in this section.

Defining classes

The `Factorial` class is rather atypical. It has only one method and no attributes. Let us define a more realistic class. Here is a class to define points in two-dimensional space:

```
class Point {  
    public double x, y;  
}
```

The attributes `x` and `y` are public, which means they are visible from outside the class. Public attributes are usually not a good idea; it is almost always better to make them private and use accessor methods:

```
class Point {  
    double x, y;  
    Point(double x1, y1) { x=x1; y=y1; }  
    public double getX() { return x; }  
    public double getY() { return y; }  
}
```

The method `Point` is called a *constructor*; it is used to initialize new objects created with `new`, as in:

```
Point p=new Point(10.0, 20.0);
```

which creates the new `Point` object `p`. Let us add some methods to calculate with points:

```

class Point {
    double x, y;
    Point(double x1, y1) { x=x1; y=y1; }
    public double getX() { return x; }
    public double getY() { return y; }
    public void origin() { x=0.0; y=0.0; }
    public void add(Point p) { x+=p.getX(); y+=p.getY(); }
    public void scale(double s) { x*=s; y*=s; }
}

```

The `p` argument of `add` is a local variable whose initial value is a reference to the argument. In our object system we can define `Point` as follows:

```

class Point
    attr x y
    meth init(X Y) x:=X y:=Y end
    meth getX(X) X=@x end
    meth getY(Y) Y=@y end
    meth origin x:=0.0 y:=0.0 end
    meth add(P) x:=@x+{P getX($)} y:=@y+{P getY($)} end
    meth scale(S) x:=@x*S y:=@y*S end
end

```

This definition is very similar to the Java definition. There are also some minor syntactic differences, such as the operators `+=` and `*=`. Both definitions have private attributes. There is a subtle difference in the visibility of the attributes. In Java, private attributes are visible to all objects of the same class. This means the method `add` could be written differently:

```

public void add(Point p) { x+=p.x; y+=p.y; }

```

This is explained further in Section 7.3.3.

Parameter passing and main program

Parameter passing to methods is done with call by value. A copy of the value is passed to the method and can be modified inside the method without changing the original value. For primitive values, such as integers and floats, this is straightforward. Java also passes object references (not the objects themselves) by value. So objects can almost be considered as using call by reference. The difference is that, inside the method, the field can be modified to refer to another object.

Figure 7.29 gives an example. This example is a complete standalone program; it can be compiled and executed as is. Each Java program has one method, `main`, that is called when the program is started. The object reference `c` is passed by value to the method `sqr`. Inside `sqr`, the assignment `a=null` has no effect on `c`.

```
class MyInteger {
    public int val;
    MyInteger(int x) { val=x; }
}

class CallExample {
    public void sqr(MyInteger a) {
        a.val=a.val*a.val;
        a=null;
    }

    public static void main(String[] args) {
        int c=new MyInteger(25);
        CallExample.sqr(c);
        System.out.println(c.val);
    }
}
```

Figure 7.29: Parameter passing in Java

The argument of `main` is an array of strings that contains the command line arguments of the program when called from the operating system, The method call `System.out.println` prints its argument to the standard output.

Inheritance

We can use inheritance to extend the `Point` class. For example, it can be extended to represent a *pixel*, which is the smallest independently displayable area on a two-dimensional graphics output device such as a computer screen. Pixels have coordinates, just like points, but they also have color.

```
class Pixel extends Point {
    Color color;
    public void origin() {
        super.origin();
        color=null;
    }
    public Color getC() { return color; }
    public void setC(Color c) { color=c; }
}
```

The `extend` keyword is used to denote inheritance; it corresponds to **from** in our object system. We assume the class `Color` is defined elsewhere. The class `Pixel` overrides the `origin` method. The new `origin` initializes both the point and the

```

class BallGame
  attr other count:0
  meth init(Other)
    other:=Other
  end
  meth ball
    count:=@count+1
    {@other ball}
  end
  meth get(X)
    X=@count
  end
end

B1={NewActive BallGame init(B2)}
B2={NewActive BallGame init(B1)}

{B1 ball}

```

Figure 7.30: Two active objects playing ball (definition)

color. It uses **super** to access the overridden method in the immediate ancestor class. With respect to the current class, this class is often called the *superclass*. In our object system, we can define `Pixel` as follows:

```

class Pixel from Point
  attr color
  meth origin
    Point,origin
    color:=null
  end
  meth getC(C) C=@color end
  meth setC(C) color:=C end
end

```

7.8 Active objects

An active object is a port object whose behavior is defined by a class. It consists of a port, a thread that reads messages from the port's stream, and an object that is a class instance. Each message that is received will cause one of the object's methods to be invoked.

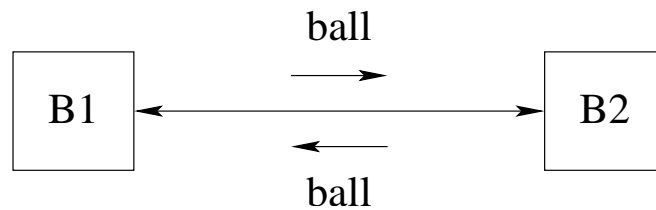


Figure 7.31: Two active objects playing ball (illustration)

7.8.1 An example

Let us start with an example. Consider two active objects, where each object has a reference to the other. When each object receives the message `ball`, it sends the message `ball` to the other. The ball will be passed back and forth indefinitely between the objects. We define the behavior of the active objects by means of a class. Figure 7.30 defines the objects and Figure 7.31 illustrates how the messages pass between them. Each object references the other in the attribute `other`. We also add an attribute `count` to count the number of times the message `ball` is received. The initial call `{B1 ball}` starts the game. With the method `get(X)` we can follow the game's progress:

```

declare X in
  {B1 get(X)}
  {Browse X}

```

Doing this several times will show a sequence of numbers that increase rapidly.

7.8.2 The `NewActive` abstraction

The behavior of active objects is defined with a class. Each method of the class corresponds to a message that is accepted by the active object. Figure 7.30 gives an example. Sending a message `M` to an active object `A` is written as `{A M}`, with the same syntax as invoking a standard, passive object. In contrast to the other objects of this chapter, which are called *passive* objects, the invocation of an active object is asynchronous: it returns immediately, without waiting until the message has been handled. We can define a function `NewActive` that works exactly like `New` except that it creates an active object:

```

fun {NewActive Class Init}
  Obj={New Class Init}
  P
in
  thread S in
    {NewPort S P}
    for M in S do {Obj M} end
  end
  proc {$ M} {Send P M} end

```

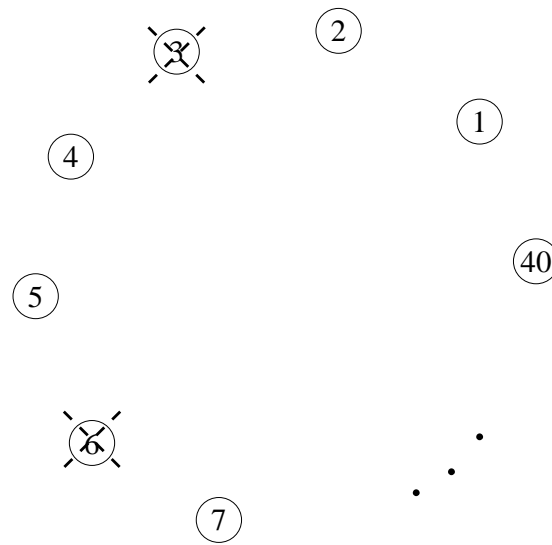


Figure 7.32: The Flavius Josephus problem

end

This makes defining active objects very intuitive.

7.8.3 The Flavius Josephus problem

Let us now tackle a bigger problem. We introduce it with a well-known historical anecdote. Flavius Josephus was a Roman historian of Jewish origin. During the Jewish-Roman wars of the first century AD, he was in a cave with fellow soldiers, 40 men in all, surrounded by enemy Roman troops. They decided to commit suicide by standing in a ring and counting off each third man. Each man so designated was to commit suicide. Figure 7.32 illustrates the problem. Josephus, not wanting to die, managed to place himself in the position of the last survivor.

In the general version of this problem, there are n soldiers numbered from 1 to n and each k -th soldier will be eliminated. The count starts from the first soldier. What is the number of the last survivor? Let us model this problem by representing soldiers with active objects. There is ring of active objects where each object knows its two neighbors. Here is one possible message-passing protocol to solve the problem. A message `kill(x s)` circulates around the ring, where x counts live objects traversed and s is the total number of live objects remaining. Initially, the message `kill(1 n)` is given to the first object. When object i receives the message `kill(x s)` it does the following:

- If it is alive and $s = 1$, then it is the last survivor. It signals this by binding a global variable. No more messages are forwarded.
- If it is alive and $x \bmod k = 0$, then it becomes dead and it sends the message `kill(x+1 s-1)` to the next object in the ring.


```

class Victim
  attr ident step last succ pred alive:true
  meth init(I K L) ident:=I step:=K last:=L end
  meth setSucc(S) succ:=S end
  meth setPred(P) pred:=P end
  meth kill(X S)
    if @alive then
      if S==1 then @last=@ident
      elseif X mod @step==0 then
        alive:=false
        {@pred newsucc(@succ)}
        {@succ newpred(@pred)}
        {@succ kill(X+1 S-1)}
      else
        {@succ kill(X+1 S)}
      end
    else {@succ kill(X S)} end
  end
  meth newsucc(S)
    if @alive then succ:=S
    else {@pred newsucc(S)} end
  end
  meth newpred(P)
    if @alive then pred:=P
    else {@succ newpred(P)} end
  end
end

fun {Josephus N K}
  A={NewArray 1 N null}
  Last
in
  for I in 1..N do
    A.I:={NewActive Victim init(I K Last)}
  end
  for I in 2..N do {A.I setPred(A.(I-1))} end
  {A.1 setPred(A.N)}
  for I in 1..(N-1) do {A.I setSucc(A.(I+1))} end
  {A.N setSucc(A.1)} {A.1 kill(1 N)}
  Last
end

```

Figure 7.33: The Flavius Josephus problem (active object version)

- If it is alive and $x \bmod k \neq 0$, then it sends the message `kill(x+1 S)` to the next object in the ring.
- If it is dead, then it forwards the message `kill(x S)` to the next object.³

Figure 7.33 gives a program that implements this protocol. The function `Josephus` returns immediately with an unbound variable, which will be bound to the number of the last survivor as soon as it is known.

Short-circuit protocol

The solution of Figure 7.33 removes dead objects from the circle with a short-circuit protocol. If this were not done, the traveling message would eventually spend most of its time being forwarded by dead objects. The short-circuit protocol uses the `newsucc` and `newpred` methods. When an object dies, it signals to both its predecessor and its successor that it should be bypassed. The short-circuit protocol is just an optimization to reduce execution time. It can be removed and the program will still run correctly.

Without the short-circuit protocol, the program is actually sequential since there is just a single message circulating. It could have been written as a sequential program. With the short-circuit protocol it is no longer sequential. More than one message can be traveling in the network at any given time.

A declarative solution

As alert programmers, we remark that the solution of Figure 7.33 has no observable nondeterminism. We can therefore write it completely in the declarative concurrent model of Chapter 4. Let us do this and compare the two programs. Figure 7.34 shows a declarative solution that implements the same protocol as the active object version. Like the active object version, it does short-circuiting and eventually terminates with the identity of the last survivor. It pays to compare the two versions carefully. The declarative version is half the size of the active object version. One reason is that streams are first-class entities. This makes short-circuiting very easy: just return the input stream as output.

The declarative program uses a concurrent abstraction, `Pipe`, that it defines especially for this program. If $l \leq h$, then the function call `{Pipe xs L H F}` creates a pipeline of $h-l+1$ stream objects, numbered from l to h inclusive. Each stream object is created by the call `{F is i}`, which is given an input stream `is` and an integer `i` and returns the output stream. We create a closed ring by feeding the output stream `zs` back to the input, with the additional message `kill(1 N)` to start the execution.

³The dead object is a kind of zombie.

```

fun {Pipe Xs L H F}
  if L<H then {Pipe {F Xs L} L+1 H F} else Xs end
end

fun {Josephus2 N K}
  fun {Victim Xs I}
    case Xs of kill(X S)|Xr then
      if S==1 then Last=I nil
      elseif X mod K==0 then
        kill(X+1 S-1)|Xr
      else
        kill(X+1 S)|{Victim Xr I}
      end
    [] nil then nil end
  end
  Last Zs
in
  Zs={Pipe kill(1 N)|Zs 1 N
    fun {$ Is I} thread {Victim Is I} end end}
  Last
end

```

Figure 7.34: The Flavius Josephus problem (data-driven concurrent version)

7.8.4 Other active object abstractions

Section 5.3 shows some of the useful protocols that we can build on top of message passing. Let us take two of these protocols and make them into abstractions for active objects.

Synchronous active objects

It is easy to extend active objects to give them synchronous behavior, like a standard object or an RMI object. A synchronous invocation {Obj M} does not return until the method corresponding to M is completely executed. Internal to the abstraction, we use a dataflow variable to do the synchronization. Here is the definition of NewSync, which creates a synchronous active object:

```

fun {NewSync Class Init}
  Obj={New Class Init}
  P
in
  thread S in
    {NewPort S P}
    for M#X in S do {Obj M} X=unit end
  end
  proc {$ M} X in {Send P M#X} {Wait X} end

```

end

Each message sent to the object contains a synchronization token x , which is bound when the message is completely handled.

Active objects with exception handling

Explicitly doing exception handling for active objects can be cumbersome, since it means adding a **try** in each server method and a **case** after each call. Let us hide these statements inside an abstraction. The abstraction adds an extra argument that can be used to test whether or not an exception occurred. Instead of adding the extra argument in the method, we add it to the object invocation itself. In this way, it automatically works for all methods. The extra argument is bound to `normal` if the invocation completes normally, and to `exception(E)` if the object raises the exception E . Here is the definition of `NewActiveExc`:

```
fun {NewActiveExc Class Init}
  P Obj={New Class Init} in
    thread S in
      {NewPort S P}
      for M#X in S do
        try {Obj M} X=normal
        catch E then X=exception(E) end
      end
    end
    proc {$ M X} {Send P M#X} end
  end
```

The object `Obj` is called as `{Obj M X}`, where x is the extra argument. So the send is still asynchronous and the client can examine at any time whether the call completed successfully or not. For the synchronous case, we can put the **case** statement inside the abstraction:

```
proc {$ M}
  X in
    {Send P M#X}
    case E of normal then skip
    [] exception(Exc) then raise Exc end end
end
```

This lets us call the object exactly like a passive object.

7.8.5 Event manager with active objects

We can use active objects to implement a simple concurrent event manager. The event manager contains a set of event handlers. Each handler is a triple `Id#F#S`, where `Id` uniquely identifies the handler, `F` defines the state update function, and `S` is the handler's state. When an event E occurs, each triple `Id#F#S` is replaced

```

class EventManager
  attr
    handlers
  meth init handlers:=nil end
  meth event(E)
    handlers:=
      {Map @handlers fun {$ Id#F#S} Id#F#{F E S} end}
  end
  meth add(F S ?Id)
    Id={NewName}
    handlers:=Id#F#S|@handlers
  end
  meth delete(DId ?DS)
    handlers:={List.partition
      @handlers fun {$ Id#F#S} DId==Id end [_#_#DS]}
  end
end

```

Figure 7.35: Event manager with active objects

by `Id#F#{F E S}`. That is, each event handler is a finite state machine, which does a transition from state `S` to state `{F E S}` when the event `E` occurs.

The event manager was originally written in Erlang [7]. The Erlang computation model is based on communicating active objects (see Chapter 5). The translation of the original code to the concurrent stateful model was straightforward.

We define the event manager `EM` as an active object with four methods:

- `{EM init}` initializes the event manager.
- `{EM event(E)}` posts the event `E` at the event manager.
- `{EM add(F S Id)}` adds a new handler with update function `F` and initial state `S`. Returns a unique identifier `Id`.
- `{EM delete(Id S)}` removes the handler with identifier `Id`, if it exists. Returns the handler's state in `S`.

Figure 7.35 shows how to define the event manager as a class. We show how to use the event manager to do error logging. First we define a new event manager:

```
EM={NewActive EventManager init}
```

We then install a memory-based handler. It logs every event in an internal list:

```
MemH=fun {$ E Buf} E|Buf end
Id={EM add(MemH nil $)}
```

We can replace the memory-based handler by a disk-based handler during execution, without losing any of the already-logged events. In the following code,

```

class ReplaceEventManager from EventManager
  meth replace(NewF NewS OldId NewId
              insert:P<=proc {$ _} skip end)
    Buf=EventManager,delete(OldId $)
  in
    {P Buf}
    NewId=EventManager,add(NewF NewS $)
  end
end

```

Figure 7.36: Adding functionality with inheritance

we remove the memory-based handler, open a log file, write the already-logged events to the file, and then define and install the disk-based handler:

```

DiskH=fun {$ E F} {F write(vs:E)} F end
File={New Open.file init(name:`event.log` flags:[write create])}
Buf={EM delete(Id $)}
for E in {Reverse Buf} do {File write(vs:E)} end
Id2={EM add(DiskH File $)}

```

This uses the System module Open to write the log. We could use the File module but then the rest of the program could not use it, since it only supports one open file at a time for writing.

Adding functionality with inheritance

The event manager of Figure 7.35 has the defect that if events occur during a replacement, i.e., between the delete and add operations, then they will not be logged. How can we remedy this defect? A simple solution is to add a new method, `replace` to `EventManager` that does both the delete and add. Because all methods are executed sequentially in the active object, this ensures no event will occur between the delete and add. We have the choice to add the new method directly, to `EventManager`, or indirectly, to a subclass by inheritance. Which possibility is the right solution depends on several factors. First, whether we have access to the source code of `EventManager`. If we do not, then inheritance is the only possibility. If we do have the source code, inheritance may still be the right answer. It depends on how often we need the replace functionality. If we almost always need it in event managers, then we should modify `EventManager` directly and not create a second class. If we rarely need it, then its definition should not encumber `EventManager`, and we can separate it by using inheritance.

Let us use inheritance for this example. Figure 7.36 defines a new class `ReplaceEventManager` that inherits from `EventManager` and adds a new method `replace`. Instances of `ReplaceEventManager` have all methods of `EventManager` as well as the method `replace`. The `insert` field is optional; it can be used to insert an operation to be done between the delete and add operations. We define

```

class Batcher
  meth batch(L)
    for X in L do
      if {IsProcedure X} then {X} else {self X} end
    end
  end
end

```

Figure 7.37: Batching a list of messages and procedures

a new event manager:

```
EM={NewActive ReplaceEventManager init}
```

Now we can do the replacement as follows:

```

DiskH=fun {$ E S} {S write(vs:E)} S end
File={New Open.file init(name:`event.log` flags:[write create])}
Id2
{EM replace(DiskH File Id Id2
  insert:
    proc {$ S}
      for E in {Reverse S} do
        {File write(vs:E)} end
      end)}

```

Because `replace` is executed inside the active object, it is serialized with all the other messages to the object. This ensures that no events can arrive between the delete and add methods.

Batching operations using a mixin class

A second way to remedy the defect is to add a new method that does *batching*, i.e., it does a list of operations. Figure 7.37 defines a new class `Batcher` that has just one method, `batch(L)`. The list `L` can contain messages or zero-argument procedures. When `batch(L)` is called, the messages are passed to `self` and the procedures are executed, in the order they occur in `L`. This is an example of using first-class messages. Since messages are also language entities (they are records), they can be put in a list and passed to `Batcher`. We define a new class that inherits from `EventManager` and brings in the functionality of `Batcher`:

```
class BatchingEventManager from EventManager Batcher end
```

We use multiple inheritance because `Batcher` can be useful to *any* class that needs batching, not just to event managers. Now we can define a new event manager:

```
EM={NewActive BatchingEventManager init}
```

All instances of `BatchingEventManager` have all methods of `EventManager` as well as the method `batch`. The class `Batcher` is an example of a *mixin* class: it

adds functionality to an existing class without needing to know anything about the class. Now we can replace the memory-based handler by a disk-based handler:

```
DiskH=fun {$ E S} {S write(vs:E)} S end
File={New Open.file init(name:`event.log` flags:[write create])}
Buf Id2
{EM batch([delete(Id Buf)
    proc {$}
        for E in {Reverse Buf} do {File write(vs:E)} end
    end
    add(DiskH File Id2)])}
```

The `batch` method guarantees atomicity in the same way as the `replace` method, i.e., because it executes inside the active object.

What are the differences between the replacement solution and the batching solution? There are two:

- The replacement solution is more efficient because the `replace` method is hard-coded. The `batch` method, on the other hand, adds a layer of interpretation.
- The batching solution is more flexible. Batching can be added to *any* class using multiple inheritance. No new methods have to be defined. Furthermore, *any* list of messages and procedures can be batched, even a list that is calculated at run time. However, the batching solution requires that the language support first-class messages.

Combining computation models

The event manager is an interesting combination of the declarative, object-oriented, and stateful concurrent computation models:

- Each event handler is defined declaratively by its state update function. Even stronger, each method of the event manager can be seen as a declarative definition. Each method takes the event manager's internal state from the attribute `handlers`, does a declarative operation on it, and stores the result in `handlers`.
- All methods are executed in sequential fashion, as if they were in a stateful model with no concurrency. All concurrency is managed by the active object abstraction, as implemented by `NewActive`. This abstraction guarantees that all object invocations are serialized. Especially, no locking or other concurrency control is needed.
- New functionality, for example replacement or batching, is added by using object-oriented inheritance. Because the new method executes inside the active object, it is guaranteed to be atomic.

The result is that event handlers are defined sequentially and declaratively, and yet can be used in a stateful concurrent environment. This is an example of impedance matching, as defined in Section 4.7.7. Impedance matching is a special case of the general principle of *separation of concerns*. The concerns of state and concurrency are separated from the definition of the event handlers. It is good programming practice to separate concerns as much as possible. Using different computation models together often helps to achieve separation of concerns.

7.9 Exercises

1. **Uninitialized objects.** The function `New` creates a new object when given a class and an initial message. Write another function `New2` that does not require an initial message. That is, the call `Obj={New2 Class}` creates a new object without initializing it. Hint: write `New2` in terms of `New`.
2. **Protected methods in the Java sense.** A *protected method* in Java has two parts: it is accessible throughout the package that defines the class and also by descendants of the class. For this exercise, define a linguistic abstraction that allows to annotate a method or attribute as `protected` in the Java sense. Show how to encode this in the model of Section 7.3.3 by using name values. Use functors to represent Java packages. For example, one approach might be to define the name value globally in the functor and also to store it in an attribute called `setOfAllProtectedAttributes`. Since the attribute is inherited, the method name is visible to all subclasses. Work out the details of this approach.
3. **Method wrapping.** Section 7.3.5 shows how to do method wrapping. The definition of `TraceNew2` given there uses a class `Trace` that has an external reference. This is not allowed in some object systems. For this exercise, rewrite `TraceNew2` so that it uses a class with no external references.
4. **Implementing inheritance and static binding.** For this exercise, generalize the implementation of the object system given in Section 7.6 to handle static binding and to handle inheritance with any number of super-classes (not just two).
5. **Message protocols with active objects.** For this exercise, redo the message protocols of Section 5.3 with active objects instead of port objects.
6. **The Flavius Josephus problem.** Section 7.8.3 solves this problem in two ways, using active objects and using data-driven concurrency. For this exercise, do the following:
 - (a) Use a third model, the sequential stateful model, to solve the problem. Write two programs: the first without short-circuiting and the second

with it. Try to make both as concise and natural as possible in the model. For example, without short-circuiting an array of booleans is a natural data structure to represent the ring. Compare the structure of both programs with the two programs in Section 7.8.3.

- (b) Compare the execution times of the different versions. There are two orthogonal analyses to be done. First, measure the advantages (if any) of using short-circuiting for various values of n and k . This can be done in each of the three computation models. For each model, divide the (n, k) plane into two regions, depending on whether short-circuiting is faster or not. Are these regions the same for each model? Second, compare the three versions with short-circuiting. Do these versions have the same asymptotic time complexity as a function of n and k ?
- 7. (*advanced exercise*) **Inheritance without explicit state.** Inheritance does not require explicit state; the two concepts are orthogonal. For this exercise, design and implement an object system with classes and inheritance but without explicit state. One possible starting point is the implementation of declarative objects in Section 6.4.2.
- 8. (*research project*) **Programming design patterns.** For this exercise, design an object-oriented language that allows “upwards” inheritance (defining a new superclass of a given class) as well as higher-order programming. Upwards inheritance is usually called *generalization*. Implement and evaluate the usefulness of your language. Show how to program the design patterns of Gamma *et al* [58] as abstractions in your language. Do you need other new operations in addition to generalization?

Chapter 8

Shared-State Concurrency

The shared-state concurrent model is a simple extension to the declarative concurrent model that adds explicit state in the form of cells, which are a kind of mutable variable. This model is equivalent in expressiveness to the message-passing concurrent model of Chapter 5, because cells can be efficiently implemented with ports and vice versa. In practice, however, the shared-state model is harder to program than the message-passing model. Let us see what the problem is and how we can solve it.

The inherent difficulty of the model

Let us first see exactly why the shared-state model is so difficult. Execution consists of multiple threads, all executing independently and all accessing shared cells. At some level, a thread's execution can be seen as a sequence of atomic instructions. For a cell, these are @ (access), := (assignment), and `Exchange`. Because of the interleaving semantics, all execution happens as if there was one global order of operations. All operations of all threads are therefore “interleaved” to make this order. There are many possible interleavings; their number is limited only by data dependencies (calculations needing results of others). Any particular execution realizes an interleaving. Because thread scheduling is nondeterministic, there is no way to know which interleaving will be chosen.

But just how many interleavings are possible? Let us consider a simple case: two threads, each doing k cell operations. Thread T_1 does the operations a_1, a_2, \dots, a_k and thread T_2 does b_1, b_2, \dots, b_k . How many possible executions are there, interleaving all these operations? It is easy to see that the number is $\binom{2k}{k}$.

Any interleaved execution consists of $2k$ operations, of which each thread takes k . Consider these operations as integers from 1 to $2k$, put in a set. Then T_1 takes k integers from this set and T_2 gets the others. This number is exponential in k .¹ For three or more threads, the number of interleavings is even bigger (see Exercises).

¹Using Stirling's formula we approximate it as $2^{2k}/\sqrt{\pi k}$.

It is possible to write algorithms in this model and prove their correctness by reasoning on all possible interleavings. For example, given that the only atomic operations on cells are @ and :=, then Dekker's algorithm implements mutual exclusion. Even though Dekker's algorithm is short (e.g., 48 lines of code in [43], using a Pascal-like language), the reasoning is already quite difficult. For bigger programs, this technique rapidly becomes impractical. It is unwieldy and interleavings are easy to overlook.

Why not use declarative concurrency?

Given the inherent difficulty of programming in the shared-state concurrent model, an obvious question is why not stick with the declarative concurrent model of Chapter 4? It is enormously simpler to program in than the shared-state concurrent model. It is almost as easy to reason in as the declarative model, which is sequential.

Let us briefly examine why the declarative concurrent model is so easy. It is because dataflow variables are monotonic: they can be bound to just one value. Once bound, the value does not change. Threads that share a dataflow variable, e.g., a stream, can therefore calculate with the stream as if it were a simple value. This is in contrast to cells, which are nonmonotonic: they can be assigned any number of times to values that have no relation to each other. Threads that share a cell cannot make any assumptions about its content: at any time, the content can be completely different from any previous content.

The problem with the declarative concurrent model is that threads must communicate in a kind of “lock-step” or “systolic” fashion. Two threads communicating with a third thread cannot execute independently; they must coordinate with each other. This is a consequence of the fact that the model is still declarative, and hence deterministic.

We would like to allow two threads to be completely independent and yet communicate with the same third thread. For example, we would like clients to make independent queries to a common server or to independently increment a shared state. To express this, we have to leave the realm of declarative models. This is because two independent entities communicating with a third introduce an observable nondeterminism. A simple way to solve the problem is to add explicit state to the model. Ports and cells are two important ways to add explicit state. This gets us back to the model with both concurrency and state. But reasoning directly in this model is impractical. Let us see how we can get around the problem.

Getting around the difficulty

Programming in the stateful concurrent model is largely a matter of managing the interleavings. There are two successful approaches:

- *Message passing between port objects.* This is the subject of Chapter 5. In this approach, programs consist of port objects that send asynchronous messages to each other. Internally, a port object executes in a single thread.
- *Atomic actions on shared cells.* This is the subject of the present chapter. In this approach, programs consist of passive objects that are invoked by threads. Abstractions are used to build large atomic actions (e.g., using locking, monitors, or transactions) so that the number of possible interleavings is small.

Each approach has its advantages and disadvantages. The technique of invariants, as explained in Chapter 6, can be used in both approaches to reason about programs. The two approaches are equivalent in a theoretical sense, but not in a practical sense: a program using one approach can be rewritten to use the other approach, but it may not be as easy to understand [109].

Structure of the chapter

The chapter consists of seven main sections:

- Section 8.1 defines the shared-state concurrent model.
- Section 8.2 brings together and compares briefly all the different concurrent models that we have introduced in the book. This gives a balanced perspective on how to do practical concurrent programming.
- Section 8.3 introduces the concept of lock, which is the basic concept used to create coarse-grained atomic actions. A lock defines an area of the program inside of which only a single thread can execute at a time.
- Section 8.4 extends the concept of lock to get the concept of monitor, which gives better control on which threads are allowed to enter and exit the lock. Monitors make it possible to program more sophisticated concurrent programs.
- Section 8.5 extends the concept of lock to get the concept of transaction, which allows a lock to be either committed or aborted. In the latter case, it is as if the lock had never executed. Transactions allow to program concurrent programs that can handle rare events and non-local exits.
- Section 8.6 summarizes how concurrency is done in Java, a popular concurrent object-oriented language.

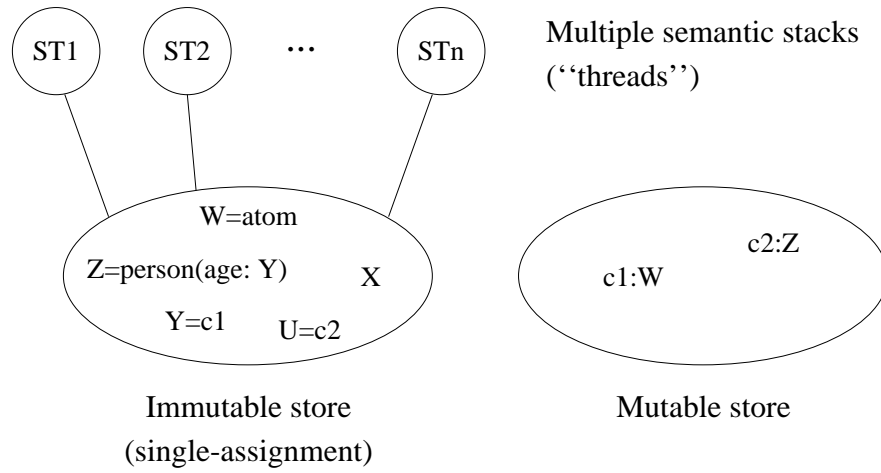


Figure 8.1: The shared-state concurrent model

$\langle s \rangle ::=$	
skip	Empty statement
$\langle s \rangle_1 \langle s \rangle_2$	Statement sequence
local $\langle x \rangle$ in $\langle s \rangle$ end	Variable creation
$\langle x \rangle_1 = \langle x \rangle_2$	Variable-variable binding
$\langle x \rangle = \langle v \rangle$	Value creation
if $\langle x \rangle$ then $\langle s \rangle_1$ else $\langle s \rangle_2$ end	Conditional
case $\langle x \rangle$ of $\langle \text{pattern} \rangle$ then $\langle s \rangle_1$ else $\langle s \rangle_2$ end	Pattern matching
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Procedure application
thread $\langle s \rangle$ end	Thread creation
$\{ \text{ByNeed } \langle x \rangle \langle y \rangle \}$	Trigger creation
$\{ \text{NewName } \langle x \rangle \}$	Name creation
$\langle y \rangle = !! \langle x \rangle$	Read-only view
try $\langle s \rangle_1$ catch $\langle x \rangle$ then $\langle s \rangle_2$ end	Exception context
raise $\langle x \rangle$ end	Raise exception
$\{ \text{FailedValue } \langle x \rangle \langle y \rangle \}$	Failed value
$\{ \text{IsDet } \langle x \rangle \langle y \rangle \}$	Boundness test
$\{ \text{NewCell } \langle x \rangle \langle y \rangle \}$	Cell creation
$\{ \text{Exchange } \langle x \rangle \langle y \rangle \langle z \rangle \}$	Cell exchange

Table 8.1: The kernel language with shared-state concurrency

8.1 The shared-state concurrent model

Chapter 6 adds explicit state to the declarative model. This allows to do object-oriented programming. Chapter 4 adds concurrency to the declarative model. This allows to have multiple active entities that evolve independently. The next step is to add *both* explicit state and concurrency to the declarative model. One way to do this is given in Chapter 5: by adding ports. This chapter gives an alternative way: by adding cells.

The resulting model, called the *shared-state concurrent model*, is shown in Figure 8.1. Its kernel language is defined in Table 8.1. If we consider the subset of operations up to `ByNeed` then we have the declarative concurrent model. We add names, read-only variables, exceptions, and explicit state to this model.

8.2 Programming with concurrency

By now, we have seen many different ways to write concurrent programs. Before diving into programming with shared-state concurrency, let us make a slight detour and put all these ways into perspective. We first give a brief overview of the main approaches. We then examine more closely the new approaches that become possible with shared-state concurrency.

8.2.1 Overview of the different approaches

For the programmer, there are four main practical approaches to writing concurrent programs:

- **Sequential programming** (Chapters 3, 6, and 7). This is the baseline approach that has no concurrency. It can be either eager or lazy.
- **Declarative concurrency** (Chapter 4). This is concurrency in the declarative model, which gives the same results as a sequential program but can give them incrementally. This model is usable when there is no observable nondeterminism. It can be either eager (data-driven concurrency) or lazy (demand-driven concurrency).
- **Message-passing concurrency** (Chapter 5 and Section 7.8). This is message passing between port objects, which are internally sequential. This limits the number of interleavings. Active objects (Section 7.8) are a variant of port objects where the object's behavior is defined by a class.
- **Shared-state concurrency** (this chapter). This is threads updating shared passive objects using coarse-grained atomic actions. This is another approach to limit the number of interleavings.

Model	Approaches
<i>Sequential (declarative or stateful)</i>	Sequential programming Order-determining concurrency Coroutining Lazy evaluation
<i>Declarative concurrent</i>	Data-driven concurrency Demand-driven concurrency
<i>Stateful concurrent</i>	Use the model directly Message-passing concurrency Shared-state concurrency
<i>Nondeterministic concurrent</i>	Stream objects with merge

Figure 8.2: Different approaches to concurrent programming

Figure 8.2 gives a complete list of these approaches and some others. Previous chapters have already explained sequential programming and concurrent declarative programming. In this chapter we look at the others. We first give an overview of the four main approaches.

Sequential programming

In a sequential model, there is a total order among all operations. This is the strongest order invariant a program can have. We have seen two ways that this order can be relaxed a little, while still keeping a sequential model:

- **“Order-determining” concurrency** (Section 4.4.1). In this model, all operations execute in a total order, like with sequential execution, but the order is unknown to the programmer. Concurrent execution with dataflow finds the order dynamically.
- **Coroutining** (Section 4.4.2). In this model, preemption is explicit, i.e., the program decides when to pass control to another thread. Lazy evaluation, in which laziness is added to a sequential program, does coroutining.

Both of these variant models are still deterministic.

Declarative concurrency

The declarative concurrent models of Chapter 4 all add threads to the declarative model. This does not change the result of a calculation, but only changes the *order* in which the result is obtained. For example, the result might be given incrementally. This allows to build a dynamic network of concurrent stream

objects connected with streams. Because of concurrency, adding an element to its input stream allows a stream object to produce an output immediately.

These models have nondeterminism in the implementation, since the system chooses how to advance the threads. But, to stay declarative, the nondeterminism must not be observable to the program. The declarative concurrent models guarantee this as long as no exceptions are raised (since exceptions are witnesses to an observable nondeterminism). In practice, this means that each stream object must know at all times from which stream its next input will come.

The demand-driven concurrent model, also known as lazy execution (Section 4.5), is a form of declarative concurrency. It does not change the result of a calculation, but only affects *how much* calculation is done to obtain the result. It can sometimes give results in cases where the data-driven model would go into an infinite loop. This is important for resource management, i.e., controlling how many computational resources are needed. Calculations are initiated only when their results are needed by other calculations. Lazy execution is implemented with by-need triggers.

Message-passing concurrency

Message passing is a basic programming style of the stateful concurrent model. It is explained in Chapter 5 and Section 7.8. It extends the declarative concurrent model with a simple kind of communication channel, a *port*. It defines *port objects*, which extend stream objects to read from ports. A program is then a network of port objects communicating with each other through asynchronous message passing. Each port object decides when to handle each messages. The port object processes the messages sequentially. This limits the possible interleavings and allows us to reason using invariants. Sending and receiving messages between port objects introduces a causality between events (send, receive, and internal). Reasoning on such systems requires reasoning on the causality chains.

Shared-state concurrency

Shared state is another basic programming style in the stateful concurrent model. It is explained in the present chapter. It consists of a set of threads accessing a set of shared passive objects. The threads coordinate among each other when accessing the shared objects. They do this by means of coarse-grained atomic actions, e.g., locks, monitors, or transactions. Again, this limits the possible interleavings and allows us to reason using invariants.

Relationship between ports and cells

The message-passing and shared-state models are equivalent in expressiveness. This follows because ports can be implemented with cells and vice versa. (It is an amusing exercise to implement the `Send` operation using `Exchange` and vice versa.) It would seem then that we have the choice whether to add ports or cells

to the declarative concurrent model. However, in practice this is not so. The two computation models emphasize a quite different programming style that is appropriate for different classes of applications. The message-passing style is of programs as active entities that coordinate with one another. The shared-state style is of programs as passive data repositories that are modified in a coherent way.

Other approaches

In addition to these four approaches, there are two others worth mentioning:

- **Using the stateful concurrent model directly.** This consists in programming directly in the stateful concurrent model, either in message-passing style (using threads, ports, and dataflow variables, see Section 5.5), in shared-state style (using threads, cells, and dataflow variables, see Section 8.2.2), or in a mixed style (using both cells and ports).
- **Nondeterministic concurrent model** (Section 5.7.1). This model adds a nondeterministic choice operator to the declarative concurrent model. It is a stepping stone to the stateful concurrent model.

They are less common, but can be useful in some circumstances.

Which concurrent model to use?

How do we decide which approach to use when writing a concurrent program? Here are a few rules of thumb:

- Stick with the least concurrent model that suffices for your program. For example, if using concurrency does not simplify the architecture of the program, then stick with a sequential model. If your program does not have any observable nondeterminism, such as independent clients interacting with a server, then stick with the declarative concurrent model.
- If you absolutely need both state and concurrency, then use either the message-passing or the shared-state approach. The message-passing approach is often the best for multi-agent programs, i.e., programs that consist of autonomous entities (“agents”) that communicate with each other. The shared-state approach is often the best for data-centered programs, i.e., programs that consist of a large repository of data (“database”) that is accessed and updated concurrently. Both approaches can be used together for different parts of the same application.
- Modularize your program and concentrate the concurrency aspects in as few places as possible. Most of the time, large parts of the program can be sequential or use declarative concurrency. One way to implement this

is with impedance matching, which is explained in Section 4.7.7. For example, active objects can be used as front ends to passive objects. If the passive objects are all called from the same active object then they can use a sequential model.

Too much concurrency is bad

There is a model, the *maximally concurrent model*, that has even more concurrency than the stateful concurrent model. In the maximally concurrent model, each operation executes in its own thread. Execution order is constrained only by data dependencies. This has the greatest possible concurrency.

The maximally concurrent model has been used as the basis for experimental parallel programming languages. But it is both hard to program in and hard to implement efficiently (see Exercise). This is because operations tend to be fine-grained compared to the overhead of scheduling and synchronizing. The shared-state concurrent model of this chapter does not have this problem because thread creation is explicit. This allows the programmer to control the granularity. We do not present the maximally concurrent model in more detail in this chapter. A variant of this model is used for constraint programming (see Chapter 12).

8.2.2 Using the shared-state model directly

As we saw in the beginning of this chapter, programming directly in the shared-state model can be tough. This is because there are potentially an enormous number of interleavings, and the program has to work correctly for all of them. That is the main reason why more high-level approaches, like active objects and atomic actions, were developed. Yet, it is sometimes useful to use the model directly. Before moving on to using atomic actions, let us see what can be done directly in the shared-state concurrent model. Practically, it boils down to programming with threads, procedures, cells, and dataflow variables. This section gives some examples.

Concurrent stack

A *concurrent ADT* is an ADT where multiple threads can execute the ADT operations simultaneously. The first and simplest concurrent ADT we show is a stack. The stack provides nonblocking push and pop operations, i.e., they never wait, but succeed or fail immediately. Using exchange, its implementation is very compact, as Figure 8.3 shows. The exchange does two things: it accesses the cell's old content and it assigns a new content. Because exchange is atomic, it can be used in a concurrent setting. Because the push and pop operations each do just one exchange, they can be interleaved in any way and still work correctly. Any number of threads can access the stack concurrently, and it will work correctly. The only restriction is that a pop should not be attempted on an empty stack. An exception can be raised in that case, e.g., as follows:

```

fun {NewStack}
  Stack={NewCell nil}
  proc {Push X}
  S in
    {Exchange Stack S X|S}
  end
  fun {Pop}
  X S in
    {Exchange Stack X|S S}
    X
  end
in
  stack(push:Push pop:Pop)
end

```

Figure 8.3: Concurrent stack

```

fun {Pop}
X S in
  try {Exchange Stack X|S S}
  catch failure(...) then raise stackEmpty end end
  X
end

```

The concurrent stack is simple because each operation does just a single exchange. Things become much more complex when an ADT operation does more than one cell operation. For the ADT operation to be correct in general, these operations would have to be done atomically. To guarantee this in a simple way, we recommend using the active object or atomic action approach.

Simulating a slow network

The object invocation {Obj M} calls Obj immediately and returns when the call is finished. We would like to modify this to simulate a slow, asynchronous network, where the object is called asynchronously after a delay that represents the network delay. Here is a simple solution that works for any object:

```

fun {SlowNet1 Obj D}
  proc {$ M}
  thread
    {Delay D}
    {Obj M}
  end
end
end

```

The call {SlowNet1 Obj D} returns a “slow” version of Obj. When the slow object is invoked, it waits at least D milliseconds before calling the original object.

Preserving message order with token passing The above solution does not preserve message order. That is, if the slow object is invoked several times from within the same thread, then there is no guarantee that the messages will arrive in the same order as they are sent. Moreover, if the object is invoked from several threads, different executions of the object can overlap in time which could result in an inconsistent object state. Here is a solution that does preserve message order and guarantees that only one thread at a time can execute inside the object:

```

fun {SlowNet2 Obj D}
  C={NewCell unit}
in
  proc {$ M}
    Old New in
      {Exchange C Old New}
      thread
        {Delay D}
        {Wait Old}
        {Obj M}
        New=unit
      end
    end
  end

```

This solution uses a general technique, called *token passing*, to extract an execution order from one part of a program and impose it on another part. The token passing is implemented by creating a sequence of dataflow variables x_0, x_1, x_2, \dots , and passing consecutive pairs $(x_0, x_1), (x_1, x_2), \dots$ to the operations that should be done in the same order. An operation that receives the pair (x_i, x_{i+1}) does the following steps in order:

1. Wait until the token arrives, i.e., until x_i is bound (`{wait x_i }`).
2. Do the computation.
3. Send the token to the next pair, i.e., bind x_{i+1} (`x_{i+1} =unit`).

In the definition of `SlowNet2`, each time the slow object is called, a pair of variables (`Old, New`) is created. This is inserted into the sequence by the call `{Exchange C Old New}`. Because `Exchange` is atomic, this works also in a concurrent setting where many threads call the slow object. Each pair shares one variable with the previous pair (`Old`) and one with the next pair (`New`). This effectively puts the object call in an ordered queue. Each call is done in a new thread. It first waits until the previous call has terminated (`{wait Old}`), then invokes the object (`{Obj M}`), and finally signals that the next call can continue (`New=unit`). The `{Delay D}` call must be done before `{wait Old}`; otherwise each object call would take at least `D` milliseconds, which is incorrect.