# An Introduction To Programming With X10

DRAFT

Bard Bloom, bardb@us.ibm.com
Jonathan Brezin, brezin@us.ibm.com
with
Stephen J. Fink, sjfink@us.ibm.com
Cal Swart, cals@us.ibm.com

Please send comments to brezin@us.ibm.com.

June 30, 2011

This is a draft of the Part I of the Guide. This draft focusses on core features of the X10 language and programming model. This Guide is a work-in-progress. We anticipate that a complete draft of Part I and a substantial part of Part II will be available in early first quarter 2011.

# Preface

## Who This Book Is For

X10 is an experimental language designed with modern programming language concepts and features to support "scale out" and "scale up" concurrency: larger and larger numbers of more and more powerful nodes that have to be used together effectively. This text introduces X10 to programmers who have some experience with an object-oriented language. To this end, we'll talk about Java and C++ from time to time, as a way to explain concepts that these languages share with X10. So, a background in Java or C++ will help in reading this book; failing that, familiarity with object-oriented "scripting" languages like JavaScript, Python, Ruby, or Smalltalk will also give much of the required background.

Those familiar with C, but not C++, may find this book useful, but some minimal knowledge of object-oriented programming really is a must—*e.g.* classes, objects, class methods versus instance methods, and so on, but nothing deep.

## The Scope Of The Book

Just as an MPI programmer can go a long way with only a half-dozen or so of the more than one hundred MPI functions, the new X10 programmer can get a lot of mileage from a carefully chosen subset of X10 constructs. Aiming for a gentle introduction, we stick to the most fundamental language constructs, that cover the most common programming tasks. The final chapter surveys what we don't cover here in detail.

Here is a quick outline of what we are going to do.

> The first chapter samples a few simple X10 programs so you can get a feeling for the language. It introduces concurrency, both local and distributed. The chapter emphasizes the basic constructs, not performance, except to the extent that it comes up as a reason for choosing one construct over another.
>
> The next several chapters flesh out the critical parts of the language: types, expressions, control flow, and so on.

## Trademarks

The following is a list of the trademarked names that appear in the text:

**Mac OS** registered trademark of Apple Computer Inc.

**Linux** registered trademark of Linus Torvalds.

**Unix** registered trademark licensed through X/Open Company, Ltd

**Windows** a shorthand for one of a number of operating systems, Windows XP, Windows Vista, and Windows 7, that are registered trademarks of Microsoft Corporation.

## Getting X10 Onto Your Machine

We've put together some installation guides for various platforms that show in detail how to set up your workstation to develop X10 programs. The platforms supported are

**Mac:** http://dist.codehaus.org/x10/documentation/install/mac-cl-install.html

**Linux:** http://dist.codehaus.org/x10/documentation/install/linux-cl-install.html

**Windows:** http://dist.codehaus.org/x10/documentation/install/windows-cl-install.html

Once you have the environment set up, you will be able to run the programs in this guide.

For Windows users, we have also developed a batch installation of the Cygwin environment that X10 requires. This will make it easier to get started than getting what you need piecemeal off the Web. The installer can be downloaded from

http://dist.codehaus.org/x10/documentation/install/cygwin-install.zip

You can refer to the Windows installation guide above for more details about its use.

If you are an Eclipse user, the X10 Development Toolkit (X10DT) is a plugin that can be installed into Eclipse 3.5.x (Galileo) or Eclipse 3.6.x (Helios). If you are unfamiliar with Eclipse, it is a free Open Source IDE, and we recommend you take a look at its introductory Web page, http://www.eclipse.org/home/newcomers.php. If you have Eclipse installed, you can go to the `Help` menu and select "`Install New Software`". You will be asked to add an update site. The url to use for X10DT is:

http://dist.codehaus.org/x10/x10dt/2.1/updateSite/.

If you want to install both Eclipse and X10DT together from scratch, there is an all-in-one installation that contains everything you need to start building X10 programs. It can be downloaded from the X10 web site:

http://x10.codehaus.org/X10DT+2.1+Installation.

The X10 web site, http://x10.codehaus.org/, is a source for the latest news about X10. The X10 mailing lists are all hosted at http://x10.codehaus.org/For+Users. You can subscribe to any of them, but the ones most likely to be of interest are

- x10-announce: for (infrequent) announcements of X10 releases, and
- x10-users: for a general X10 user list.

## Acknowledgements

Particular thanks go to David Grove and Igor Peshansky, who have gone out of their way to answer our questions and to lay out the rationale behind X10's design. Rachel Bellamy, Vijay Saraswat, Janice Shepherd, Mandana Vaziri, and Yoav Zibin were kind enough to read a early versions carefully, and the guide is much the better for their comments.

# Contents

# Part I

# Basic X10

# 1 A Whirlwind Tour of X10

In this chapter, we'll look at a couple of quick examples that illustrate X10 in action. The next two chapters will then fill in a lot of the details.

## 1.1 Hello!

Enough suspense! You knew it was coming, and here it is! [1]

```
1  // Copyright 1977 Greeter's Anonymous. All rights reserved
2  /** classic first code example */
3  public class HelloWorld {
4     /** //
5      * writes "Hello, World" to the console
6      * @param args the command line arguments
7      */ //
8     public static def main(args:Array[String](1)) {
9       x10.io.Console.OUT.println("Hello, World");
10    }
11 }
```

The working copy of this code is `intro/HelloWorld.x10`. Let's step through it, a line at a time:

**line 1:** `// Copyright 1977 Greeter's Anonymous. All rights reserved`
Comments in X10 are the same as Java or C++: they either begin with "`//`" and go through the end of the line, or begin with "`/*`" and end at the *first* "`*/`" that follows. Because the first "`*/`" ends the comment, "`/*...*/`" *comments do not nest*.

**line 2:** `/** classic first code example */`
Comments in the style `/** ...*/` that immediately preceed a declaration are

[1] See file HelloWorld.x10

called "X10Doc" comments. X10Doc is a set of conventions for publishing the APIs for whole directory trees of X10 source files. X10Doc follows that same format as Java's "JavaDoc", but instead of using the "javadoc" command, you use the "x10doc" command to process it. The final product is a nicely organized HTML site. All of our sample code uses X10Doc documentation, but we usually do not copy it into the displays in the text. The documentation for the X10 library http://dist.codehaus.org/x10/xdoc/ is all generated from X10Doc comments. [2]

**line 3:** `class HelloWorld {...}`

X10's classes serve essentially the same purpose as classes in other object oriented languages, Java and C++ in particular. There are some differences, of course, which we'll point out as they arise in the discussion.

A class normally will have the same name as the file in which is declared—*e.g.* `HelloWorld` is found in `HelloWorld.x10`. C++ programmers should realize that, unlike C++, X10 relies on file names to find class declarations. We'll go through the rules in detail in Chapter 2.

**line 6** `* @param args the command line arguments`

Inside X10Doc comments, lines that start with an asterisk ("*") are copied into the HTML with the asterisk (and leading spaces) deleted. This line says that `args` is an argument for the method whose declaration follows, namely `main()`.

**line 8:** `public static def main(args: Array[String](1))`

Program execution starts, as in Java and C++, with a method named `main`, which takes the command-line arguments as a collection of strings. But here we start to see some differences in syntax from Java and C++:

- The keyword `def` begins a method declaration. This makes it easy to tell what is a method and what isn't. (In contrast, in Java and C++, you have to look for clues – sometimes small ones – to tell the difference between a method and a field.)
- There is no return type specified here. In fact, *you don't normally need to specify the return type for a method.* This differs from both Java (you must supply it) and C++ (the default is `int`). The X10 compiler looks for the return statements in a method and normally can infer the return type.

  If you *do* wish to specify it, then, unlike Java and C++, it *follows* the argument list. For example, "`def doIt(t: T): U {...}`" declares a method named `doIt` with one argument of type `T` and a return value of type `U`. Notice the '`:`' that preceeds the types in both places: it is the required syntax. The return type of `main()` is `void`, meaning that no value is returned.

  So why might you specify the return type, if you don't have to? Basically, writing down the return type gives valuable documentation, particularly for longer methods where the return type is not immediately obvious. It

[2] If you want to really get to know JavaDoc, its home page is at oracle.com, but for a quick executive summary sufficient for almost all purposes, see the Wikipedia entry.

also prevents some mistakes: if you expect to return a `Boolean`, but one of the seventeen **`return`** statements accidentally has an `Int` instead, X10's type inference will happily pick `Any`, which can be anything, as the return type of the method. If you specify the `Boolean` return type, X10 will flag the `Int` return as an error, which is probably what you want.

Occasionally, the compiler will force you to specify a return type to resolve type checking difficulties. For example, X10 might infer that method `m` is `Boolean` — but in a subclass, you want to have `m` return other kinds of things. So you would need to declare `m` to be `Any`, rather than the inferred `Boolean`, in the parent class.

- X10 has generic types, along the same general lines as Java and C++. X10 uses square brackets to hold the actual type, *e.g.* `Array[String]` for declaring an array of `String`s.
- X10 arrays, like FORTRAN arrays, may be multi-dimensional. The "`(1)`" that follows `Array[String]` asserts that the array is one-dimensional, or in other words, is just like the usual Java or C++ array.
- The general syntax for assigning a type to an identifier is

  *identifier*: *type*,

  as in `args:Array[String](1)`. White space before or after the '`:`' is ignored by the compiler. In fact, white space in X10 is treated as it is in Java and C++: normally ignored, except to the extent that it is needed to separate tokens or appears in string literal constants.

**line 9:** `x10.io.Console.OUT.println("Hello, World");`

Like Java, X10 groups classes into units called "packages". For example, the input-output classes in X10's standard library all belong to the package `x10.io`. The class `Console` is part of that package. Package names are used both as prefixes to provide unique names for classes *and to locate the classes*.

`Console.IN`, `Console.OUT`, and `Console.ERR`

are the standard input, output, and error streams. The method `println` prints a string, followed by an operating-system dependent line-ender: either a single newline character for Unix based systems, or a carriage return-newline pair. If you don't want the newline, use `print` instead.

We'll give some more details about packaging in chapter 2.

Here's the command for compiling `HelloWorld` for execution by a Java-based runtime ("`%`" is the command line prompt):

```
% x10c HelloWorld.x10
```

To run it, you use the `x10` command:

```
% x10 HelloWorld
Hello, World
```

If `HelloWorld` had required some command-line arguments, you could have added them at the end of the command line.

There is also a C++ runtime. To use it, you need to compile using `x10c++` rather than `x10c`. The usual C compiler convention `-o` *`filename`* for naming the executable is used.

```
% x10c++ HelloWorld.x10 -o hello
% runx10 hello
Hello, World
```

You'll need the `runx10`: you cannot invoke `hello` directly, because some special setup is required to bootstrap the X10'c C++ runtime. We don't actually need anything special for `hello`, but X10 code that sets up the call to `main()` *is* generic and needs information supplied by `runx10`.

## 1.2 Two CPUs Are Better Than One

The point of X10 is concurrent programming: giving you control over clusters of multiprocessors. We'll get started on this by parallelizing a simple piece of serial code that computes an approximation to the number $\pi$. Along the way, we'll introduce some more X10 syntax and write our first loops.

### 1.2.1 $\pi$ via Monte Carlo

The unit circle is the set of points $(x, y)$ in the plane that satisfy $x^2 + y^2 \leq 1$, and its area is $\pi$. We are going to explore a particularly simple method of estimating $\pi$. Figure 1.1 shows the one-quarter of the unit circle that lies in the unit square, $0 \leq x, y \leq 1$. The unit square has area 1, and the shaded part inside the circle has area $\pi/4$. Now imagine picking points at random in the unit square. What fraction will also lie in the unit circle? If the points are really random, the answer ought to be the fraction of the square that lies inside the unit circle, namely: $\pi/4$.

One way to estimate $\pi/4$, then, is to pick a large number of points $(x, y)$ in the unit square at random and see what fraction actually land in the unit circle. This sort of process is called a "Monte Carlo" algorithm.

If ever there were an easily parallelized type of algorithm, Monte Carlo is it: if we have 1,000 processors, we let each generate points independently, and at the end, we just have to merge the results. The only trick is to make sure that each of the 1,000 processors starts in a way genuinely random with respect to the others, so that they don't just duplicate each other's efforts.

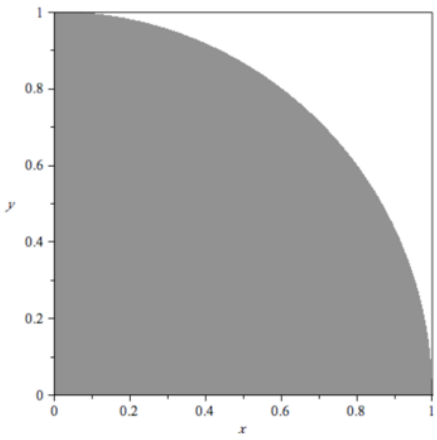


Figure 1.1: The intersection of the unit circle with the unit square

### 1.2.2 Getting Started: A Serial Version

Let's look first at a serial version in Figure 1.1, because it introduces a number of X10 idioms that we'll need in the parallel version. You can find the source in MontePi1.x10. [3]

**line 1:** `Random` is a class from the X10 standard library. Whenever you need a class that is not implemented in the file you are editing, the compiler needs to be told how to find it. There are two ways to do so, which are the same as in Java:

1. You can specify the *fully qualified name* of the class in an `import` statement, as line 1 does for `Random`. In the next chapter, we'll go into more detail about the naming conventions.
2. Or, you can omit the `import` statement, but if you do, you have to write out the fully qualified class name at each use. Iif we had left it out here, we would have had to rewrite line 5 as

   ```
   val r = new x10.util.Random();
   ```

   The standard X10 library provides some types which are so commonly used that the compiler is kind and does not force you to import them explicitly or write them out in detail at each use. `Int`, for example, is really `x10.lang.Int`, and `Console` in line 13 is really `x10.io.Console`.

**lines 3, 5, 8, 9, 12,** In line 3, we declare `N` to be a `static val`. The keyword "`val`" means that `N` names a value, `10000` in this case. One cannot assign a new value

[3] See file MontePi1.x10

```
1  import x10.util.Random;
2  public class MontePi1 {
3      static val N = 10000;
4      public static def main(s: Array[String](1)) {
5          val r = new Random();
6          var inCircle:Double = 0.0;
7          for (j in 1..N) {
8              val x = r.nextDouble();
9              val y= r.nextDouble();
10             if (x*x +y*y <= 1.0) inCircle++;
11         }
12         val pi = 4*(inCircle/N);
13         Console.OUT.println("Our estimate for pi is " + pi);
14     }
15 }
```

Listing 1.1: Serial Monte Carlo Approximation of Pi

to `N` later on in the code: it is a constant, in the same way the `const` is used in C and `final` in Java.

The keyword "`static`" means that the value is associated with the class `MontePi` itself. The remaining `val`s, `r`, `x`, `y` and `pi` are not part of the class: they are just local variables of the method `main()`.

The compiler will happily figure out the type of a `val` whose value appears in its declaration, as it does in each of the five declarations here. You can, if you wish, provide the type yourself. For example, we could have written line 5 as

```
val r:Random = new Random();
```

There's not much point in this case, though, to spelling things out. Adding "`:Random`" helps neither people nor the compiler read the code.

**(B: I added a discussion of $<:$. This probably needs to get moved; it's too long for here. :B)** *Most of the time, you should not specify the type of a* `val` *whose initial value appears with its declaration.* You can trust the compiler to come up with a correct type for your value, or to give you an error message.

You might want to give a type for the sake of someone reading the program – someone like yourself six months later, or even the compiler. You can give an exact type, and insist that all that the compiler can know about the variable is the type you give it, with the `:Random` syntax.

It's usually better to give *approximate* type information, where you tell *some* things about the variable – the things you expect people reading your code to care about, and the things you want the compiler to check. The compiler will

figure out the exact type, but it will check everything that you said you wanted. The syntax for this is to use "<:" instead of just ":", like this:

```
val r <: Random = new Random();
```

For `Random`, there aren't a lot of choices for what you could say for partial information. If you were giving another name to `s`, the one-dimensional array of `String`s declared on line 4, you have more choices: [4]

```
1 public class ArgvPartialInfo {
2     public static def main(s: Array[String](1)) {
3         val a <: Object = s;
4         val b <: Array[String] = s;
5         val c <: Array[String]{rank != 3} = s;
6     }
7 }
```

On line 3, we give very little information, just saying that `a` is an `Object`. On 4, we say that `b` is an array of strings, but, unlike the declaration of `s` on line 2, we don't say that it's a *one-dimensional* array. On 5, just because we can, we say that it's an array but not a three-dimensional array: it could be one-dimensional or four-dimensional or eighty-dimensional.

The difference between ":" and "<:" in `val` declarations is that ":" erases information and "<:" doesn't. So, if you write

```
val b : Array[String] = s;
```

all that X10 will know about `b` is that it is an array of strings, of some unknown dimensionality. But if, as on line 4, you write

```
val b <: Array[String] = s;
```

X10 will remember that `b`, like `s`, is one-dimensional. This is important information later on if you want to subscript it with a single index, `b(0)`. That's the right thing to do with a one-dimensional array, but wrong for an array whose dimensionality is unknown. **(B: This new bit has gotten quite long, and should be put into a separate section :B)**

You *do* need to provide the type when you don't want to initialize the `val` in the declaration itself. Typically this happens when the `val` depends on some choices that you can't neatly write in one line:

[4] See file ArgvPartialInfo.x10

```
val howMany: Int;
if (aBoolean) {/* howMany gets set one way here */}
else {/* and gets set differently here */}
```

The `if` and `else` blocks are both free to do any calculation they need to in arriving at `howMany`'s value, so long as they don't try to use `howMany` itself before it has been set. The rule is

> *Control cannot reach a use of a* `val` *without first reaching an assignment that sets the* `val`*'s value.*

In other words, there is no such thing as "default value" for a `val`. It must be set explicitly by you, and once set, cannot be changed.

The initializer for a `val` may be any legal X10 expression that can be evaluated at run-time—they need not be compile-time constants. Lines 8, 9, and 12 all show examples of initialization expressions.

**lines 6:** The keyword `var` introduces the declaration of a variable. A declaration like "`var inCircle:Double = 0.0;`" says that `inCircle` names some storage that holds a value of type `Double` whose initial value is `0.0`, and this value may be updated as the code runs. In the lingo of the trade, one says that "`inCircle` *references* a `Double`".

`Double` values are double-precision IEEE floating points, exactly like Java's and C++'s `double`.

You do need to supply the type for a `var` even when an initial value is provided. The rationale is a bit involved, so we ask you just to take our word for it that for now, the compiler needs to be told the type of every `var`.

**lines 7- 11** Another new ingredient in the code is the "`for`" loop, lines 7 through 11. `for (j in 1..N) {` executes its body (lines 5-10) once with `j` bound to `1`, then once with `j` bound to `2`, and so on, ending (if nothing stops it sooner) with `j` bound to `N`.

`1..N` is an example of an `IntRange`, a value that describes a range of integers. They're useful for looping over, as we do here. They are also useful for declaring the subscripts of arrays, and a variety of other situations.

X10 has old-style **`for`** loops too, like `C++` and `Java`'s traditional **`for`** loops. We could have written this one: [5]

```
1 for(var j:Int = 1; j <= N; j++) {
2   //...
3 }
```

[5] See file OldStyleFor.x10

(`j` needs to be declared as a `var`, not a `val`, because we're changing it. `j++` means "Add one to `j`," just as it does in C and Java.)

Each time through the loop, we call the random number generator `r` twice to get the coordinates of a point (lines 8, and 9). Happily, `r.nextDouble` returns a value between 0 and 1, so we can use it "as is." In line 10, we check whether the point lies in the unit circle, and if so, we increment `inCircle` by 1.

As it happens here, we don't need `j` in the loop, but we do want to emphasize that *the scope of* `j`*'s declaration is the loop and nothing but the loop*, so when you leave the loop, `j` will be unavailable.

**line 12:** On exit from the loop at line 12, `inCircle/N` is the fraction of points in the circle, which is going to be a positive number less than 1, so we have to use a `Double` (or if we don't care about the precision, a `Float`) to capture the value. That's why we made `inCircle` a "`Double`" in line 6. When we do the division here, the compiler will arrange to convert `N` to a `Double`, too, and will use double precision floating point division.

Suppose we had said to ourselves, "incrementing a `Double` by 1 inside that loop has got to be more expensive than incrementing an `Int`. So let's declare `inCircle` to be an `Int`." That's fine, but when we get to line 12, we have to be careful to convert it to a `Double`. One way to do this is with `as`:

```
val pi = 4 * (inCircle as Double / N);
```

Time to try compiling and running the code. Here is our console log for the run:

```
% x10 MonteCarloPi
The value of pi is 3.1368
```

Not a brilliant guess at $\pi$, but we didn't really try all that many points. Your answer might vary: in fact, the answer will vary with each run because, whenever `Random` creates a new generator, it uses the current time to create a new starting point for computing its values. The first two digits, 3.1, though, should be stable. Good luck!

### 1.2.3 We Can Do Better

There are some pretty primitive aspects to our first cut at $\pi$. In this section we'll introduce a few features of X10 that will help us spruce up the code a little bit.

To begin with, that "`static val N = 10000;`" in line 3 is really, truly rigid. We have access to a perfectly good set of command line arguments. Why not use the first, if supplied, to set the number of points to try? That would have let us try 1,000,000 points right away to see how much better we could do than 10,000. The code we need is simple enough:

6

```
1 public static def main(args: Array[String](1)) {
2     val N = args.size > 0 ? Int.parse(args(0)) : 10000;
```

Some comments:

**line 1:** The command line arguments come in as the array `args`. The declaration

```
args:Array[String](1)
```

should, as we have already mentioned, be read: "`args` *is a value whose type is an array of strings indexed by a single integer.*" X10 arrays, unlike Java or C++ arrays, may be indexed by arbitrarily many integers, so you have to tell the compiler what sort of indexing you want. Don't be confused here: the "`(1)`" is not the size of the array: it is the *kind* of index you need (a single integer, in this case).

**line 2:** X10 uses the property `size` to get the number of elements in an array. Using "`length`", as Java does, would be misleading, because X10 arrays can be $n$-dimensional, not just 1-dimensional.

X10 uses ordinary parentheses, and not square brackets, to access array elements. Thus, since `args` is indexed by a single integer, `args(k)` is the entry in `args` indexed by the integer `k`. X10, like its relatives C++ and Java, normally starts array indexing from 0. However, the X10 programmer can specify other index domains. There is no reason to get fancy about `args`, however, so its first element is `args(0)`. Arrays in X10 are a whole subject unto themselves that we will get to in Chapter 3.

We use `Int`'s static method `parse` to convert the command line input from a `String` to an `Int`. Then the conditional operator, "`?:`" allows us to choose a value: it begins by testing its first operand, which must must evaluate to **`true`** or **`false`** , *i.e.* a `Boolean`. If it evaluates to **`true`**, the value of the expression is the second operand; otherwise the value is the third. Once again, X10 is consistent with Java and C++.

The next step in improving the code is a little more involved. We used a random number generator that the X10 library provided for us. Suppose, though, that for some reason we wanted to try another one. It would be nice if the generator were just another parameter to the computation, so we could play with a bunch of them if we wanted to. To get there, we are going pull the main loop out of `main` and put it in its own method, one parameter of which is the random number generator: [7]

[6] See file MontePi2.x10

[7] See file MontePi2.x10

```
1 public static def countPoints(n: Int, rand: ()=>Double) {
2   var inCircle: Int = 0;
3   for (j in 1..n) {
4     val x = rand();
5     val y = rand();
6     if (x*x +y*y <= 1.0) inCircle++;
7   }
8   return inCircle;
9 }
```

The new ingredient here is the declaration of `rand` in the first line. As usual, its type follows a colon (":"), but what's there is not just a name, as usual, but a sort of "expression", `()=>Double`, which is read: "*a function that takes no arguments and returns a* `Double`." There is no strictly analogous construct yet in Java, although one is planned, and the closest thing in C++ is the "function pointer" `double (*rand)()`.

Note, too, that we compute `inCircle` as an `Int` here. It's a little faster than using `Double` and besides, we know that the result is an integer, so it makes sense to declare it as such.

How do we create the function to pass in as `rand`? Here's one approach: in `main()`, put [8]

```
1 val r = new Random();
2 val rand = () => r.nextDouble();
3 val inCircle = countPoints(N, rand);
4 val pi = (4.0 * inCircle)/N;
```

**line 1** says that `rand` is a function with no arguments whose body is the expression `r.nextDouble()`, which is its return value. This is, as you would guess, just a simple example of a much more general facility, and we'll see a lot of examples later that will flesh out how to use it.

One important thing to understand is that the declaration of `rand` captures the runtime value of `r`. If we put this code in the body of a loop, then each time through the loop, `rand` would use the new value of `r` that is yielded by the constructor `new Random()`.

The right-hand side of the declaration is often called a "closure" in the literature (because of the way variables from the surrounding context (like `r` here) are captured and kept until needed). You'll also see languages like X10 describing themselves as supporting "first-class functions", which is short for "functions as first-class data" and simply means they allow you to work with functions in exactly the same way you would with any other sort of data, like `Int`s or `String`s: you can assign one, pass it as an argument, save it as an array element, *etc.*

[8] See file MontePi2.x10

**line 3** replaces the whole loop in lines 7 through 11 of our original with the call to our new method `countPoints`.

**line 4** We use a factor of `4.0` here, rather than `4` as we did in our original. The type of `4.0` is `Double`, which forces the whole expression to be treated as a `Double`. (We could also have used `as Double`, as we did before.) We'll say more about this sort of automatic conversion in section 3.1.

The cleaned-up version of this code is MontePi2.x10.

There are one or two things we could do to pretty it up even more, but enough for now. It is time to look at how to parallelize it.

### 1.2.4 Enter The Second Processor

We are going to present several parallel versions of our code. We'll begin with a version that assumes shared memory: multiple threads running on a single machine. Most PCs these days have dual-processor CPUs, so a factor of 2 speedup is available right out of the box, *if* your code can effectively use both processors.

For our $\pi$ calculator, the changes are simple: we just have to be able to say "start $n$ activities going, each with its own independent random number generator, and when each has done it's share of the work, sum the hits from all $n$ and divide by the total number of points tried."

X10 avoids the term "thread", because it (together with "process" ) has a variety of meanings in different contexts. Instead, X10 uses the term *activity* to mean a sequential thread of control. We'll be more loose here and use whichever term seems more natural (to us!) at the moment, but you should be aware of X10's convention when reading other literature.

Figure 1.2 shows the relevant part of our new, parallel `main`. We'll go through it line by line.

9

**lines 1-4:** If command-line arguments are supplied, we read the number of points to try, `N`, from the first, and the number of activities to use, `nAct`, from the second. Otherwise we just use 100,000 points and 4 activitiess. The number of points each thread will try is `nPerThread`. Since we are now using multiple activities, we've switched to `Long` integers—64 bits—instead of ordinary `Int`s.

**line 5:** The right-hand side is the X10 idiom for constructing an array that is indexed by a single integer running from `1` to `nActs`. The entries in the array are initialized to `0L`, the `Long` way to say zero, which is the default value for a `Long` that doesn't have its value specified some other way. You might wonder what declaring `inCircle` here to be a "`val`" implies: what is it that cannot be changed

[9] See file MontePiAsync.x10

```
1  val N = args.size > 0 ? Long.parse(args(0)) : 100000L;
2  val nAct : Int = args.size > 1 ? Int.parse(args(1)) : 4;
3
4  val nPerAct = N/nAct;
5  val inCircle = new Array[Long](1..nAct);
6
7  finish for(k in 1..nAct) {
8    val r = new Random(k*k + k + 1);
9    val rand = () => r.nextDouble();
10   val kval = k;
11   async inCircle(kval) = countPoints(nPerAct, rand);
12 }
13
14 var totalInCircle: Long = 0;
15 for(k in 1..nAct) {
16   totalInCircle += inCircle(k);
17   Console.OUT.println("ic("+k+") = "+inCircle(k));
18 }
19
20 val pi = (4.0*totalInCircle)/(nPerAct*nAct);
```

Listing 1.2: Shared-memory parallel code for computing $\pi$

because `inCircle` is a `val`? The answer is that `inCircle`'s value is always going to be the array it is initialized by the right-hand side of this line, but during the program's run, the individual elements of that array may be assigned to as needed.

**line 1:** The number `100000L` is the `Long` version of `100000`. You can also use a suffix of `S` for `Short`, and `Y` for `Byte`[10], and any of these with a `U` for the unsigned version.

**line 7:** There is nothing unusual about the "`for`" loop part of this line. The interesting part is the "`finish`". The whole point of this loop is to spawn some number of independent activities, each computing how many hits out of a possible `nPerAct` land in the circle. We can't do any further processing until we are sure that all of these activities have run to completion. That is what "`finish`" guarantees: control will not reach the statement after that guarded by a `finish` until all of the activities spawned in the `finish`'s statement have completed. So when we get to line 11, we can be sure that every entry in `inCircle` has been correctly set.

`finish` is followed by a statement, like `for` in this example. It can be any statement, though. If you want to start two activities and wait for them to finish, you can write:

[10] `B` is a hexadecimal digit, so X10 couldn't also use it as a byte marker.

11

```
finish {
  async do_this();
  async do_that();
}
```

**line 8:** The constructor `Random()`, if called with no arguments, uses the number of milliseconds from some fixed time as the "seed" to begin generating its random numbers. Alas, less than a millisecond may elapse between the creation of two or more of our activitiess, and when that happens, we get the same sequences in several activities, not the independent sequences we need. So we've spiced things up by using a simple polynomial in the loop counter `k` to generate a unique seed for each activity. Why not just `k`? Why `k*k + k +1`? Just to spread the starting points out a little more in the hope that our activitys really get independent sequences.

**lines 10 and 11:** This is X10's idiom for spawning an activity at the current processor. The `async` statement is readied for execution in its own activity, and control may then be returned to the originating activity whenever the X10 runtime's activity manager wishes: it could be immediately, or it could be after the new activity has been allowed some time. The important point—particularly if we are running on a multi-processor—is that we do not have to wait for the new activity to complete before returning control to the original activity. The effect of the `for` loop is, therefore, to get `nActs` activities up and running *concurrently*.

`async`, like `finish`, can be applied to any statement. If you want to start a multi-statement activity, use a { block }:

12

```
1 async {
2   do_this();
3   do_that();
4 }
```

Why do we introduce `kval` in line 10? Using `kval`, which is a constant, in the `async`, rather than `k`, which is a variable, ensures that that activity is using the value of `k` in force when the `async` is spawned. Were we to use `k` inside the `async`, and it happened that the execution of the `async` got sufficiently delayed, the `async` could see a value of `k` greater than that at time of the `async`'s creation, and then it would set the wrong array entry in line 11.

[11] See file FinishTwo.x10
[12] See file MultiLineAsyncExample.x10

Line 11 ends by calling `countPoints`, as in our serial code. We've allowed its first argument, the number of points to try, to be a `Long`. 64-bit integers are probably overkill here, but with so many activities at our. command, we can afford to think *big*.

**line 15:** When we get to line , we can compute the total number of hits in the circle out of the `N` points we generated, because we can be sure that every entry in `inCircle` has been correctly set.

**line 20:** Once the loop in lines 15-18 has computed the total number of hits in the circle, we have to normalize the result to get the final answer. `N` is the total number of points we wanted to try, and each activity actually got `N/nAct == nPerAct` points to try. If `nAct` doesn't divide `N` evenly, though, we only wind up using `nPerAct*nAct` points, which explains the denominator here. Using a `Double`, `4.0`, as the first term in the right hand side forces the compiler to generate code for converting the two `Long`s to `Doubles` before the arithmetic is performed.

Here's our console log for running the code from a Unix-style terminal window. The Unix `time` command executes the program and then produces three time estimates: total CPU usage ("real"), wall-clock elapsed time ("user"), and system overhead ("sys"). This timing is for a 3GHz dual-processor laptop.

```
% time x10 MontePiAsync 10000000 1
The value of pi is 3.1402504

real 0m2.426s
user 0m3.368s
sys 0m0.165s
% time x10 MontePiAsync 10000000 2
The value of pi is 3.1405268

real 0m1.370s
user 0m1.893s
sys 0m0.109s
```

Not bad: an overall factor of $3.368/1.893 = 1.78$ speed-up for the observed time-to-completion out of a best possible of 2.

You can find the whole program in `MontePiAsync.x10`.

## 1.3 A Thousand CPUs Are Better Than Two

### 1.3.1 Distributing Work

To get heavy-duty concurrency, we have to distribute work across many processors, which usually means we have to scale out to more than one (shared-memory) machine.

To this end, X10 provides a type, `Place`, that is best thought of as an address space in which activities may run. The physical reality is that different `Place`s may refer to the same physical processor and may share physical memory, but from the programmer's point of view:

- The running program has a single address space.
- The distinct `Places` partition that address space: no two `Places` have any storage in common.
- But, since there is a single global address space, an activity at one `Place` may refer directly to storage at another.

`Place.MAX_PLACES` is the number of `Places` available to a program. It is fixed at program start-up and cannot be altered thereafter.

Each `Place` has an integer id: if `p` is a `Place`, then `p.id` is its id. An activity can find out at which `Place` it is executing by evaluating the expression `here`. The keyword `here` is reserved for this purpose alone. The id of the current activity's `Place` is `here.id`. The `Place` whose id is "`i`" can be got by evaluating `Place.place(i)`.[13]

The X10 runtime begins a program's execution by creating a single activity, the "root" activity, that calls the program's `main()`. The root activity's home `Place` is called `Place.FIRST_PLACE`, and by convention, it is the `Place` whose id is `0`.

So the question is: how does information at one `Place` get to another? One simple way is to use an "`at(p)`" statement: if the identifier `p` names a `Place`, and if `computeAnInt()` is a method that computes an `Int`, then

```
val anInt = at(p) computeAnInt();
```

means: **(B: s/thread/activity/ in the following :B)**

> Pause the thread for this activity. Go to the `Place p`, and resume the activity by calling `computeAnInt()` in a thread at `p`. Send the result back here to the original `Place`, restart the thread there for the activity by assigning the value to `anInt`. The activity then continues at the original `Place`.

[13] If you've programmed using the MPI library, `Place.MAX_PLACES` is analogous to what you get by calling `MPI_Comm_size` and the `id` is analogous to what `MPI_Comm_rank` gives you. The UPC equivalents are THREADS and MYTHREAD.

If you like to think about implementation, you can think of the single X10 activity being built out of two threads: the thread for the requesting activity and the thread for the remote activity. From the point of view of the X10 programmer, though, exactly one activity is going on here, because the requesting thread is blocked while the remote thread does its thing, so no matter how many "hardware threads" we use to carry out this computation, it is strictly serial: it is one activity. This is one reason why X10 uses the term "activity" for a serial computation, rather than "thread."

If you do not want to wait around for the value to be computed and assigned, things are not so simple. You might think, for instance, that something like the obvious "`async val anInt = at(p) computeAnInt();`" might work, but it doesn't. Just as with variables declared in `for` loops, the declaration of `anInt` in an `async`'s body means that it is not available outside of it. This is consistent with Java and C++ (and just about every other language): a declaration within a statement's body is visible only in that body.

The secret is to separate the assignment from the declaration:

```
val anInt: Int;
finish {
   /* some code not using anInt can go here */
   async { anInt = at(p) computeAnInt(); }
   /* maybe more code not using anInt here, too! */
}
/* at last: anInt can be used here! */
```

As the comments in this code suggest, the `async` has to be inside a `finish` block, and `anInt` cannot be used until control leaves the block.

Passing a "`var`" into an `async`'s block is also possible, but (of course) risky because of the possibility of race conditions, an example of which we'll discuss in the next section. If you change `val` to `var` in line 1, the code works as before, but be aware that the `finish` must be present in the same scope as the `var` that is used in the `async`. That is, you cannot just spawn an activity into which you pass a variable unless that activity has a visible bound on its lifetime. For example, the code

```
def syncIt() {
    var anInt: Int;
    async anInt = 3; // compiler won't accept this
}
```

will cause the compiler to complain:

```
Local variable "anInt" cannot be captured in an async if
there is no enclosing finish in the same scoping-level
as "anInt"; consider changing "anInt" from var to val.
```

What the compiler wants is a `finish` surrounding the `async` within the scope of the declaration of `anInt`. To be precise, what will *not* work is:

```
def syncIt() {
  finish {
    var anInt: Int; // anInt is local to the finish block
    async anInt = 3;
  }
  use(anInt); // no! anInt is not defined here!
}
```

What *will* work is inserting the `finish` so that `anInt` is alive and visible when the `finish` completes:

```
def syncIt() {
  var anInt: Int;
  finish {
    async anInt = 3;
  }
  use(anInt);
}
```

Now that we know how to move data around and in and out of `asyncs`, we are ready to rework our code.

### 1.3.2 A First Try At Multi-Place Code

Let's get started with our multi-processor code with some high-level pseudo-code. We are going to go all out and not only use several `Places`, but at each `Place`, we'll use several activities. Here we go:

`main`: Read the command line to get the number of places to use. For each `Place`, call the function `countAtP` to get that one `Place`'s contribution, and add up all of them to get the final answer.

`countAtP`: Add up the counts from several threads at one `Place`.

`countPoints`: Called once per thread. It is the same as its namesake in `MontePi2` and `MontePiAsync`. This is where we actually call the random number generator to get the points to test.

In thinking about this code, keep in mind that a really high-performance computer can provide literally thousands of `Places`, but, for this sort of CPU-intensive activity, at any given `Place`, it is likely to be worthwhile running at most a dozen or so threads,

probably fewer. That said, we might ask ourselves whether it makes sense to use different strategies for accumulating our results in `countAtP`, which we expect to have very few contributors, versus `main`, which may have thousands.

When we only have two or three integers to add together, it might make sense to use a single `var count:Long` to accumulate the total count, rather than using an array (as we did in the `main` for `MontePiAsync`). Here's a first cut:

```
1  public static def countAtP(pId:Int, threads:Int, n:Long) {
2    var count: Long = 0L; // 0L == Long integer literal 0
3    finish for (var j: Int = 1; j<= threads; j++) {
4      val jj = j;
5      async {
6        val r = new Random(jj*Place.MAX_PLACES + pId));
7        val rand = () => r.nextDouble();
8        count += countPoints(n,rand); // trouble, as we'll see
9      }
10   }
11   return count;
12 }
```

Sadly, this code has a very nasty bug, which is due to the variable `count` being shared by the whole set of threads. The trouble is in line 8, where `count`'s value is updated. Broken down, line 8 involves the following steps:

**Step 1:** Load the value of `count` into the CPU.
**Step 2:** Call `countPoints`.
**Step 3:** Add the return value from the call to the loaded value of `count`.
**Step 4:** Copy the sum from the CPU into `count`.

If you are a veteran of the parallel programming wars, you will recognize this as a classic opportunity for a *race* condition. For the newcomers, here is a scenario that shows what can go wrong:

> We begin at line 2 with `count`'s value is initially set to 0. Suppose that the value of `threads` coming in is `2`. The `async` in line 5 will then get called twice, so we'll have two threads, call them `T1` and `T2`, executing the code in lines 6-8 in parallel. Figure 1.2 is a graphic view of one possible time-line for the two threads.
>
> Suppose that `T1` begins executing first. When it gets to line 8, and does the first step: it loads the value of `count`, which is still 0. But now comes trouble: suppose that just after this step completes, the operating system's thread manager suspends `T1` for some reason.
>
> If a few nanoseconds later, the thread manager lets `T2` start *rather than restarting* `T1`*, which, after all, is only fair, since* `T1` *already got at least*

*some CPU time*. When `T2` gets to line 8, it, too, loads `count` into the CPU. But, because `T1` never completed updating `count`, `T2` finds the same value, 0, stored there that `T1` did. So when `T2` gets to step 3 in its execution of line 8, it adds the return value to 0, and stores the result back into `count`, so `count` now is whatever `T2` computed.

At some point `T1` will be restarted, and because it will start executing exactly where it was suspended, it will be at step 2, the call to `countPoints`. `T1` will *not* repeat the first step, loading `count`. Threads also restart exactly where they where suspended. So `T1` add its contribution to what it thinks the value of `count` is—namely, 0. The result is that `T1` winds up overwriting `T2`'s value in `count` with its own, not adding the two, which is what we wanted.

Both activities having now completed, `T2`'s contribution has been lost.

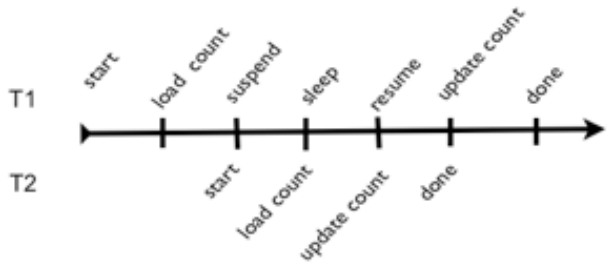


Figure 1.2: One possible timeline for `T1` and `T2`

Disaster!

You can see why this is called a race. It is a particularly insidious sort of bug, because sometimes you get the right answers, and sometimes you don't. After all, the operating system did not *have* to suspend `T1` at a bad moment. It just happened to. The event that led the thread manager to suspend `T1` may have come from an external event having nothing to do with `MontePi` at all. Maybe it was an I/O event of some kind that simply had to take precedence over `T1`—so, too bad: `T1` loses. That's life!

The cure is simple enough: we just need to replace line 8 with some code that ensures that once one activity starts executing that code, *no other activity can begin that code until the first one finishes.*

```
8 atomic count += countPoints(n, rand));
```

Guarding an X10 statement this way, with the keyword `atomic`, does just what we want: we are guaranteed that once an activity enters the statement, no other activity may enter it until the original activity completes it.

In our original scenario, this means that once T1 starts executing line 8, T2 will be blocked from entering line 8 while T1 is still active there, even if for some reason the operating system suspends T1 for a while. T2 will be suspended when *it* reaches line 8, at least until T1 finishes there. This slows things down, but you get the right answer.

Even though our new line 8 is correct, it really is not a good solution. The problem is that virtually all the time spent executing the statement is in the expensive call to countPoints. countPoints does not depend on any resources shared by the two threads, so there is no problem about the two executing the call to countPoints concurrently. The only shared resource is count, which doesn't appear in countPoints at all. So what we really need to do is to split the line into two:

```
8 val countForJ = countPoints(n, rand);
9 atomic count += countForJ;
```

Because only one activity at a time can execute an atomic statement, clearly the smart thing is to keep it as small as possible.

Races can occur whenever multiple activities share a resource. In our example, they shared a piece of storage, count, but they could equally well, for example, share an output stream. Suppose three activities call Console.OUT.println at the same time. What happens? Answer: it depends! Or perhaps better: "We're off to the races!" Sometimes each line will print as desired, sometimes the two lines will be interleaved, and sometimes all three will be. Try the following code, for example, on your machine a bunch of times:

```
public class HelloAsync {
   public static def main(argv:Rail[String]!) {
      async Console.OUT.println("Hello, World");
      async Console.OUT.println("Hola, Mundo");
      async Console.OUT.println("Bonjour, Monde");
   }
}
```

Here's our console log for our own first shot at running it:

```
% x10 HelloAsync
BHelonlo,j oWuorr,l dM
onde
Hola, Mundo
```

"Hola, Mundo", got delivered intact, but "Hello, World" and "Bonjour, Monde" got pretty well interleaved. Who knows what might have happened on another run! Even

more amusing: why was the *second* of the three `asyncs` the one that wasn't interrupted? The first, maybe–the last, not too surprising–but, the *second*? Moral: where there are races, the outcomes really are unpredictable.

Bad as this behavior is, it is all that X10 can reasonably guarantee, because enforcing atomicity—on calls to `Console.OUT`, for example—is not free, and knowing exactly when and where it is required is something that really only the programmer can know. So X10's `atomic` statement makes it easy for the programmer to enforce atomicity, and that's really the best thing.

Putting all this discussion to use, we get a first cut at `MontePiCluster` in Figure 1.4.[14] [15]

You've now seen almost all of the principal ingredients for writing multi-threaded X10 code: `async` to spawn activities, `finish` to know when a set of activities is complete, `at` to shift the action to some other processor, and `atomic` to maintain the integrity of access to shared data. We'll fill out the picture in Chapter 4. But first we want to take some time to look at how object oriented programming looks in X10.

**Exercise:** You might want to try the following slight variation on the Monte-Carlo code. Write a method that uses one or more `Places` to sum the values of a function `f:(d:Double)=>Double` over a sequence of `n Doubles d, d+delta, ... d+n*delta`:

$$\sum_{k=0}^{n-1} f(d + k\delta) \tag{1.1}$$

For debugging sake, try some simple `f`'s to begin, like `(d:Double) => 1.0`. Then you can go hog-wild using your own functions (*e.g.* `(d:Double)=>d*d`), or the functions in `x10.Math`, like `sin`, `log`, and `sqrt`.

The amusing questions are: what part of the sum is a given activity responsible for, and how do we combine the partial results? Combining the results is essentially the same as what we've done for `MontePiCluster` here. Splitting up the sum, though, requires some thought. Here's one approach:

If there are $p_{all}$ `Places` in all, and there are $a$ activities in parallel at each `Place`, there will be $ap_{all} = a_{all}$ activities in all. Since there are $n$ values to be summed, each activity should handle roughly $n/a_{all}$ additions—"roughly" because $a_{all}$ might not divide $n$ evenly. One solution is to let the $j$-th activity ($j = 0, 1, ..., a-1$) at the `Place` whose id is `p` take care of the values $d + k\delta$, where $k$ runs over $(pa + j) + ha_{all}$ for $h = 0, 1, \ldots$

We'll come back to this sort of loop in gory detail when we dive deeper into X10 arrays.

[14] See file MontePiCluster.x10
[15] See file MontePiCluster.x10

```
1  import x10.util.Random;
2  /**
3   * A parallel version of the Monte Carlo estimate for pi that uses
4   * several Places and several threads at each place.
5   */
6  public class MontePiCluster {
7      /**
8       * At the current Place, spawn some threads, each of which
9       * generates n random points and return the total number
10      * (combining all of the threads results) that fell inside
11      * the circle.
12      * @param pId: this process's id: used to create the seed for the
13      * random number generator.
14      * @param threads: how many threads to use at this Place
15      * @param n: how many points for each thread to generate
16      * @return the total for all the threads of the number of points
17      * that landed inside the circle.
18      */
19     public static def countAtP(pId: Int, threads: Int, n: Long) {
20        var count: Long = 0;
21        finish for (j in 1..threads) {
22           val jj = j;
23           async {
24              val r = new Random(jj*Place.MAX_PLACES + pId);
25              val rand = () => r.nextDouble();
26              val jCount = countPoints(n, rand);
27              atomic count += jCount;
28           }
29        }
30        return count;
31     }
32     /**
33      * Generate n points at random in the unit square, and return
34      * the number that fell within the unit circle.
35      * @param n the number of points to generate
36      * @param rand the function generating the random numbers
37      * @return the number of points that landed in the circle
38      */
39     public static def countPoints(n:Long, rand:()=>Double) {
40        var inCircle: Long = 0;
41        for (j in 1..n) {
42           val x = rand();
43           val y = rand();
44           if (x*x +y*y <= 1.0) inCircle++;
45        }
46        return inCircle;
47     }
```

Listing 1.3: Multi-Place, Multi-Activity Monte Carlo Approximation of $\pi$

```
  /**
   * There are three optional command line arguments: args(0) is the
   * number of points to generate, and args(1) is the number of
   * parallel activities to use, and args(2) is the number of
   * threads to use at each Place.
   */
  public static def main(args: Array[String](1)) {
    val N = args.size > 0 ? Long.parse(args(0)) : 1000000L;
    val places = args.size > 1 ? Int.parse(args(1)) : Place.MAX_PLACES;
    val tPerP = args.size > 2 ? Int.parse(args(2)) : 4;
    val nPerT = N/(places * tPerP);
    val inCircle = new Array[Long](1..places);
    finish for(k in 1..places) {
      val kk = k;
      val pk = Place.place(k-1);
      async inCircle(kk) = at(pk) countAtP(kk, tPerP, nPerT);
    }
    var totalInCircle: Long = 0;
    for(k in 1..places) {
      totalInCircle += inCircle(k);
    }
    val pi = (4.0*totalInCircle)/(nPerT * tPerP * places);
    Console.OUT.println("Our estimate for pi is " + pi);
  }
}
```

Listing 1.4: Multi-Place, Multi-Activity Monte Carlo Approximation of $\pi$, continued

# 2 A First Look At X10's Types

X10's type system is not quite the same as any other object-oriented language's, but it is not really all that different, either. We'll look at some of the main features here. We don't assume that you are familiar with any particular language already, but we do assume that you have had some experience with at least one object-oriented language. We'll make references to Java and C++, since they are so widely taught as a first language, but only to help compare X10 with them.

## 2.1 What's In An `.x10` Source File?

An `.x10` source file comes in three parts:

- first, an optional "package" statement
- then zero or more "import" statements
- and finally, one or more type declarations, at most one of which is declared to be public.

Let's look at each of these parts in detail.

> **Optional package name:** The first line in the file, other than blank lines and comments, can be a "`package`" statement. The syntax is
>
> ```
> package packageName;
> ```
>
> The package name, by convention, is a "`.`"-separated series of lower-case identifiers, *e.g.* `com.ibm.math`. The identfiers can only consist of letters, underscores ("_"), and numbers. It must begin with a letter.
>
> You don't need to have a package spec, and our examples will generally not bother with them, although we give some examples later for those unfamiliar with what is at stake. When no `package` declaration appears, the declaration in the file is said to be "in the default package".
>
> Packages were introduced to allow large projects to organize their classes. They provide firewalls behind which to hide classes that are needed but

should not be generally visible, and they provide a systematic scheme for avoiding name clashes. Before we explain all this, we need the next ingredient in the file, `import` statements.

**Optional Import Statements** After the package statement, if there is one, come the `import` statements:

```
    import fullPackageName.properName;
or
    import fullPackageName.*;
```

The full package name is, as we've said, a sequence of "`.`" separated lower-case names. The final part, "`properName`", should *always* be a capitalized name: for example, "`import x10.util.Random`".

The import statement tells the compiler where to find whatever it is you are importing. "`x10.util.Random`", for example, means that in the file system where `Random.x10` was created, there will be a directory named `x10` with a subdirectory `util` that contains the source file `Random.x10`. Which brings up an important point: how does the compiler know where to look for these directories? What you have supplied is a *relative path* for the class's `.x10` file. Someone has to provide the absolute path that serves as the relative path's starting point. The command-line option `-classpath` does this job:

> If no class path is specified, the compiler will look in the current directory and in the X10 standard library. If it needs to look elsewhere, you need to say so by using
>
> ```
> -classpath path1;path2;...
> ```
>
> in your `x10c` or `x10c++` commenad line. The value of the option uses the same system-dependent syntax that you use for the `PATH` environment variable. We have shown the Unix style here.

We don't want to belabor the point here: X10 is consistent with Java in its use of class paths, and you can find a thorough discussion of that in Wikipedia's article on class paths or any introduction to Java.

The second form of the `import` statement—the one with an asterisk, rather than a class name, as its final part—instructs the compiler to import every class in the package. "`import x10.util.*`." gets you every last class in the `x10.util` package. When to use the catch-all and when it is better to do individual imports is a matter of taste.

You don't *have* to import `x10.util.Random` to use it in your code, but if you don't, you need to refer to everywhere you use it as `x10.util.Random`—you can't just write `Random`.[1]

[1] This should explain why, even though a package spec is a proxy for a file system path, dots and not slashes are used to punctuate it: first, the dot is file-system independent, and second, it is consistent with the notation for selecting a field within an object.

Another situation where you might have to use the fully qualified name is the rare, but not unknown, case of having two classes with the same proper name. Suppose, for instance, we needed to use both X10's class `Random` and another class named `Random` that was supplied by some other library. How do we tell the compiler which one to use for any given call? Use package names in the code at every reference:

```
val ours = new x10.util.Random();
val theirs = new com.foo.bar.Random();
```

When we said the packages help avoid name clashes, it is this sort of example that we had in mind.

**One or more type declarations:** After the optional package and import statements come one or more of three possible types of "type" declarations: either classes, interfaces or structs. We assume you're familiar with classes, and we'll get to the other two in sections 2.3 and 2.4. The overwhelming majority of the time, exactly one appears, but there is no hard and fast rule. One rule does apply: there can be at most one type in the file that is declared `public`, and the name of that type must be the same as the file's name, minus the "`.x10`" file type. Public classes are usable by anyone, anywhere. A type that is declared without that keyword, *e.g.* just "`class X {...}`", is accessible only for use by classes *in exactly the same package.* This is one of the protection features packages provide.

If you want to put several type declarations in one file, make sure that the first has the same name as the source file. Be kind, though: remember that when someone who is reading some X10 code sees a reference to a class `C` and wants to look at the source for that class, they really expect to find `C` in `C.x10` in the appropriate directory, whether it is private to a package or not.

**That's it:** Yes, nothing more, other than comments, appears in an X10 source file.

## 2.2 What's In A Class?

The best way to see what's in a class is an example that says it all. `Ship` in Figure 2.1, on page 30, is it. We work our way through the code line-by-line. First, some terminology:

A **field** of a class holds a piece of data.
A **method** is a piece of code (a function)
The **members** of a class are the fields, methods and other types defined within it.

2

```
1  public class Ship {
2      public static DEFAULT_SIZE = 500;
3      public val name: String;
4      private var passengers: Array[String](1);
5      private var onBoard: Int = 0;
6
7      public def this(name: String) {
8         this(name, DEFAULT_SIZE);
9      }
10     public def this(name:String, initialCapacity:Int) {
11        this.name = name;
12        passengers = new Array[String](1..initialCapacity);
13     }
14
15     static def resize(size:Int, na:Ship) { /*...*/ }
16
17     public def addPassenger(name: String) { /*...*/ }
18     def throwOverboard(name: String, why: String) { /*...*/ }
19     protected def showPassengers() { /*...*/ }
20
21     public static def main(args: Array[String](1)) { /*...*/ }
22
23     static class FareClasses { /*...*/ }
24     protected class Galley {
25         private val platesPerPerson = 4;
26         public def platesRequiredToday() {
27            return Ship.this.onBoard * this.platesPerPerson;
28         }
29         def this(){}
30      }
31     private var galley : Galley = null ;
32     public def galley() = this.galley;
33     public def makeGalley() {
34        this.galley = this.new Galley();
35     }
36
37    }
```

Figure 2.1: A class with lots of stuff.

**line 2:** `public static DEFAULT_SIZE = 500;` A "`static`" field of a class is a constant associated with the class. Its lifetime is the same as that of the class: it is available once the class is available (*i.e.* once the class is "loaded") and stays available until the application terminates. We could have said "`static val DEFAULT_SIZE`", but the "`val`" is implied. The initial value need not be a literal constant, as it is here—it can even be a run-time calculation, but the expression should not involve any non-static methods or data.

There are no variable static members. This may seem a bit harsh to Java and C++ programmers, but there is a good reason. X10, unlike these languages, has been intended from the start to make full use of a multi-processor system. By restricting the data associated with a class to that which is put in place at the time the class is loaded, the X10 runtime is free to copy the loaded class from `Place` to `Place` as needed, with every `Place` always seeing a consistent view of the class.

**line 3:** a public `val` instance member.
An instance field that is a `val` names a constant that must be computed either as part of its declaration or by the constructor. The whole point of an *instance* `val` is that it is a *per instance* constant, not a *per class* one, so the constructor would normally use information from its arguments—the values that make a particular instance what it is—to compute the field's value.

We've shown this instance member as `public`. As with classes, we could use nothing here, just "`val name:String`". The member could then only be used by classes in exactly the same package (which normally would mean *from the same directory*).

There are two other access keywords available: `protected` and `private`. A *protected* member is usable by any class in the same package, and by any class that *extends* the class in which the member is declared. See section 2.2.1, page 33. A *private member* is just that: usable only in the class it is declared. Thus, from most widely available to least:

`public`, `protected`, *no spec at all*, `private`

**lines 4-5:** `var` instance members.
Here are two instance fields that are variables. Both are private for good reason: `onBoard` is the number of valid entries in the `passengers` array, and the last thing we want to do is to allow the general public complete, direct access to either the array or its vital statistics. `onBoard` will be updated every time we add a passenger, or throw one overboard. `passengers` will be reset when we are at capacity and want to add another passenger. We have guaranteed that code responsible is part of *this* class.

**lines 7-13:** Constructors. We show a couple of constructors here. We chose to make both public, but we need not have done so: like any other method, we can restrict the availability of a constructor. We could have made one private and one protected if that is really what we needed, unusual though that would be.

Notice the way that the first constructor invokes the second via `this`. *It is important that this call be the first executable line in the constructor's body.*

**line 15:** `static def resize`

Static methods are "class methods", in the sense that they are properties of the class, not properties of an instance of the class. Static methods have access to all of the static fields and methods of the class, but only an instance of the class may refer to instance fields and methods.

This method has the default access: no access keyword (`public`, `protected`, or `private`) is specified. Default access means that only things in the exact same package have access. Now we didn't specify a package for `Ship`, so it is in the default package. If `DieselShip` is another class and is also in the default package, it can invoke `resize`. The syntax it would have to use is `Ship.resize(...)`. Inside `Ship` itself, you don't need the "`Ship.`" part. `resize(...)` by itself will do. The moral: static methods *really belong to the class in which they are defined.*

**lines 17-19:** instance method declarations with varying access

Here we have three instance methods with varying degrees of protection. We'll allow anyone to add a passenger, because if the boat threatens to sink under the weight, we can always say no. We're a little more careful about who can throw a passenger overboard: this is pretty drastic, and different kinds of ships may have different policies. Finally, respecting its customers' privacy, `Ship` reserves to its package and its subclasses the right to prescribe how instances show who is on board.

**line 21:** `main` again

You've seen `main` enough times that all we feel we have to say here is that you don't need one except in a class that will begin a program's execution.

**line 23:** A static class field is a value associated with the class, like any other, except that it names a class. Within the body of `Ship`, you write "`new FareClasses()`" in order to construct an instance. From a class *not* declared within `Ship`, you would need to write "`new Ship.FareClasses()`". In this example, we opted to give the class the default accessibility: same package can use, others not. We could have chosen any of the other three.

**line 24:** A class that is an instance member.

**Fine Point:** `Galley` is an example of what is often called an "inner class". If you are not familiar with the notion, we'll give a brief discussion of it here. To be honest, though, it is something you are not likely to need very often, at least as an X10 programmer, so you might wish to skip over this.

An instance of an inner class, like `Galley`, depends on an instance of its outer class, here `Ship`, for its very existence. It doesn't make sense to have a galley without a boat, since a galley is defined as "a kitchen in a boat".[3] So, on line 34,

[3] Or an airplane or a camper, as is known to pedants, pilots, vacationers, and other such people of uncertain character.

when we construct a `Galley`, we have to give it a `Ship` to build the galley in. That's why we write the constructor as **`this`**`.`**`new`** `Galley()` rather than simply **`new`** `Galley()`.[4]

Every instance of `Galley` is associated, then, with an instance of `Ship`. The `Galley` can refer to its `Ship`, and to the instance members of its `Ship`. Line 27 shows how this is done. The expression "`Ship.this`" is X10's syntax for this `Galley`'s `Ship`. The total number of plates the `Galley` needs is computed by multiplying the `Galley`'s instance field `platesPerPerson` by the field `onboard` of its ship.

The tight linkage to ships (or whatever the required container is) aside, inner classes are classes like any other.

In our discussion of inheritance hierarchies (see "extra credit:" on page 35), we'll say some more about how instances of inner classes can be used. For very subtle reasons, we can't initialize `galley` when the `Ship` is being constructed. We'll explain this more in §**??**.

We wish we could tell you that this is all there is to a class declaration. It isn't, but it is more than enough to get you going.

### 2.2.1 Inheritance And More

X10 supports simple (single) inheritance: all X10 classes constitute a tree, in which the root is the class `x10.lang.Object`, and a class `Parent` is the parent of a class `Child` when `Child`'s declaration says that it extends `Parent`:

```
class Child extends Parent { ... }
```

For example:
5

```
1  public class DieselShip extends Ship {
2    public val fuelTankCapacity: Double;
3    public var gallonsRemaining: Double;
4
5    public def this(name:String, maxPsgrs:Int, tankSize:Double) {
6       super(name, maxPsgrs);
7       fuelTankCapacity = gallonsRemaining = tankSize;
8    }
9    public def throwOverboard(name: String, why: String) {
10     super.throwOverboard(name, why);
11     /* more stuff here */
12   }
```

[4] Honestly, **`new`** `Galley()` would work too; X10 will use **`this`** automatically, just like it does for method calls.

[5] See file DieselShip.x10

```
13    /* ... more methods can go here */
14 }
```

A `DieselShip` is a `Ship` whose method of propulsion we know. It is in that sense that `DieselShip` is an extension of `Ship`: it is a `Ship` and more. "`DieselShip` *is a direct subclass of* `Ship`" is another common way of saying the same thing, and we'll use both.

> More generally, a class `S` is a *subclass* of another class `C` if there is a chain of classes `S`, $S_1$, ..., $S_n$, `C` each of which is a *direct* subclass of the next.

In declarations in which no `extends` clause appears, like those for the classes `MontePi` or `HelloWorld`, the class extends `x10.lang.Object` by default.

Because it extends `Ship`, an instance of `DieselShip` will have instance fields `name`, `passengers`, and `onBoard`, and methods `resize`, `addPassenger`, and so on. They are said to be *inherited from* `Ship`. The call to `super` in line 6 invokes the super-class `Ship`'s constructor to make sure that the instance of that class on which the `DieselShip` instance is built has these fields correctly initialized.

The call to `super` must be the first executable statement in the constructor, just as shown here. If your code does not call `super` explicitly like this, the compiler will insert a call to `super()`—notice: the call is made with no arguments—so in this situation the compiler must be able to find a constructor for the super-class with no arguments. The good news is that if you provide no constructor at all in your class declaration, the compiler will create this default constructor (needs no arguments, does nothing) for you.

Notice that `DieselShip` provides its own version of the method `throwOverboard`, so rather than inheriting that method, it *overrides it*. The expression `super(name,why)` in the body of the override invokes the parent's version of `throwOverboard`.

> Be careful! While every instance of `DieselShip` inherits the private members of `Ship`, *only the methods already in* `Ship` *have access to them.* Thus, `DieselShip` can affect the array `passengers` only by invoking methods like `addPassenger` that it inherits from `Ship` and has permission to call.

Subclasses are free to override any of the methods they inherit. The only limitation is that the subclass cannot make a method more narrowly available (*e.g.* `protected` rather than `public`) than it is in its parent.

If a method `m` expects an argument that is an instance of a class `Ship`, then it is okay to call `m` with an instance of `DieselShip`. Suppose, for example, we have a class `CruiseLine` whose instances have fleets of ships that sail the world with lots of passengers. A `CruiseLine` instance might want to add a `Ship` to its fleet, so it has a method

```
public def addShip(ship: Ship) { ... }
```

Because a `DieselShip` *is* a `Ship`, we can call `addShip` with a `DieselShip` as the argument.

A similar rule holds for return values. If a method is supposed to return a `Ship`, it can return one that just happens also to be a `DieselShip`.

**Fine Point:** This gets us to the point where we can give an example of how an instance of an inner class might find itself being used outside the class in which it was declared. Subclassing gives us a way to make an otherwise invisible class visible:

- Suppose there is a public class `Kitchen`, and suppose that in our declaration of `Ship` we had written "`protected class Galley extends Kitchen { ... }`".
- Let's add to `Ship` a public method `getGalley()` that returns the private field `galley`.
- If a class has a method whose signature is "`isWellEquipped(k: Kitchen)`", and if `anArk` is a `Ship`, then the call

  ```
  isWellEquipped(anArk.getGalley())
  ```

  is perfectly okay, because a `Galley`, wherever it came from, is a `Kitchen`. In effect, `galley` has "leaked out" to stand on its own in whatever class the method `isWellEquipped` is declared.

Bottom line: you need to be a `Ship` to construct a `Galley`, *but not necessarily just to use one.*

### 2.2.2 Concrete classes versus wishful thinking

Our declaration of `Ship` is an example of wishful thinking, because if you are the least bit realistic about it, a ship without any means of propulsion isn't going to get you very far. On the other hand, there are a large number of very different propulsion systems: we've got `DieselShip` already, and `SteamArk`, `SailArk`, and `RowedArk` are also good possibilities.

So, `Ship`, lacking, as it does, any means of getting anywhere, is, in a very clear sense, not a *concrete* class: you will never see an instance that is not actually an instance of some subclass. Folks just don't build boats that go nowhere, excepting of course for certain tourist-trap seafood restaurants and the occasional glorified liferaft.

But that is not to say that `Ship` can't capture a lot of concrete information–passenger lists, fare schedules, itinerary, and so on–that all ships share, no matter their means of

propulsion. That makes `Ship` a good example of what is called an *abstract* class: a class that one does not intend to instantiate, but which is the ultimate parent of a set of classes that one does.

To convert `Ship` to being an abstract class, one just adds the keyword "`abstract`" to its declaration. We'll call this version `AbstractShip`.
6

```
1 abstract public class AbstractShip {
2     public static DEFAULT_SIZE = 500;
3     public val name: String;
4     // and all the other code as from Ship.x10
```

An abstract class is permitted to declare method signatures that it itself does not implement, but that any of its subclasses that are not themselves abstract *must* implement. Here is an example. Notice that there is no function body: there is just a trailing semicolon. [7]

```
1 public abstract def costPerPassengerMile(): Double;
```

It is clearly unreasonable to talk about cost per mile without knowing how your ship is propelled, and it is equally reasonable to demand that if you want to really function as a ship, you must provide this information.

A concrete class like `DieselShip` will have to implement `costPerPassengerMile`. In `DieselShip`, its declaration will look like:
8

```
1 public def costPerPassengerMile(): Double = 0.018;
```

The keyword `abstract` does *not* appear, because this version of `costPerPassengerMile` is not abstract. The function body appears instead.

## 2.3 Interfaces: More Abstract Than Abstract

While `Ship` may not be as complete as a real ark should be, it can implement a lot of important functionality that all its progeny can use directly: a passenger list is a passenger list, no matter what else is different! There are situations though, where all that one wants to specify is functionality that has no common implementation, but that a variety of classes will implement, each in its own way. This sort of specification is called an *interface*.

[6] See file AbstractShip.x10
[7] See file AbstractShip.x10
[8] See file DieselShip2.x10

Those of you who are coming from a background in C++ will want to work carefully through our discussion of interfaces, because they constitute the closest approximation to multiple inheritance that X10 provides.

A nice example of an interface is `Arithmetic`:[9] [10]

```
1 public interface Arithmetic[T] {
2   def add(t: T): T;
3   def subtract(t: T): T;
4   def multiply(t: T): T;
5   def divide(t: T): T;
6   public static VERSION = "1.1";
7 }
```

In detail:

**line 1:** Like a class, a top-most interface is either public or accessible only in the package of which it is a part. An interface declared within the body of a class is *always* static (it is never per-instance and cannot refer to instance members or methods in the surrounding context), but it can have any access limit: public, protected, default, or private.

The "`[T]`" following `Arithmetic` says that the interface is parameterized by a type `T`. We've seen this before, in the argument `args` for `main()`, whose type is `Array[String]`. The lingo of the trade is that `Arithmetic` is a *generic interface*, and `Array` is a *generic class*. There is nothing sacred about a generic class or interface being parameterized by one type: one can use as many as needed: "`class X[T1, T2, T3,...] {...}`".

**lines 2-6:** An interface may only declare abstract methods, so (unlike the story with abstract classes), you don't need the keyword `abstract`. What line 2 says is that

> if a class `A` implements the interface `Arithmetic[B]`,
> if `a` is an `A`, and `b` is a `B`,
> then the expression `a.add(b)` produces a result that is a `B`.

In real life, we might have two classes `Monomial` and `Polynomial` to implement polynomials with `Double` coefficients. It would be natural, then, for `Monomial` to implement `Arithmetic[Polynomial]`: when you add a monomial to a polynomial, and the result is a polynomial.

When you add two or multiply `Polynomials`, you get another `Polynomial`. So, we want `Polynomial` to implement `Arithmetic[Polynomial]`. This isn't a weird recursive thing, though it's got the imposing-sounding name of "F-bounded quantification" (which most programmers should remember purely for the sake of

[9] For extra slickness, we could have programmed it so that the operations use the usual symbols: `a*b` instead of `a.multiply(b)`. See §**??** for how to do this.

[10] See file Arithmetic.x10

sounding erudite at job interviews). It just says that polynomials can perform arithmetic with polynomials and the results will be polynomials – which is just what you'd expect.

X10 has a few standard interfaces that are often used in this F-bounded way. For example, there's `Comparable[T]`. A class implementing `Comparable[T]` can be compared to a value of type `T`. Most of the time, you want to compare things of the *same* type. So, classes tend to implement `Comparable[`themselves`]`.

Another important point is that the methods declared by an interface are always `public`. So even though the keyword `public` does not appear here, it is implied. We'll say more about this when we discuss how you use interfaces.

**line 6:** Interfaces may also declare static constants. That's the only kind of constant they can declare – `val` means **`public static`** `val` in an interface, just the way that `def` means **`public`** `def`.

In fact, some interfaces do nothing but declare constants: [11]

```
1 public interface MathConst {
2   val PI = 3.14159265358979323846264433;
3   val E = 2.71828182845904523536028874;
4   val GAMMA = 0.57721566490153286060651;
5   val SOFA = 2.219531668871;
6   val DOUBLE_DIGITS = 15.95;
7   val DOUBLE_E_MAX = 307.95;
8 }
```

Some of these, like `PI` and `E`, are useful mathematical constants that will be the same for all X10 programs. Others, like `DOUBLE_DIGITS` and `DOUBLE_E_MAX`, might change from computer to computer: these are the right values for IEEE double-precision numbers, but they would be different on a machine that used quadruple precision for the `Double` type.

Another common use is to give names to the legal values for the arguments of one or more of the methods. Here are some names for some standard colors: [12]

```
1 interface Color {
2   val RED = 0xFF0000;
3   val BLUE = 0x0000FF;
4   val PEACOCK = 0x33A1C9;
5 }
```

We can refer to these as `Color.RED` and so on:

```
13
```

[11] See file MathConst.x10
[12] See file Color.x10
[13] See file Discoloration.x10

```
1 class Discoloration {
2   static def example() {
3     Console.OUT.println("RED = " + Color.RED);
4   }
5 }
```

If we're inside of a class that implements `Color`, we can leave off the `Color.` and refer to `Color.RED` as just plain `RED`:

14

```
1 public class Colorizer implements Color {
2    static def example() {
3       Console.OUT.println("Red = " + RED);
4    }
5 }
```

A class implementing `Color` has to use `Color`'s value for `RED`. Would you want someone to implement the interface with its own value for `RED`? That would ruin code reuse.

Finally, one interface can extend another. They can do so better than classes can, even: one interface can extend several other interfaces, but a class can only extend a single other class.

15

```
1 interface Colored {
2   def color(): Int;
3 }
4 interface Shape{}
5 interface Shaped {
6   def shape():Shape;
7 }
8 interface ColorForm extends Colored, Shaped {}
```

One could say "`ColorForm` is a sub-interface of `Colored` and `Shaped`", though some people think that "`ColorForm` *refines* `Colored` and `Shaped`" sounds better.

[14] See file Colorizer.x10
[15] See file ExtendingInterfaces.x10

### 2.3.1 How interfaces get used

Let's implement a class of fractions, `Frac`. Fractions are useful because you can express some common values exactly. $\frac{1}{3}$ is a fraction that is exactly one-third. The closest that X10 can come to this value is 0.3333333333333333, which is pretty close, but off by more than $3 \times 10^{-}16$. If you're not careful, these little errors can really mess up your computation; the discipline of scientific computing teaches how to minimize the problem. Using fractions, which have exactly the right value with no round-off error, is one way.

The syntax

```
class Frac implements Arithmetic[Frac] { ... }
```

asserts that the class `Frac` implements all of the methods declared in the `Arithmetic[Frac]` interface–with exactly the argument types and return types shown there. In plain English, this is saying that you can add, subtract, multiple and divide one `Frac` by another and get an `Frac` result.

While `Frac` is one candidate for the class that appears as the `T` in the `Arithmetic[T]`, we'd probably also like to add `Frac`s to other kinds of numbers – at least `Double`s, and if we were doing a full-strength version we'd have `Long`s, `Int`s, and perhaps other types as well.

16

```
1 class Frac implements Arithmetic[Frac],
2                       Arithmetic[Double]
3 {
```

At first blush, this is a daunting declaration: think how many methods must be implemented, especially for the library version! But the truth is that none of these methods is all that involved, so it is just a matter of some (necessary!) drudgery. Here the end more than justifies the means.

The implementation is straightforward. The constructor for `Frac`s always puts fractions in lowest terms, so that `new Frac(10,20)` is the same as `new Frac(1,2)`:

17

```
1 public def this(var n:Int, var d:Int) {
2   if (d < 0) { n = -n; d = -d; }
3   if (d == 0) throw new Exception("Division by zero!");
4   var gcd : Int = gcd(n,d);
5   this.n = n / gcd;
```

[16] See file Frac.x10
[17] See file Frac.x10

```
6   this.d = d / gcd;
7 }
```

Fraction arithmetic uses the familiar rules. Since the constructor reduces everything to lowest terms, we don't need to do any extra work to do so:

18

```
1 public def add(f:Frac) = new Frac(n * f.d + f.n * d, d * f.d);
2 public def negate() = new Frac(-n, d);
3 public def subtract(f:Frac) = this.add(f.negate());
4 public def multiply(f:Frac) = new Frac(n * f.n, d * f.d);
5 public def divide(f:Frac) = new Frac(n * f.d, d * f.n);
```

The `Frac`-to-`Double` versions are easily implemented using a method to convert a fraction to a `Double`:

19

```
1 public def asDouble():Double = (n as Double) /(d as Double);
2 public def add(dbl: Double) = this.asDouble() + dbl;
3 public def subtract(dbl: Double) = this.asDouble() - dbl;
4 public def multiply(dbl: Double) = this.asDouble() * dbl;
5 public def divide( dbl: Double) = this.asDouble() / dbl;
```

Notice that in this class, the method `add` is explicitly declared to be `public`. While `public` is all there is in an interface declaration, it is *not* all there is in a class declaration, so you must be specific here. The compiler will complain if you do not.

Now suppose that somewhere there is a class that has a method "`doIt(x: Arithmetic[Frac])`". Since `Frac` implements `Arithmetic[Frac]`, we can use a `Frac` as the argument to `doIt`, *e.g.*:

20

```
1 doIt(new Frac(1,3));
```

We end this section with some "side bars" that somehow didn't quite fit in the flow of the rest of this section, but are worth a moment's thought.

[18] See file Frac.x10
[19] See file Frac.x10
[20] See file Frac.x10

**Anything you can do...**

The 1946 Broadway musical *Annie Get Your Gun* has a wonderful song that begins "Anything you can do, I can do better. I can do anything better than you." X10 has an operator that says the same thing much more briefly: if `T` and `U` are two types, then the expression `T<:U` is a boolean that, if true, means that a value of type `T` may be used anywhere a value of type `U` may be, or in other words, that `T` is either `U` itself or a *subtype* of `U`.

**(B: This isn't a good explanation of <:, and I think that the explanation goes elsewhere and should be used a lot more. :B)** One use of the "operator" `<:` is in `val` declarations. Saying "`val t <: T`" declares `t`'s type to be a subtype of `T`. That way, if `t` is initialized in one of several arms of an `if` statement, differing types may appear in the various assignments, so long as all are subtypes of `T`. Also users of `t` should only count on getting a `T`.

For example, if we have an interface like `Caloried`, which tells how many calories something has: [21]

```
1 interface Caloried {
2   def calories():Int;
3 }
```

We can write a class which manages a list of ingredients, and can count their calories. Note that this wouldn't even compile without the constraint on `T`. It is only that constraint that gives us permission to call the `calories()` method in line 7. [22]

```
1  class Recipe[T]{T <: Caloried} {
2    val ingredients: List[T] = new ArrayList[T]();
3    public def add(t:T) { ingredients.add(t); }
4    public def totalCals() {
5       var s : Int = 0;
6       for(ingredient in ingredients)
7           s += ingredient.calories();
8       return s;
9    }
10 }
```

A class, like `Flour`, can implement `Caloried` in the ordinary way: [23]

```
1 class Flour implements Caloried {
2   private val name: String, cal:Int ;
3   public def this(name:String, cal:Int)
```

[21] See file NeedsNutrients.x10
[22] See file NeedsNutrients.x10
[23] See file NeedsNutrients.x10

```
4     { this.name = name; this.cal = cal; }
5   public def calories() = this.cal;
6 }
```

Then, we can use `Flour` in a `Recipe`:

24

```
1 static def example(): void = {
2   val flours <: Recipe[Flour] = new Recipe[Flour]();
3   flours.add(new Flour("1 cup bread flour", 495));
4   flours.add(new Flour("1 cup whole-wheat", 520));
5   flours.add(new Flour("1 cup rye", 360));
6   assert flours.totalCals() == 1375;
7 }
```

(This is different from writing a class which simply manages a list of `Caloried` values. The difference doesn't matter much in this example, but it would matter if the constraint were `T <: Arithmetic[T]` – in which case we couldn't even talk about a `List[Arithmetic]`, since X10 insists on knowing what type of thing the arithmetic is being done on.)

**Why interface methods are public:**

Suppose a class `A` implements an interface `I`, and suppose that `B` is a subclass of `A`. Because an instance of `B` might appear anywhere an instance of `A` is expected, we see that `B` *must also implement* `I`. This is one reason why the methods in an interface must be declared `public`.

There is another reason that is more pragmatic than principled. The point of an interface is to tell the outside world what some object is capable of doing. A `private` method, on the other hand, is just a *hidden* means to accomplish a class's goals. Why should anyone care whether a class implements a method that can only be called from within that class?

An interface itself may declared with any access limit that makes sense in the context: if the interface is being declared inside a class (and it definitely can be), then protected, default and even private access make sense. For example, a private interface might describe the type of an argument of one or more of the class's private methods, and there might be several static or inner classes declared inside the class that implement that interface: not a common situation, but not unreasonable.

[24] See file NeedsNutrients.x10

**A matter of convention:**

There is a sort of convention that interface names should begin with "I", Suppose for example, that we had used `INumber`, instead of `Arithmetic`, for our interface name. We could then have declared a class named `Number` that implements it. A matter of taste of course, but not an unreasonable convention, particularly in situations where there is a "standard" or "canonical" implementation of the interface. A variation on this theme is to append "`Impl`" to the name of an interface to create the name of a concrete class that implements it: *e.g.* interface `XmlParser` is implemented by `XmlParserImpl`. You will see both conventions widely used.

**For C++ programmers:**

We said that interfaces provide a sort of "multiple inheritance." What we meant is that a class may declare that it implements any number of interfaces. That is certainly reasonable: all you are saying is that the class provides several named sets of methods. It is, of course, a weaker notion than multiple inheritance, because, when a class implements an interface, no instance members or methods are actually inherited: all of the code is in the class, or is in one of its ancestors up the inheritance chain.

## 2.4 Giving Up Inheritance To Get Efficiency

### 2.4.1 The Problem

Supporting inheritance does not come without costs when a program runs. Consider the class `DblDbl`, a model of innocence:
25

```
1 public class DblDbl {
2    public val x: Double;
3    public val y: Double;
4    public def this(a: Double, b: Double) {
5       this.x = a; this.y = b;
6    }
7    public def normSq() = x*x + y*y;
8 }
```

Let "`public def doIt(p:DblDbl) {...}`" be a method declared in some other class, and suppose that in the body of `doIt` we find the call `p.normSq()`. What does this call really cost? The problem is that the value passed to `doIt` for the argument `p` might really be a reference to an instance of some proper subclass of `DblDbl`. We

[25] See file DblDbl_Class.x10

only know the precise class of `p` once we start executing the body of `doIt`. Because `p`'s class can have its own implementation of `normSq`, making sure we are calling the correct `normSq` means looking up its address in `p`'s class's descriptor *each time the call is executed*.

The moral is that an instance `obj` of a class that may have subclasses must carry with it a reference to the actual class of the instance `obj`. Applying this to instances of `DblDbl`, we see that in addition to the 16 bytes required to store the two instance fields, we have to keep at least 4 bytes for the reference that allows us to find its class. If there are only a couple of hundred `DblDbl` instances around, and if we only call `normSq` a couple of thousand times in any given run of the code, who cares? But what if we had an array a million `DblDbls` and were traversing it repeatedly calling methods that some subclass might override?

Think about it. The million `DblDbls` have 16 megabytes of data that we're really interested in, but are costing us at least another 4 megabytes or more for type information. A million calls to `normSq`, moreover, require a million method lookups that we'd really love to avoid. Now, if there really are subclasses of `DblDbl`, and they really need their own implementation of `normSq`, then fine: we'll gladly pay the cost in time and space, because we are getting something in return. But if not?

These considerations are one reason why a performance-oriented language avoids using the normal class hierarchy for primitive types like 32-bit integers: no language that expects high performance can allow a 4-byte integer to consume 8 bytes or more and to be accessed indirectly.

There are two different approaches one can take, as a language designer, to solving this problem. One is to keep the primitive value types to a handful close to the physical reality of the CPU and treat them specially in the language itself. This is what Java chose to do. The other approach is to keep the implementations of the primitive types as close to that of classes as possible, which is what X10's designers did.

The question X10 had to face is: how much of what classes give us can we keep without incurring any substantial overhead? There are two sets of issues here: serial issues like "don't waste space", and "don't waste time looking stuff up", and distribution issues like "don't create the possibility of inconsistencies between what one processor thinks the value of the "thing" is and what another thinks, so that communication between the processors is required to maintain storage integrity".

### 2.4.2 The Solution

To begin with, we have already seen that we need instances whose types are exactly what they are declared to be: no inheritance. To achieve the storage integrity, all of the instance fields must be `vals`, so that an instance once constructed is immutable and hence can be copied freely whereever its value is useful. X10 calls these things "`structs`", to distinguish them from classes. [26] X10 made one other critical decision:

[26] They're not C++-style structs. They're X10-style structs, which are pretty different from C++-style structs.

when an instance of a struct appears as a field or is assigned to an identifier, it is the data in the instance fields that is stored, and not a reference to that data. For primitives like `Int`, this is a critical efficiency consideration: what's stored for an `Int` its value, not to the address of its value, so you don't have "follow a pointer" to get to the value.

Struct declarations are essentially the same as `class` declarations. For example, we can change the class `DblDbl` to be a struct just by replacing the keyword "`class`" with "`struct`" in its declaration:[27]

[28]

```
1 public struct DblDbl {
2   public val x: Double;
3   public val y: Double;
4   public def this(a: Double, b: Double) {
5     this.x = a; this.y = b;
6   }
7   public def normSq() = x*x + y*y;
8 }
```

A few comments:

**line 1:** The accessibility rules for a struct are the same as for a class. We chose "`public`" for the example, but default, `protected` and `private` access may be assigned as appropriate.

Structs may be generic—that is, they may, like `Array`, have type parameters—although this one does not.

**lines 2 and 3:** As we noted earlier, the instance members of a struct are always `val`s. For that reason, you don't need to say "`val`" as we did here. It is okay if you do. If you are not sure whether in the long run you are going to want this type to be a struct or a class, throwing in the `val` makes it easier to go back and forth.

We've only shown instance members in this example, but structs can have static member fields as well. Indeed, one can also define classes, interfaces, and structs in the body of a struct, just as one can in a class.

**line 4:** Unlike their C namesakes, X10 structs can declare methods, both instance methods and static methods. We've shown a public method here, but both private and default access can be used. Structs *never* have subclasses, so "protected," which just broadens the default access by allowing access to subclasses, is useless for structs.

[27] You can't always turn a class to a struct this easily – or at all. For one difference, structs don't support inheritance, so, if your class uses inheritance, you've got some recoding to do.

[28] See file DblDbl_Struct.x10

The keyword `new` is *not* required when creating a new instance of a struct—in contrast to the case for a class. Here is an example that assigns a `DblDbl` value: [29]

```
1 var p: DblDbl = DblDbl(1.2, 3.4);
```

The "`new`" is gone from the right-hand side. If you put the "`new`" in, though, the compiler will happily accept it.

### 2.4.3 Equality, Classes, and Structs

One important difference between struct and class implementation is the treatment of "==". If `s1` and `s2` are two instances of the same struct, then "`s1 == s2`" is true exactly when when their instance fields have identical values. If, for example, `s1` and `s2` are `DblDbls`, then `s1==s2` is equivalent to `(s1.x==s2.x) & (s1.y==s2.y)`. Remember that for instances of a class, "==" tests for references to same instance, not equality of values. But: there are *no* references to instances of a struct, just the instance data itself. Since the data is all there is, that's all you operate on—with "==" or anything else.

Here's an example: `Class` and `Struct` are identical except that `Class` is a class and `Struct` is a struct. Two `Classes` with the same data are not == to each other, but two `Structs` with the same data are ==. [30]

```
1  public class ClassVsStruct {
2    static class Class(a:Int) {}
3    static struct Struct(a:Int) {}
4    public static def main(argv:Array[String](1)) {
5      val c <: Class = new Class(1);
6      val d <: Class = new Class(1);
7      val s <: Struct = Struct(1);
8      val t <: Struct = Struct(1);
9      assert c != d;
10     assert s == t;
11   }
12 }
```

### 2.4.4 Fine Points

We close this section with some fine points about structs that are interesting and certainly part of the expert's toolkit, but are not critical for a first reading.

[29] See file DblDbl_Struct.x10
[30] See file ClassVsStruct.x10

**Fine Point: what all structs and classes share.** Both structs and classes can implement an interface, and there is one interface, called `x10.lang.Any` that is the minimal interface that all structs and classes must implement. As you would guess, it is not particularly rich:

```
public interface Any {
   def toString():String;
   def typeName():String;
   def equals(that:Any):Boolean;
   def hashCode():Int;
}
```

The first three are the ones that most programmers care about. The purpose of the fourth method, the hash code, is to give a compact way of identifying an object.

The method `toString` returns a representation of an instance as a `String`. Because `toString` is always available, conversion to `String` is possible in any context where a `String` is known by the compiler to be required.

The method `typeName` returns the precise name of the type as it is understood by the X10 compiler. It is most useful as a learning aid and debugging tool. If ever you are in doubt about what the compiler thinks the type of some object "`e`" is, the call

```
Console.OUT.println(e.typeName());
```

will tell you. Try a few, for instance: `3.typeName()`.

The method `equals` tests whether its invoker and its argument "have the same value." It should be thought of as a user-overridable version of the comparison operator "==". Therefore, `a.equals(b)` is the same as `a==b` by default. Remember, though, that, as we just remarked above, "==" has a different meaning for class instances than for struct instances.

Many classes override `equals`. A familiar example is the `String` class: if `a` and `b` are `Strings`, then `a.equals(b)` is `true` when the contents of their underlying byte arrays are the same.

`hashCode` is related to `equals`. If two values are `equals`, then their `hashCodes` ought to be equal, too. Some standard data structures rely on this fact, especially `HashMaps` – the hash codes that the `HashMaps` use are `hashCode` values.

If you aren't careful about this, you can write a program that looks like it puts a value in a `HashMap`, but it can't be found once it's there. **(B: Give this example, and describe the right way to do it :B)**

**Fine Point: structs really are values.** The rules for what can be part of a struct are designed so that the compiler can determine the size of an instance from the declaration. The reason is storage efficiency: once the size is known, no more space need be allocated than is needed to store the value in memory, and at runtime, we don't have to carry around size information for every instance.

One consequence of the storage rules is that you cannot define the member fields of a struct in terms of the same struct or collections of that struct. If, for instance we try to declare a struct `DblDblListItem`

```
public struct DblDblListItem {
   public p: DblDbl; // the point
   public next: DblDblListItem; // !!! no: illegal
}
```

so that we can form lists of `DblDbl`s, the compiler will complain. To understand why, ask yourself, "How big is a `DblDblListItem`?" Suppose that it were $n$ bytes.

A `DblDbl` itself is (say) 16 bytes. So adding the sizes of `p` and `next`, we get a total of $n + 16$ bytes. But that has to be the same as the size of the whole `DblDblListItem` in which these are the instance fields. That whole value is only $n$ bytes, because it's a `DblDblListItem` and that's how big a `DblDblListItem` is. So, $n + 16 = n$. Something is wrong with this.

The point is that we really wanted `next` to be a reference to the next item: an address, and not the item itself—and if `DblDblListItem` were a class rather than a struct, that is exactly what `next` would be: a reference. C++ programmers would use a pointer in this sort of situation, a physical address: `DblDblListItem *next`. Its size, like the size of any object reference, is some small number of bytes that is independent of what its target is. But in X10, there is no such thing as reference to a struct, nor can you take its "address." A struct is a value. Period!

The short answer, then, to "How *does* one design an X10 struct that is a linked list?" is that *one doesn't*. Linked lists need pointers, and structs don't have pointers. Linked lists are done using objects in X10, not structs.

[31]

### 2.4.5 Performance of Structs

Since X10's primitive types, `Int` and `Double` and so forth, are largely defined by standard mechanisms of the language, you can define your own types that are nearly as efficient ... sometimes. At the very least, you can avoid the biggest inefficiencies of objects by using structs. But structs have a few slow spots of their own, and if you're not careful, you can lose everything you gained.

Consider the assignment:

```
var x: T;
var y: T;
...
```

[31] We wrote out some code for you to show that if you are really desperate, you *can* do it, but be warned, this is very, very bad X10. Do not use this as a model for *anything* The use of the `Any` interface there is particularly ill-advised. See list/DblDblList.x10.

```
x = y;
```

When `T` is a class, the assignment copies a reference that is `y`'s current value, to `x`. This takes a fixed amount of time, independent of how big the class `T` is.

When `T` is a struct, however, what gets copied is the entire current value of `y`, not a pointer to the value. To the extent that the size of the value is greater than the size of a reference, copying the value is more expensive than the copying the reference. Our `DblDbl` is a good example: its size is 16 bytes, and a reference could use as few as 4 bytes.

The same thing comes up when passing parameters:

```
public def doIt(t: T) { ... }
var x: T;
doIt(x);
```

When `T` is a class, the value passed to `doIt` is the reference. If `T` is a struct, the whole value of `x` has to be copied.

To summarize: the benefits of structs are:

- no cost for memory allocation,
- no cost for garbage collection,
- faster method invocation, and
- better storage utilization for large aggregates.

and the costs are:

- assignments and argument passing may be more expensive, and
- you lose the benefits of the class hierarchy for organizing common functionality.

The situation is further complicated by the tricks code optimization can use to minimize the assignment costs, so even knowing that there might be a problem, it may be difficult to tease out the cost, if any, short of running the code to see. In addition, when you are working with classes, you can avoid some of the cost of method invocation by declaring a class to be "`final`"—*e.g.* "`public final class DblDblListItem`". Final classes cannot be subclassed, which implies that their methods cannot be overridden, and this, in turn, allows the compiler to reduce the cost of method invocation.

# 3 A Potpourri of Types

We have already mentioned a few of the basic types that X10 provides in its standard library. We will look at some more of them here. Our discussion will be far from complete, but it will at least get you started and aware of where to look as you need more. A complete API for the library can be found in http://dist.codehaus.org/x10/xdoc/.

The library is very much a work in progress. For that reason, some of this chapter may become out-of-date reasonably quickly, mostly, if not entirely, because X10's resources will rapidly become richer. We will try to keep this guide in step as best we can.

## 3.1 The Numbers Game

### 3.1.1 The Players

X10 provides the usual sorts of basic numeric types:

**Byte, Short, Int, Long:** 8-bit, 16-bit, 32-bit, and 64-bit signed integers.

**UByte, UShort, UInt, ULong:** 8-bit, 16-bit, 32-bit, and 64-bit *unsigned* integers.

**Float, Double:** single and double precision IEEE floating point.

It also supplies a double-precision `Complex` type that Fortran and C++ programmers have come to expect. All of these types are in the package `x10.lang`.

In terms of what these types mean and how they get used, X10 is consistent with what other languages do, so we won't go into a lot of detail here. In particular, implicit conversions will be performed as needed. For the signed numeric types, the conversions are between a type and any type to its right:

```
Byte < Short < Int < Long < Float < Double
```

And similarly, for unsigned types:

```
UByte < UShort < UInt < ULong < Float < Double
```

Let us be a little more clear about our vocabulary here:

> **Coercions:** Suppose we declare a variable "`i`" to be an `Int`: `var i: Int;` The expression "`i as Long`" is X10's syntax for saying "convert `i` to be a `Long`." In one sense, this conversion involves no substantive change to `i`'s value as an integer. On the other hand, when we realize `i as Long` in memory, or in CPU register, the full 8 byte representation of a long integer will be used. This sort of conversion—in form, but not in substance—is usually called a *coercion*.
>
> **Casts:** Another sort of conversion is that from `Int` to `Byte`. The value of the expression "`i as Byte`" is the result of masking out the leading 24 bits of `i` and regarding what is left as a `Byte`, meaning an integer in the range from -128 to 127. If, for example, `i` is 4095, which in hex is `0xfff`, then masking out the leading bits leaves us with `0xff`, so `4095 as Byte == -1 as Byte`.
>
> A conversion like that from `Int` to `Byte`, or more dramatically, from `Long` to `Double` that may involve more than a change to the physical representation of the value is called a *cast*.
>
> The example of "`4095 as Byte == -1 as Byte`" shows that casting can be dangerous. Here, at least, there is nothing subtle going on, so if that wasn't the programmer's real intention, there is some chance that the problem can be spotted. One may not always be so lucky. For example, casting `Int` to `Float` can lose precision for the same reason that `Int` to `Byte` can, because `Float` only has 23 bits (plus a sign bit) to hold the value. It uses the remaining 8 bits in its 32 bit word for the exponent. So casting `23456789 as Float` yields `2.3456788E7`. Note carefully the last digit. If that last digit had some significance, you are in trouble—and the problem is going to be *very* hard to see.

Thus, some of the implicit conversions above, like `Int` to `Long` are really coercions, while others, like `Long` to `Float`, are really casts. Where a cast is what is at stake, X10 usually forces you to be explicit: "`Int i = someValue(); Byte b = i as Byte;`". The explicit cast "`as Byte`" says you are prepared for the consequences of the conversion.

### 3.1.2 Signed versus unsigned

This section is largely an aside for Java programmers, because the distinction between "signed" and "unsigned" integers does not exist in Java and so may not be familiar. What is at stake?

From the dawn of computer history, CPUs have supported two kinds of arithmetic instructions: one set in which one of the bits that make up the integer's value is used to hold its sign, and a second set in which all integer values are treated as non-negative. Having the extra bit as part of the value doubles the size of the largest positive integer

you can express–something that was pretty important when the normal machine "word" was 16 bits wide. Today, with 64 bit words becoming the norm, the extra bit is not all that big a deal: $2^{63} - 1$ is plenty big—how often will you really need $2^{64} - 1$?

When the Java standard was first developed, the designers felt that, in their experience, exposing both types, as languages like C had done, had been a source of more errors than the additional power warranted. Thus, unsigned values never made it into Java. Well, they made it back into X10. There *are* some arguments for having it, aside from doubling the largest value, not the least:

When you declare something to be "unsigned", you are telling the reader of your code something: negative values are neither expected nor welcome, as, for instance, if you are tracking the number of elements in a list. It is nice to have code that really says what you really mean.

Another argument is that if you want to use a numeric type to represent one of the 8-bit ISO character types–*e.g.* ISO-8859-1, which handles European languages nearly completely–then `UByte` is a better match than `Byte`. Certainly, everyone *thinks* of the mapping from numbers to characters as being a map from the integers 0, ..., 255 to characters. Also with `UByte`, you can test for control characters with one inequality: they are precisely those whose value as a `UByte` is less than `0x20`. Similar considerations apply to multi-byte encodings of characters.

## 3.2 String Theory

X10 support for character-based applications is limited at the moment, but with enough implemented to provide a base for applications that are not fundamentally about manipulating character data. Internationalization, in particular, has only very limited support, and regular expressions have not yet been implemented. What has been implemented will, with a few exceptions, not surprise anyone coming from C++ or Java. The same can be said for file input and output: the implementation is limited, get richer, and not surprising.

The basic types for character work are `x10.lang.Char`, `x10.lang.String`, and `x10.util.StringBuilder`. Since `StringBuilder` is in the package `x10.util`, it must be imported explicitly; the other two need not be.

### 3.2.1 `Char`

An instance is a 16-bit Unicode character. [1] Like `Byte` and `Int`, `Char` is a struct, but it is *not* a numeric type: there is *no* implicit conversion to any numeric type. The only explicit conversion to a numeric type is the method `ord` that returns an `Int`:

[1] Strictly speaking, this is true, but in practice, support for multi-byte character sets in both `Char` and `String` is very limited. For the moment, you would be best advised to stay with 7-bit ASCII or 8-bit ISO character sets.

```
'a'.ord() == 0x61
```

will evaluate to `true`. You *can* cast integer values to characters

```
i: Int = ...;
c: Char = i as Char;
```

does what you expect: produces the `Char` whose "`ord`" is `i`.

Programmers used to avoiding calls to `isdigit` by writing `'0' <=c && c <= '9'` will find that the compiler complains that you cannot compare integers and characters. (This will probably be fixed in an upcoming release of X10). For now, when you are writing X10, call `isDigit`, `isLowerCase`, `isSpaceChar`, and so on. All of these utilities are identical in function with their counterparts that have been around since the birth of C.

A 16-bit Unicode character can be written as an escape sequence \u*nnnn*, in which each of the `n`'s is a hex digit. As usual, the following escape sequences may be used:

\b == \u0008: backspace BS
\t == \u0009: horizontal tab HT
\n == \u000a: linefeed LF
\f == \u000c: form feed FF
\r == \u000d: carriage return CR
\" == \u0022: double quote "
\' == \u0027: single quote '
\\ == \u005c: backslash

### 3.2.2 `String`

A `String` is an *immutable* array of `Char`s. Immutability means that, for example, once you declare

```
s: String = "hi";
```

you can, if you wish, access the first character in `s` as `s(0)`, but you *cannot* reassign via "`s(0) = 'o';`".

String literals are delimited by "double quotes," as in `"begin,end"`. The same escapes apply within `String` literals as for `Char` literals.

Any object, be it from a class or struct, is converted to a `String` by calling its method `toString()`, and there is always a default implemention that is provided and does something sensible. You are free, of course, to provide your own implementation in the classes and structs you create. For example:

2

[2] See file Cast.x10

```
1 public class Cast {
2     public static def main(args: Array[String](1)): Void {
3        Console.OUT.println("I am a "+ new Cast());
4     }
5     public def toString() { return "Cast of Thousands"; }
6 }
```

In line 3, the effect of the "+" operation is to concatenate the two `Strings`. When the code is run, the console output is:

```
%x10 Cast
I am a Cast of Thousands
```

There are some basic methods already implemented for `String`s, and over time more will be added. There are four methods for extracting substrings:

`substring(from:Int):` The argument `from` is the index of the first character to keep. For example, , if we have `val s = "misguided"`, then `s.substring(3)` will be `"guided"`.

`substring(from:Int, to:Int):` The first argument is again the starting index, and the second, `to`, is the index 1 greater than where you wish to stop. Thus if `val s = "misguided"` as before, then `s.substring(3, length(s)-1)` will be g̈uide"—and not (as one might think) `"guided"`.

`trim()` Remove leading and trailing white-space: `" abc ".trim()` is `"abc"`). In fact, it trims all ASCII control characters as well. [3]

`split(s: String)` If you need to break up a comma-separated list, or something like that, use the `split()` method:

```
val s = "a,b, c, d";
val parts = s.split(",");
```

yields an array of four substrings: `"a"`, `"b"`, `" c"` and `" d"`. The argument for `split` in the C++ runtime is currently a `String`, but in the Java runtime, it is a regular expression. Eventually, both will accept a regular expression, which will make it easier to clean up the white-space trailing the commas at the same time you get rid of the commas.

[3] To be absolutely precise:

If an empty string, or one whose the first and last characters are represented by characters whose codes greater than `0x20`, which is the ASCII code for the space character, then to the same string is returned.

If the string has no character with code greater than `0x20`, then an empty string is returned.

Otherwise, if `s` is the string, if `start` is the index of the first character in `s` whose code is greater than`0x20`, and if `end` is the index of the last character in `s` whose code is greater than`0x20`, then the returned string is `s.substring(start, end+1)`.

There are also four approaches to comparing `String`s. Let `s` and `t` be two `String`s:

**Equality as objects:** `s == t` evaluates to true if `s` and `t` name the same `String` object. `String` is a class, not a struct, *so what* `s` *names is a reference to a* `String`, not a literal string value.

`s != t` is true when `s` and `t` do *not* name the same `String` object.

**Equality as values:** The expression `s.equals(t)` evaluates to true if the current values of `s` and `t` are the same—in other words, if the two underlying byte arrays are byte-for-byte the same. If you don't care about case, use `s.equalsIgnoreCase(t)` instead.

**Infix comparison operators:** The operators <, <=, >=, and > compare two `String` values using lexicographic (dictionary) order. For example, `"good"<"goof"` will evaluate to `true`, since `d` comes before `f`.

**Comparison methods:** The method `compareTo(s:String):Int` returns a signed integer that is *negative* if `s < t`, 0 if `s.equals(t)`, and *positive* if `s > t`. The method `compareToIgnoreCase(s:String):Int` can be used for comparisons that are *not* case-sensitive. The notion of "order" is the same as for the infix operators.

Searching `String`s can be done with the methods

`s.indexOf(what,where)`, `s.lastIndexOf(what, where)`: returns the indices of the first and last occurrence of the `String` or `Char` "`what`" in `s`, if it is present, -1 otherwise. The second argument, `where`, is optional. It is an integer and specifies the index of position in `s` at which to begin the search.

`s.startsWith(what)`, `s.endsWith(what)`: returns true or false, true meaning that the `String` "`what`" appears at the beginning or end of `s`.

For those with fond memories of C's `printf` or Java's `format`, X10 gives you

```
format(fmt:String, args:Array[Any]):String
```

whose first argument is a format string like those of its C and Java ancestors, and whose second argument is the array of whatever is to be formatted.[4] For example, executing

```
s = String.format("%s %d %s %10.2f", ["int",32,"dbl",3.2]);
```

leaves `s` with the value `"int 32 dbl       3.20"`.

Support for the formatting options is limited at the moment, but, as in our example, most simple cases work as they have done since `printf` was introduced some forty years ago.

[4] The type `Any`, as we discussed in section 2.4.4, page 48, is the interface all values have in common.

```
1 public static def main(args: Array[String](1)) {
2   val inputPath = args(0);
3   val outputPath = args(1);
4   val I = new File(inputPath);
5   val O = new File(outputPath);
6   val P = O.printer();
7   for (line in I.lines()) {
8     P.print(line);
9   }
10  P.flush();
11 }
```

Listing 3.1: A simple line-reading file copier

### 3.2.3 StringBuilder

One often has to build up a string whose length is not known in advance and which isn't short enough that it makes sense to just build it up as a concatenation `a+b+c+...` That is the job of `x10.util.StringBuilder`. We remind you that you have to import this class, like all others in the `x10.util` package. For practical purposes, a `StringBuilder` has two methods: `add(a: Any)` and `result()`. The method `add` uses its argument's `toString` method to compute the `String` to append, and `result` simply returns the `String` you have been accumulating. Here's an example of how one uses a `StringBuilder`: [5]

```
1 val sb = new StringBuilder();
2 sb.add(2);
3 sb.add(" and ");
4 sb.add(3.0);
5 sb.add(" is "+5.0);
6 val s = sb.result();
7 assert s.equals("2 and 3.0 is 5.0");
```

## 3.3 Files

Let's start simply, with the program in figure 3.1. [6]

The fun begins with line 4.

**lines 4 and 5:** The basic drill for reading and writing files is to get hold of a `File` object, and use it to get a stream either inbound or outbound. Executing "`I = new`

[5] See file StringBuilding.x10
[6] See file FileIO.x10

`File(inputPath)`" converts a file-system path into an object that represents the corresponding file, which may or may not yet exist. This is more in the spirit of Java than C++: this step is bypassed in a lot of C++ applications and only the streams are visible.

The reason for having a separate `File` class is to be able to perform file-level operations. For example, the call `I.exists()` returns `true` when the file constructed in line 7 actually exists, and `I.isDirectory()` returns `true` when that file not only exists, but is a directory. Like the `String` class, `File` is a work in progress. You will eventually also be able to do operations like making a directory and renaming a file.

Most of the time, you will wind up using an instance of `File` as an argument to a constructor that builds an input or output stream with some desirable properties like buffering, or handling line enders appropriately in text files.

**line 6:** The call `O.printer()` returns an output stream that is an instance of the class `Printer`. This is the class to which the familiar `Console` streams `OUT` and `ERR` belong, and it is an example of a stream that helps you write line ends in an operating system-independent way. You have already seen the `println` method in action already. The `print` method, which we use here, is the same as `println`, but does not append the line ender. [7]

When the `Printer P` is created, the file to which `P` refers is opened for writing *and is created if need be.* If the file exists already, its contents are discarded.

**line 7 to 9:** There's no mystery, we hope, about what the loop does: iterate over the lines in `I`, copying each to `O` via the stream `P`. While it is clear that `I.lines()` represents the sequence of lines in `I`, it is probably not obvious *how* it does. The answer is that it implements the interface `x10.lang.Iterator`. An `Iterator` is always associated with some underlying collection—in our example, the sequence the lines in `I`. If the collection contains elements of type `T`, the type is `Iterator[T]`, and it has two methods:

> **next(): T** returns the next element, as yet unseen, from the collection.
>
> **hasNext(): Boolean** returns `true` if a call to `next()` at this point will succeed.

Written in terms of `Iterator`'s methods, our `for` loop looks like

```
for(lines = I.lines(); lines.hasNext(); ) {
  P.print(lines.next());
}
```

[7] `Printer` also has a `printf` method that is X10's take on the familiar C function, but where the data to be formatted is most efficiently passed as an `Array`, although if you have 6 or fewer items to format, you can just string them out, C-style, as extra arguments.

Although there is some chance that, for some `Iterators`, a smart implementation might generate better code for the `for...in` loop than for the loop written out explicitly in terms of the iterator, one should view the `for...in` loop as a concise, readable way of writing the loop, not as an optimization.

**line 10:** We call `flush()` to make sure that, in case the `Printer` has been buffering our output, all of it gets sent *now*. Do not count on the output stream being flushed when the program terminates—even when it terminates normally. Better safe than sorry here.

If you compile `FileIO.x10` and then execute

```
%x10 FileIO FileIO.x10 out.put
```

in the same directory as `FileIO.x10`, you should find that `out.put` is a copy of `FileIO.x10`.

This example is fine for text files. Our next examples are two classes that show how to read and write a binary file.

Let's look at the writer first, because we need it to generate input for the reader. Our goal is to create a file whose contents is a sequence of `Doubles`. We don't want the `String` literals, we really want to store the numeric values as they appear in memory. We'll read the `Strings` in from the keyboard (the "standard input stream"): [8]

```
1   val I = new File(args(0));
2   val W = new FileWriter(I);
3   while(true) {
4     val line = Console.IN.readLine().trim();
5     if (line.length() == 0) break;
6     val dbl = Double.parse(line);
7     W.writeDouble(dbl);
8   }
9   W.close();
10 }
```

We've skipped the imports here—by now, you know what's needed there, and the working file is available in files/WriteDBL.x10.

**line 2:** We use the class `x10.io.FileWriter` to stream the output. It has methods like `writeInt`, `writeFloat`, and so on, that allow you to write binary data easily out to a file. In this example, we are going to use `writeDouble`.

Just as when one creates a new `Printer`, when one creates a new `FileWriter`, either a new file will be created, or if a file of that name already exists, all of its contents will be discarded. In either event, you start with an empty file.

[8] See file WriteDBL.x10

**line 3:** The keyword `while` introduces a loop. The loop's syntax is `while(`*`boolean-Expression`*`)` *`statement`*. The expression is called the *loop's test*, and the statement is the *loop's body*. When execution reaches the test, it is evaluated, and if the value is `true`, the body is executed. This sequence—check the test, execute the body—is continued until the test fails: *i.e.* evaluates to `false`. Execution then passes to the code that immediately follows the body.

In our example, the test is *always* true, so the loop is terminated from within its body, by the `break` statement on line 5.

**lines 4 to 5:** We read a line of ordinary text from the standard input stream and trim any white-space characters from the beginning and end of the line. If the line is now empty, we break out of the loop. The statement "`break;`" causes execution to continue at the first statement following the loop's body.

**lines 6 to 7:** The input line is converted to a `Double`, and that value is written, bit-for-bit, to the output file.

**line 9:** We explicitly close the stream here. This should guarantee that any output that has been buffered, but not yet actually written out, does get written: in other words, it should guarantee that `flush()` is called before the file's info is discarded. You can, of course, call `flush` explicitly at any point where you want make sure your output to that point is made persistent. This is particularly worth remembering in the early stages of debugging, where normal completion may not occur.

We compiled our `WriteDBL.x10` and ran it to create a file for the reader to read:

```
% x10 WriteDBL dbl.out
1.25
...
%
```

If you look at the bytes in `dbl.out`, the first 8 look like `0x3ff4000000000000`. Since $1.25 = 5/4$ in hex is $0x1.4$, this is about what we'd expect: some exponent stuff followed by `0x14`.

Now it's the reader's turn.
[9]

```
1 public static def main(args: Array[String](1)) {
2   val inputPath = args(0);
3   val I = new File(inputPath);
4   val R = new FileReader(I);
5   while(true) {
6     Console.OUT.println(R.readDouble());
7   }
```

[9] See file ReadDBL.x10

```
8 }
```

You can see that it is almost line-for-line the same as the writer—no surprise, really. The one thing that is different is figuring out how to terminate the `while` loop in lines 5-7. The problem is that `FileReader` does not offer a method for testing when we are at the end of a stream. You might think, therefore, that `readDouble` might return a nonsense value on end-of-stream, so we could test for that. It does not. To see what's going on, look at the console log we got by running the code shown with input from the file `dbl.out` that we just created:

```
%x10 ReadDBL dbl.out
1.2
2.3
3.4
5.678
x10.io.EOFException
   at x10.io.InputStreamReader.read(InputStreamReader.java:73)
   at x10.io.Marshal$LongMarshal.read(Marshal.java:877)
   at x10.io.Marshal$DoubleMarshal.read(Marshal.java:1057)
   at x10.io.Reader.readDouble(Reader.java:164)
   at ReadDBL.main(ReadDBL.java:107)
   ...
```

We read the file and got just the output we would have expected, and the first attempt to read beyond the data we provided caused an "end-of-file exception". Which brings us to our next topic: exceptions.

## 3.4 Exceptions

### 3.4.1 The Basics

An exception in programming, as in life, is literally an event out of the ordinary. Being out of the ordinary, exceptions are a rarity. If they happened frequently, they would be part of the rule, and not an exception. In programming, an exception is a way of performing an *abnormal return* from a method call:

> An abnormal return is one where the called routine cannot sensibly return a value to its caller (or simply return to its caller if no return value is at stake) because some situation, usually something terribly wrong, has arisen that the caller may not have enough context to handle. Often, that context is only available to a method that is many steps earlier in the call chain. The problem is to get control efficiently to where it can be exercised effectively, without unduly penalizing the "normal", exception-free paths through the code.

Instead of each method in the call chain having to look at the problem and then having to admit that it doesn't know what to do about it, we force a method that thinks it *can* handle it to say so *before* the problem occurs, not *after*. The construct is called a `try` statement. If you're not familiar with it, no problem: we'll describe it in detail in a moment. The important thing for now is that not every method need take responsibility for every sort of problem that might happen while it, or something it calls, is executing.

When running `FileIO.main` for example, the attempts to read and write the files *could* fail, but how often does that really happen? Almost never. If our `main` really wanted to deal with the error, it could put the code inside a `try` statement that says it is willing to deal this exception—we'll show the more cautious code in a moment.

The lingo of the trade is that exceptions are either "thrown" or "raised." X10 belongs to the "throw" camp. The methods that want to handle an exception are said to "catch" it. To say that an X10 method "throws an exception" means that execution reaches an X10 `throw` statement, which looks like:

**throw new** $ExceptionSubclass(explanation)$

For example, if a file open failed because an incorrect path was provided, you might see

```
throw new x10.io.FileNotFoundException("Bad path '"+path+"'");
```

What happens when this statement is executed is that an instance of the class being thrown, `x10.io.FileNotFoundException` in our example, is constructed. The class whose name appears in the `throw` must be a subclass of `x10.lang.Exception`[10]. The newly created instance has a field that holds the argument in the constructor we called "*explanation*" above. The catcher can retrieve it by calling the method `getMessage()`. The X10 run-time manager will also provide the instance with the call chain at the point of the "throw", which can be retrieved (as an array of `Strings`, one for each method in the call chain) by calling the method `getStackTrace()`. The X10 run-time manager will then look back up that call chain until a method is found that says that it wants to handle any exception whose type is a subclass of the type named in the `throw`. Control is returned to that method at an appropriate point—we'll see where in our next example.

Let's rewrite our `ReadDbl.main` so that it handles all its I/O problems gracefully: [11]

```
1 public static def main(args: Array[String](1)) {
2   val inputPath = args(0);
3   val I = new File(inputPath);
```

[10] Strictly speaking, you only need a subclass of the parent class of `Exception`, which is called "`Throwable`". As a general rule, application code should use or subclass `Exception` for problems that application code can reasonably be expected to handle. `Throwable` should be reserved for X10 and for situations that you don't expect application code to handle.

[11] See file ReadDBL2.x10

```
4    var r: FileReader = null;
5    try {
6       r = new FileReader(I);
7       while(true) {
8          Console.OUT.println(r.readDouble());
9       }
10   } catch(eof: x10.io.EOFException) {
11         Console.OUT.println("Done!");
12   } catch(ioe: x10.io.IOException) {
13     Console.ERR.println(ioe);
14   } finally {
15     if (r != null) r.close();
16  }
17 }
```

Here's the line-by-line:

**line 4:** Our reader `r` is now a `var`, not a `val`, because we are going to use it when we reach line 15, and we cannot be sure that the assignment in line 6 will succeed. Remember, a mistyped input file path will cause the constructor to throw an exception—an `x10.io.FileNotFoundException` in fact, which happens to be a subclass of `x10.io.IOException`. So the assignment in line 6 may be aborted.

**line 5:** This is the beginning of the `try` statement, the "try block". The brace following the keyword `try` is required. The catch blocks in lines 10 and 12 describe the types of exceptions that this `try` statement promises to handle. For example, if the assignment in line 6 does *not* complete normally, the exception thrown will be an `x10.io.IOException`, which will cause control to go to line 11, the body of the catch block that promises to handle `x10.io.IOExceptions`.

**lines 7-9:** The `while` loop is unchanged. It goes its merry way until the end of file exception is thrown, but now, since there is a catcher at line 10, control will go to line 11 when the exception is thrown.

**line 10:** The try block is ended by the closing brace, "}". What follows is a catch block, which, in English, says: "if an exception was thrown in the try block that is an `x10.io.EOFException`, or is an instance of a subclass of `x10.io.EOFException`, then return control to my body, namely, the statement bracketed by my braces. Set the identifier `eof` to the object that was thrown." At end-of-file, we don't care about the `eof`'s message or the call-chain traceback, so we simply ignore `eof`. We're just done!

All that remains, once the `println` completes, is to be polite and close up the `FileReader r`. Because there is a "finally block" at line 14, control goes to its body, and the close is done there. Control then falls through to whatever

statement (if any) comes next. If there had been no finally block, then once the body of the catch block had been executed, control would have passed to the first statement following the *last* catch block.

**line 12:** Here we have a second catch block, one that is intended to catch errors. When you have several catch blocks, they behave like "`if...else if ...else if ...`" sequences: the first one that can do the job is the one that the run-time picks: later blocks are ignored. *This is very important:*

> If we had put the `IOException` block *first*, then, because `EOFException` is a subclass of `IOException`, the `IOException` block would get control on end-of-file as well as whenever an error occurred. This is definitely not what we want, and this can be a pernicious error, because it is not always obvious what classes are subclasses of what other classes. This is why, if we had reversed the order of our catch blocks, the compiler would have signalled it as an error: "`Unreachable catch block for EOFException. It is already handled by the catch block for IOException`". Try compiling files/ReadDBL3.x10 to see this in action.

**line 13:** The argument to `println` is just the exception `ioe`, which may seem surprising: why not `ioe.getMessage()`? The answer is that since `ioe` is not a `String`, but is in a context here where a `String` is expected, the compiler replaces `ioe` with `ioe.toString()`. For all exceptions, the method `toString()` simply calls `getMessage()`, so `println(ioe)` winds up being the same thing as `println(ioe.getMessage())`. **(B: This digression seems a bit misplaced – and easy to lose. Could we make it larger and more clear? give it its own section somewhere? :B)**

**line 14:** The statement following the keyword `finally` will be executed whether or not the try block terminates without throwing an exception. The `finally` block will be executed even if an exception was thrown that is *not* one we have a catcher for in this try statement. One of the catch blocks may be executed before the entering the `finally`, if there is a catcher for the error that occurred, but no matter what, control will reach the `finally` before leaving the method.

That is why we had to be careful to provide an initial value for `r`: if the constructor in line 6 aborted, control would have gone to the second catch block, and when it completed, control would wind up in the "finally" at line 14. Because we were careful to initialize `r` in line 6, we are on safe ground accessing its value here, no matter how we got here. Actually, the compiler would have complained if `r` had *not* been set along some path that starts at its declaration and reaches its use in line 15. It is an error in X10 for code to read an identifier's value before the identifier has been initialized.

A `try` statement may have a `finally` but no `catch` blocks: no matter what exception occurs, the `finally`'s statement will get executed. The reason is exactly the sort of thing we saw in this example: if you have acquired some resource and need to release

it, putting the release in a `finally` guarantees that it happens. A `try` statement need have no `finally` block. In that case, when the body of the try block completes, control goes to the first statement that follows the last of its catch blocks.

## 3.5 Functions

### 3.5.1 Function Types

We have already seen examples of how one can create functions in X10 and pass them as arguments. Indeed, functions are values like any other from the point of view of assigning them, passing them around as arguments, and so on. As a "type", though, they form a distinct unit, being neither class, struct, nor interface.

The basic syntax for function *types*—as opposed to the syntax for specific method or closure definitions—is:

```
(arg1Type, arg2Type, ...)  => returnType
```

For example, we could declare

```
var doSum: (Array[Double](1)) => Double;
```

The value of `doSum`, when it is assigned, will be a function that takes a single argument, of type a singly-indexed array of `Doubles`, and returns a value of type `Double`. When it is helpful for people reading the declaration to have a name for the argument, you can supply one:

```
var doSum: (a: Array[Double](1)) => Double;
```

Here it is not very useful, but there obviously are a lot of situations where the variable name and argument types by themselves do not reveal your intent.

The argument list for a function type may be empty, as it was for our random number generator in the Monte Carlo calculation in 1.2.3.

Getting back to `doSum`, we know that, being a `var`, it can be set (and reset!) whenever we wish. It would be polite, however, to assign an initial value in the declaration that, if nothing like what we eventually want, at least would betray an attempt to use `doSum` before it was properly set. Because `Double` is a `struct` we cannot set it to `null`. For `Doubles` the closest thing to `null` is the value "not a number!", `Double.NaN`—not `0.0`, which is often a reasonable value after the fact, as well as initially.

```
var doSum: (Array[Double](1)) => Double
        = (a: Array[Double](1)) => Double.NaN;
```

or to let it throw an exception:

```
var doSum: (Array[Double](1)) => Double
     = (a: Array[Double](1)): Double => {
         val msg = "doSum called before being set.";
         throw new IllegalOperationException(msg);
       };
```

These two examples show the two patterns for creating function "values". One is exemplified by the right-hand side of the first assignment, in which the arrow "=>" is followed by an expression that is the return value of the function. The other possibility is to write out the function body in the usual statement form, bracketed by braces, which is what we did to throw the exception. When you use this format for the function body and want to return a value, you need to use a `return` statement to do so, just as you would in a method definition.

Notice that in *both* of these assignments, a semicolon follows the body of the function. In the first case, it looks natural enough, but in the second, where the semicolon follows the closing brace, you may feel that it looks a little strange. Remember, though, that what we have here is an assignment (to a `var` of its initial value), not a method definition, and assignments expect to be terminated with semicolons.

The bottom line is that the syntax for function literals is the same as that for function types, except that for a literal, the body follows the arrow rather than (as for types) the return type. You need not specify the return type for a function literal explicitly any more than you need do so for a method definition, unless (as in our second assignment here) the literal does not return the type of value you really want.

### 3.5.2 A Few Words On Arguments

We are going to give a few examples here that lay out X10's rules for what happens when you pass an argument to method and then assign a value to it. While this discussion should present few surprises, particularly to Java programmers, we suggest you bear with us and read on, as the parallels are not exact.

Consider, to begin with, the following attempt to pass some `Int` arguments into a method. [12]

```
1 public class TryArgs {
2   public static def tryargs(val a: Int, b: Int, var c: Int) {
3     a = b = 1;
4     c = 1;
5   }
6 }
```

[12] See file TryArgs.x10

You'll find this code in `primitives/TryArgs.x10` When you compile it, you will get a pair of error messages:

```
% x10c TryArgs.x10
TryArgs.x10:3: Final variable "b" might already have been
            initialized
TryArgs.x10:3: Final variable "a" might already have been
            initialized
```

The complaint about attempting to reset "a" after you declared it to be a `val` is what you would expect: you cannot assign a value to an already initialized `val`. [13] Why the same complaint should hold for "b" is less obvious, but now you know: *unless you explicitly say that an argument is a* `var`*, it is going to be a* `val`. Thus, of the three arguments, only `c` may be reset in the method body.

The next thing to try is modifying a `var` argument in the body of the method, as in: [14]

```
1  public class TryArgs2 {
2    public static def tryargs(var c: Int) {
3      c = 1;
4      Console.OUT.print("In tryargs, c is "+c);
5    }
6    public static def main(args: Array[String](1)): Void {
7      var ta2: Int = 0;
8      tryargs(ta2);
9      Console.OUT.println(". In main, ta2 is "+ta2);
10   }
11 }
```

[13] The term "`final`" comes from Java's way of describing read-only data fields.
[14] See file TryArgs2.x10

# 4 Dealing With Concurrency

In our discussion of the Monte-Carlo computation of $\pi$, we introduced four of the five main constructs that X10 uses to describe concurrency:

- `at` to do something at a different place;
- `async` to start a parallel computation, or in X10 terms, to spawn an activity;
- `finish` to wait for a bunch of concurrent computations to be done;
- `atomic` to keep things from happening at times when they shouldn't—or if you prefer, to keep access to shared resources orderly.

The missing construct is `when`, which is related to `atomic`, in that it is used to have one activity wait for another one to do something. We'll give some examples in a moment—see section 4.3.3.

There are a few other X10 constructs, and some useful concurrent data structures, but if you know the five, the rest are easy. For example, the `AtomicInteger` class (section 4.3.2) gives you a way to work with an integer variable atomically at lower cost than the generic `atomic` statement would incur. Bottom line: what remains to be learned has more do with efficiency and grace than much fundamentally new.

## 4.1 Here, There, and Everywhere

Every bit of running X10 has a *place* where it is running. We've already seen places in use in Section 1.3.1. In most other parallel programming languages, each place has a unique integer id, and that is how you work with it in the running program. X10 also provides a unique id, but as part of a larger object whose type is `x10.lang.Place`. Places are thus objects, so you can talk about them and compute with them as you would with any other object.

Why not just an integer? One reason is that X10 assumes that different places might be running on different kinds of processors. Some places may be implemented on a subset of the cores in a graphics processing unit (a "GPU"), others might be more conventional CPU's, and still others on special purpose chips perhaps associated with

sensors of some kind. How nice, then, if you can ask a place whether it is a GPU by evaluating "`place.isCUDA()`"—CUDA is one of several architectures for GPUs, and `isCUDA`, as it happens, really is a public method for `Places`.

The concept of `Place` has, therefore, been carefully kept abstract: an X10 program's places partition its address space. That's all they do. Places can correspond to processors, but they don't have to. If you are implementing X10 on a multiprocessor, you would probably set it up to have one `Place` per CPU. That would give your users precise control over where their data and computation goes.

But you needn't stop there. In principle, you could also implement a "uniprocessor mode" for the multiprocessor. You could then have a runtime flag that turns the mode on, with the effect that the multiprocessor and its memory are treated as a single place and the operating system's thread manager distributes the work among the available processors. From the point of view of the X10 language, this is a perfectly reasonable thing to do.

You can also go the other direction and arrange to have many `Places` per processor, whether or not the physical processor is a multiprocessor. For example, you might want to test your supercomputer code on a laptop by putting eight or sixteen `Places` on its single core. This may not run very fast, but it will run—or at least hobble—and will give you a chance to find some bugs.

### Getting Work Done At Other Places

The set of `Places` available to a program is fixed when the program starts. [1]

The constant `Places.ALL_PLACES` tells you how many places there are. The following program displays it on the console: [2]

```
1 public class PlaceCounter {
2   public static def main(argv:Array[String](1)) {
3     Console.OUT.println("This is running on " +
4       Place.ALL_PLACES + " places.");
5   }
6 }
```

**(B: WE NEED MORE HERE ABOUT THE COMMAND LINE AND ENVIRONMENT. Run this with a couple different sets of parameters, showing the parameters on a couple of settings and how you control the number of places. Evidently multi-place is broken on Macs just now, alas. :B)**

[1] X10 isn't built for dealing with changes in the available hardware in mid-application. It can't handle what amounts to plugging new computers in in mid-run. While there *are* applications for which plugging new computers in mid-run makes sense, handling a dynamic set of `Places` across all applications would slow down those computations that do not require that flexibility, and it is these more limited application's performance that is X10's primary concern.

[2] See file PlaceCounter.x10

If an activity needs to get a computation S done at another place p, it executes the statement at(p) S. The initiating activity's thread that is requesting the at is suspended, all necessary data from the initiating place is copied to p, and then S is executed at p. For example, the for loop here displays "Hello...' from every available place exactly once: [3]

```
1 public class HelloFromEveryPlace {
2   public static def main(argv:Array[String](1)) {
3     for(p in Place.places()) {
4       at(p) {
5         Console.OUT.println("Hello from " + here);
6         assert here == p;
7       }
8     }
9   }
10 }
```

**line 3** Here we loop over a collection. We've already seen looping over a range like 1..N. In fact, X10 allows loops over anything that looks like a collection of things – we'll see the details in Section **??**.

Place.places() implements the interface Sequence, which is the interface that C and Java arrays present: a fixed size and indexed by a single integer. The entries in the Sequence are all the Places available to the program, indexed by their ids. The for...in loop i iterates over all the entries of the Sequence in order. In this example, that means each Place should be visited in order by id.

**line 4** The body of the loop is this single at(p) statement. Let's look carefully at what happens as the loop is executed.

*Startup:* When execution reaches the for loop, the action is at a Place that we'll call pMain.[4]

*The top of the loop:* p gets assigned the next entry in the Sequence Place.places(), unless none remain, in which case we exit the loop.

*The transfer:* The thread executing the for is suspended. A new thread is spawned at p.

*The body:* The thread at p executes the body of the loop.

*The bottom of the loop:*. Once the thread executing the body completes its work, it dies, and the thread for the for loop is resumed at pMain. The effect is to go back to the top of the loop.

[3] See file HelloFromEveryPlace.x10

[4] Normally, main is called because the class is being invoked to do its thing from the command line: "x10 HelloFromEveryPlace". In that case, pMain by convention is the Place with id==0. But for our purposes right now, it does not matter at what Place main is executing.

The important thing to understand here is that *there is no concurrency*. Even though every `Place` participates, and even though there is one more thread than there are `Places`, *there is still only one thread active at a time.* From the point of view of the X10 language, what we are looking at here is a single activity: it happens to involve a lot of `Places` and threads, but still: it is a single, serial set of operations.

**line 5** The identifier `here` is reserved by X10. It always refers to the `Place` where the computation is happening at the moment the occurrence of `here` is accessed. On line 5, that is the `Place p` to which the `at` on line 4 sent us.

**line 6** Just for the sake of the example, we check that `p` really is the same as `here`. This is an assumption that we expect always to be true, and would be very upset if it turned out to be false. The `assert` statement is designed for just that purpose.

(In this case, the rules of the language guarantee that the two really are the same. We wouldn't check on it with an `assert` in most real programs – unless we suspected that the compiler was broken.)

`assert E` checks that an expression `E` whose value is a boolean (that is, whose value is either `true` or `false`) is `true`, and, if it's not, it reports the error to the standard error stream and *aborts the activity that failed the assertion.* Thus, `assert` both documents and enforces your assumptions in a nice, compact way. Another virtue of using `assert` over a plain old `if` is that there is a compiler switch "`-noassert`" that turns off all the `assert` statements in your code.

By the way, something subtle has happened here that is easy to overlook. In order to compare `p` with `here`, *the thread at* `p` *has to know the value of the variable* `p`: the value of `p` got sent from one place, `pMain`, to another, `p`. We say that the `at` statement "captured the variable `p`". We'll learn much more about what `at` statements have to capture shortly.

In `HelloFromEveryPlace`, `at(p)` is, in effect, being used as a command: "go to `p` and execute the statement that follows." It can also be used just to compute an expression. Again, there is no concurrency: "`x = at(p) e`" suspends the thread computing `x`, computes the expression `e` at `p`, copies the value back to `x`'s `Place`, and resumes `x`'s thread, which stores the value into `x`. We saw this in action in our multi-place version of the Monte Carlo computation of $\pi$. Here's another example: [5]

```
1 public class AtExpr {
2   public static def main(argv:Array[String](1)) {
3     val pMain = here;
4     val pNext = pMain.next();
5     val nextNext = at(pNext) here.next();
6     Console.OUT.println("The next next is "+nextNext.id);
7   }
8 }
```

[5] See file AtExpr.x10

**Line 3:** `main` starts at somewhere that we'll call `pMain` again.

**line 4** Whenever `p` is a `Place`, `p.next()` is the next one: the one whose id is 1 more than `p`'s. If `p` is the *last* `Place`, then `p.next()` is the `Place` with id 0: `next()` wraps around.

**Line 5** At `pNext` we ask what the next `Place` is and assign that value, back at `pMain`, to the variable `nextNext`.

## 4.2 Concurrency: Walking *and* Chewing Gum

For those too young to remember, the gibe "He wasn't smart enough to walk and chew gum at the same time" was famously (and very unfairly) aimed at President Gerald Ford. You don't want people saying that about your code. One of the main reasons for one processor sending a computation—particularly a big one—off to another processor is that while the second processor handles the computation, the first processor can continue working on other things.

Now, `at` gets you to other processors, but, as we have been emphasizing, by itself, `at` does not lead to a parallel, concurrent thread. For example, if the two calls to `bigComputation` in

```
val big1 = at(Place.places(1)) bigComputation(100,200);
val big2 = at(Place.places(2)) bigComputation(200,100);
```

are independent of one another, there is no reason not to do them concurrently. But the code, as we have written it above, will wait for the assignment to `big1` to complete before initiating the second, because the first `at` will suspend its own thread until its remote thread finishes.

Getting `Places` 1 and 2 do most of the work using this straight-line code yields no speedup *per se*, and it won't do so no matter what `bigComputation` is actually doing. The two constructs that we need to get the two `Places` working simultaneously are:

`async S:` to start a new activity that runs in parallel with its originator. The new activity executes the statement `S` and then dies.

`finish S:` to execute the statement `S` and on reaching the end of `S`, suspends its own activity until every activity spawned by executing `S` has completed.

If you come from the C world, this is an extension of the familiar "`fork/join`" duo, except that `finish` allows you to wait for a whole set of activities, whereas `wait` is a per-thread operation. Here's how X10's two work together to make our example hum: [6]

[6] See file WalkAndChew.x10

```
1  val n = 3;
2  val big1: GlobalRef[Cell[String]] = GlobalRef[Cell[String]](new Cell[String]("!"));
3  val big2: GlobalRef[Cell[String]] = GlobalRef[Cell[String]](new Cell[String]("2"));
4  val pMain = here;
5  finish {
6    async at(Place.place(0)) {
7      val bc1 = bigComputation(n,n);
8      at(big1.home) big1()() = bc1;
9    }
10   async at(Place.place(0)) {
11     val bc2 = bigComputation(n,n);
12     at(big2.home) big2()() = bc2;
13   }
14 }
15 assert big1()().equals(big2()());
```

## (B: Picture! :B) (B: Explain that funky `big2()()`! :B)

**Line 4:** We capture the `Place` we are coming from, so we can let the `Places` that are doing the dirty work know where to send their results.

**Line 5:** This `finish` guards two `asyncs`. When control reaches here, the two `asyncs` will get executed. Control will then reach the end of the `finish`'s block at line 14. The `finish`'s activity will be stopped until the two `asyncs` have both finished. Only then will line 15 be reached.

**Lines 6 and 10:** Each `async` creates an activity that runs in parallel with the `finish`. That activity computes its value at the remote `Place` and then "goes home" to slam the result into memory there.

**Lines 8 and 12:** Here we see the "`at`"s capturing `bc1` at `Place` 1 and `bc2` at `Place` 2. and assigning their values back home, which is wherever the original activity was executing.

**Line 15:** When we get here, we *know* that both `big1` and `big2` have been set to their new values, so the comparison is safe.

If you want to play with this code, we have provided the serial version in WalkThenChew.x10, and the parallel version in WalkAndChew.x10

You might also want to experiment by changing line 10 with "`at(Place.places(2)) async`", in which the new activity is spawned at the *remote* `Place`, rather than at the current one.

Finally, a more realistic example of this sort of program is a shared-memory (single `Place`) version of the *Quicksort* algorithm, a working version of which you can find in QSortInts.x10.: [7]

[7] See file QSortInts.x10

```
1  public class QSortInts {
2    /**
3     * top-level call: sorts the input array in place using the
4     * quicksort algorithm.
5     * @param data the array of Ints to be sorted.
6     */
7    public static def sort(data: Array[Int](1)) {
8      val r = data.region;
9      val first = r.min(0), last = r.max(0);
10     sort(data, first, last);
11   }
12   public static def sort(data:Array[Int](1),
13           left:Int, right:Int) {
14     var i: Int = left, j: Int = right;
15     val pivot = data(left + (right-left)/2);
16     while (i <= j) {
17       while (data(i) < pivot) i++;
18       while (data(j) > pivot) j--;
19       if (i <= j) {
20         val tmp = data(i);
21         data(i++) = data(j);
22         data(j--) = tmp;
23       }
24     }
25     finish { // when you are here i > j
26       if (left < j) async sort(data, left, j);
27       if (i < right) async sort(data, i, right);
28     }
29   }
```

**Line 10** The algorithm is recursive: the call `sort(data,left,right)` sorts the slice of the array between the indices `left` and `right`. We assume that `left <= right`.

**Line 15** This is a naive choice of "pivot" point: we hope that the element in the middle of the slice is close to the median for the slice. This loop reorganizes the slice into two subarrays: (1) those elements less than the pivot, and (2) those greater.

**Fine Point:** If you're wondering why we wrote `left + (right-left)/2` rather than the more natural `(left+right)/2`, the answer is *integer overflow*: the intermediate sum `left+right` may overflow, but if `left` and `right` are non-negative `Int`s and `left <= right`, then the difference `right-left` is not going to overflow, and since `left + (right-left)/2 <= right`, our computation of the middle index is safe.

**Lines 16-24** This loop partitions the input slice of `data` into two parts, and reorganizes `data` so that the left part consists of the entries that should precede the pivot, and the right part consists of those that should follow it. This is straightforward serial code.

**Lines 25-28** We can use two independent activities to sort the left- and right-hand portions of the slice. The `finish` guarantees that when control reaches the end of the method, the whole slice specified by its arguments has been sorted.

**Exercise:** If you stare at this code for a while, you'll probably realize several things. First, we don't really need *two* `asyncs`: we only need an extra `async` when we have to sort *two* subarrays. If there's only one (which can happen if by some evil chance, the pivot turns out to be the largest or smallest element in the slice), there's no gain to spawning a new acitivity: the current activity can handle it. Also, we really don't have to put the "`finish`" in the recursive call, *if we make that method* `private`*, so no one will mistakenly call it directly*. The `finish` gets moved into the public, top-level `sort` method, because we don't really care in what order the spawned activities finish: all we care is that *all* of them finish. This brings up an important point: it does not matter how, when, or where the activities inside a `finish` get spawned—all must be complete before the `finish` will allow its own activity to resume. So the activities spawned by the recursive calls are all monitored by the one `finish` at the top level when you use it to guard the call to `sort` in line 10.

If you want to see our version of this improved code, you can find it in QSortInts2.x10. In QSortInts3.x10, we push it one step farther with a less naive choice of pivot.

**Fine Point:** Finally, a different question, why just sort `Ints`? Why not sort data of any type whatever, so long as we know how to compare two things of that type using the binary operators "<" and ">"? To do that, we need syntax that allows us to declare the operators "< " and ">", and `x10.util.Ordered[T]` that is exactly what we need. It requires its implementors to provide four "methods":

```
operator this < (that: T): Boolean;
operator this > (that: T): Boolean;
operator this <= (that: T): Boolean;
operator this >= (that: T): Boolean;
```

These declarations are just like the abstract method declarations we've seen in interfaces before, except that the keyword phrase "`operator this`" is used rather than "`def`". Of course, there is nothing special about comparison operators. You can also implement any of the arithmetic and bitwise operators for any class or struct you create. [8]

If a class or struct `T` implements `Ordered[T]`, and if `t1` and `t2` are two instances of `T`, then these declarations say we can execute `t1 < t2` to determine whether `t1` is less than `t2`, and of course the same for the other three operators..

[8] For examples, take a look at `x10.lang.Arithmetic.x10`, where all of the numeric operators are declared, and `x10.lang.Bitwise.x10`, where you'll find the shifts, and's, and or's.

How do you declare concrete versions of these operators in your own classes and structs so that they can implement this interface? It's as easy as you would think: add the keyword "`public`" and provide the body of code that implements the operator:

```
public operator this < (that: T): Boolean {
   // your implementation comparing "this" to "that"
   // goes here
}
```

The only difference, once again, from a method declaration is that `operator` **`this`** replaces `def`.

There is one final problem: how does one say "my class `T` implements `Ordered[T]`" when you go to use `T` in a declaration? The operator "`<:`" that we introduced on page 42 does the trick:

```
public class QSort[T]{ T <: Ordered[T] } { ... }
```

You can find our version of the code with these modifications in chapter-concurrency/QSort.x10.

It is a good exercise to run this code on some big examples in order to understand the cost for writing generic code versus type-specific code. We provided some minimal wrappers for `Int` and `String` as examples to get you started using our generic code. Strings are an interesting variant, because if they have an average length much greater than 8 or 10, then—unlike the story for Int—significantly more effort may be required to do the comparisons than the overhead of the array operations and `async` management. This should reduce the relative penalty for going generic (which should be independent of the item size and comparison cost). We also have provided a "native" Quicksort for strings, QSortStrings.x10, that you can use to compare with the generic.

### 4.2.1 More About Asyncs

When control reaches `async S`, a new activity in parallel with the executing activity is spawned to execute `S`. The two activities are both running in the same `Place` and have access to the same variables, but otherwise they are independent of one another, except for such synchronization as the originating activity imposes—for example, executing the `async` within a `finish`.

The best way to think about multiple activities happening concurrently at a single `Place` is to imagine there is one master activity, "*the scheduler*", that runs with special privileges. The scheduler alone gets to decide, at each point in time, which activity is going to run. It will generally limit the time the activity can be run before it interrupts the activity on behalf of the other waiting activities. It may also choose to interrupt an activity part-way into its time slot, even in mid-statement, if it perceives a need, like an

I/O operation that must be serviced promptly. Finally, an activity can ask the scheduler to suspend it until some event occurs: "As soon as the value of the variable `x` is greater than 12, let me resume, please."

Having the scheduler around is a little like having a partner helping you by writing some of the code. Sequential code that you write all by yourself can be pretty bad, hard enough to get right. Concurrent code can be much, much worse: it has all the problems of sequential code plus a few really bad ones of its own because your co-author, the scheduler, neither knows nor cares about your intentions.

We are going to look carefully at three problems that you don't find in sequential code:

1. **Data races:** As we saw when we went to implement the multi-cluster Monte-Carlo computation of $pi$ (page **??**), the scheduler can create situations where several activities sharing a resource (like a storage location or an I/O stream) can run into trouble because they failed to coordinate their actions. We'll expand on that example in a moment.

2. **Deadlock:** Deadlock happens when some family of activities can make no progress because each one has asked the scheduler to suspend it because it is waiting for one of the other activities to update some variable. We'll see an example of how this might happen in §4.3.4.

3. **Error handling:** Errors occur in serial code, too, of course. But parallelism adds a level of complexity. Suppose an activity `A` spawns an activity `B` to do some job—store a row in a relational database, perhaps. `A` goes on doing whatever it has to do while `B` does its thing. Suppose that `A` doesn't need any *result* returned by `B`, it just needs `B` to do what it was asked to do. What happens if `B` fails? How does `A` find out? When? It may or may not be okay for `A` to continue when `B` fails. Somehow we've got to get help from the scheduler, who is responsible for the overall management of our activities. We'll discuss all this in Section **??**

None of these problems is X10's fault. In fact, X10 has features that make writing parallel code safer and more readable than many high-performance languages. (There are languages that give even safer concurrency, but they pay for it by having programs run substantially more slowly.) Our aim is that by the time you finish this book, you will be comfortable using X10 to write serious concurrent code that's visibly—and in many cases, even *provably*—safe.

### 4.2.2 Finish Really Means "Finish Everything Everywhere"

The `finish S` construct executes the statement `S` and then waits until *all* activities started by `S` are finished. We've seen examples that show it waiting for the activities started by `asyncs` appearing explicitly in `S`, but you should be aware that it also waits for any activities spawned by those `asyncs`, and even activities spawned by methods called while executing `S`. If you worked through all of the Quicksort examples, you will have seen this at work. Here is a much simpler program, with a more direct example:

9

```
1  var a:Boolean = false, b:Boolean = false, c:Boolean = false;
2  def nestAsyncs() {
3   finish {
4       async {
5         async { a = true; }
6         async { b = true; }
7       }
8       async { c = true; }
9    }
10   assert a && b && c;
11 }
```

When `nestAsyncs()` is invoked, four activities get started: one at line 4, which in turn starts the activities at lines 5. and 6, and finally the original activity spawns another at line 8. The `finish` beginning at line 3 and ending at line 11 waits for all four of these to finish before control proceeds to line 10. So, the `assert` there is always going to be succeed.

We can rewrite the program so that some of the activities are spawned by a different method. Even this does not fool `finish`, which still waits for all of the activities to finish. [10]

```
1  var a:Boolean = false, b:Boolean = false, c:Boolean = false;
2  def spawnAsyncs() {
3     async {a = true;}
4     async {b = true;}
5  }
6  def nestAsyncs() {
7    finish {
8       async spawnAsyncs();
9      async { c = true; }
10   }
11   assert a && b && c;
12 }
```

The activity started in line 8 calls `spawnAsyncs`, which spawns two `asyncs`. At the point where we reach the end of the `finish` at line 10, as many as 4 `asyncs` may be alive. The main activity is blocked until all 4 finish.

[9] See file NestAsyncs.x10
[10] See file IndirectAsyncs.x10

### 4.2.3 When To Use `finish`

Most `async`s should have a `finish` controlling them. The usual idiom is:

```
finish {
   /* start some async computations here */
}
/* use the results here */
```

If you don't have a `finish` around the `async`s, then you have no way of dependably knowing when all of its results are available. Sometimes this may be exactly what you want—some examples:

- Perhaps you have just finished some big part of the computation and are spawning off an activity to write the answer to a file. If the current activity doesn't need the file, there's no need to wait for the child activity to finish writing.
- Perhaps at some point there are several ways to get the result you need. You might want to fire off one `async` per way. For example, "compute from scratch", "look it up in the database", and "search the Web". Here all you want is one answer, so all you care about is when the *first* computation in the set succeeds. You will probably want to have some way to kill the others then, rather than letting them run and waste time and resources. We'll look at this sort of pattern in section **??**.
- Perhaps the activity is some kind of background chore, like listening for incoming network traffic and servicing it. It doesn't particularly have a final answer. On the contrary, it's supposed to run quietly doing its thing forever. Waiting for it to finish would be ridiculous. It's not supposed to.
- Perhaps the newly spawned activity will run for a long time, but the main activity, although it *does* need the result, has to proceed with some other work: you care about the answer, you just don't want to be idle while you wait for it. Some other communication mechanism is needed for the child activity to tell the main activity when its result is available. We'll worry about this pattern in section **??**.

While none of these is particularly outré, none of them is terribly common in X10 programming, either. Most activities should have a `finish`.

**Basic Question:** Is there a point in the currently executing program that you *might* need to be sure a set of `async`s has completed?

If the answer is "Yes!", you obviously need put the `async`s inside a `finish`. If you're not sure the answer to that question is "No!", err on the side of caution: you probably want the `finish`. Remember that the scheduler will eventually cause you problems if it possibly can.

**A Less Obvious Question:** Can one or more of the `asyncs` have terminated only because some unrecoverable error cause them to be aborted, not because they actually completed the task they were set to do?

From the point of view of the `finish`, done is done: it does not ask *how* an `async` terminated, it just wants to know that it *has* terminated. Is there hope that if you knew why the abnormal termination (an "*abend*") occurred, could you recover? Or the flip side: if an abend occurred, dare you continue? We'll discuss ways to handle these questions in section 4.4. For the moment we just wanted to make sure you were aware of what `finish` does *not* do for you.

## 4.3 Data Races

Let's quickly review that story with a very simple example: **(B: The following needs to be checked when X10 is buildable :B)** [11]

```
1 public class AsyncInc {
2     public static def main(args: Array[String]) {
3         var n : Int = 0;
4         finish for (var m: Int = 0; m<2; m++) {
5             async for(var k: Int = 0; k<10000; k++) n++;
6         }
7         Console.OUT.println("n is "+n);
8     }
9 }
```

This code appears to be incrementing the variable `n` 20,000 times. But does it? We ran it several times on a 2-core laptop, and the console display for the third run was "`n is 17573`", and not "`n is 20000`". We then ran it a few more times and pretty consistently we got an answer near 17,500. So we somehow lost roughly 2,500 out of 20,000 increments, or one in every eight.

**Fine Point:** The reason for having the big `for` loop in the `async` is that we want the thread it creates to do enough work that there is a reasonable, if still small, probability that it *will* get interrupted.

As we described in parallelizing our Monte Carlo code (§1.3.2), the cause for our troubles is that while `n+=1` may look like "one operation", it is not: it is not *atomic*. The X10 compiler is going to treat `n+=1` as a 3 step process: (1) fetch the current value of `n`, (2) add 1 to to get the new value, and (3) store the new value back in `n`.

The problem is clear: if there really are three steps, then the scheduler is free to interrupt either activity after any one of them. The way the outer loop is written here, there will be two activities $A$ and $B$ running at once. Some small percentage of the time, the

[11] See file AsyncInc.x10

scheduler will interrupt one of them—-let's say it is $A$—-before it completes step 3. Suppose that $B$ then gets to run and happens to start by loading the current value of `n`, which we'll call `nStart`. It is the same value that $A$ was about to update, because $A$ never finished storing its update. Next suppose that the scheduler allows $B$ to run 1,000 iterations of the inner loop, and then suspends it to give some other activity a chance to run. At this point, `n`'s value is `nStart + 1000`. If $A$ gets to run next, it will pick up where it left off, namely, by storing into `n` what it, $A$, thinks the updated value should be now: `nStart+1`. So instead of `n` being `nStart + 1001`, `n` is now `nStart + 1`. 1000 increments have been lost. Forever.

Because we put everything in a big `for` loop, we created 20,000 chances for the scheduler to interrupt at a bad spot. Sure enough, by the third time we ran the program, it did: perhaps it did only once, but once was enough. That is the true horror of this sort of bug: failures happen, but they happen only rarely and are therefore easily overlooked. If we'd made the `async`'s `for` loop a lot smaller, say 2,500 as the limit rather than 10,000, we might have gone dozens or even hundreds of executions before seeing any failures at all.

If your activities have any bugs in them of this sort, you should expect that the scheduler will eventually find and reveal them. If you are lucky, it will *not* happen during an important demo. If you are unlucky, your program will pass all the tests and demos just fine, and the error will happen when many peoples' lives are depending on the automobile, airplane, or medical equipment your program is controlling.

So: it is best to imagine that *the scheduler is a wicked, conscience-free gnome out to ruin everything if you let it*, and then make very sure that you never let it.

We are, unfortunately, not just being facetious here. Concurrent and distributed programming is notoriously tricky, precisely because it is so hard to know what is going to happen next. The structured patterns that X10 provides for dealing with concurrency and the discipline these patterns enforce are based on several decades of lessons too often learned the hard way.

### 4.3.1 Curing Races: Atomic Power

There are two basic constructs in X10 that allow activities to coordinate their access to shared data:

`atomic S:` `S` is an X10 statement. Once one activity begins to execute `S`, no other activity may enter that code until the first activity finishes. In other words, at any one moment in time, *at most one activity can be running* `S`.

`when(expr) S:` Here `expr` is a boolean expression and `S` is a statement. When an activity's execution reaches the `when`, the expression `expr` is evaluated, and if it is false, the activity is suspended. Otherwise, the statement `S` is executed. The scheduler resumes a blocked `when` to execute `S` only after `expr`, by virtue of some other activity's work, has become true. The whole construct `when(expr) S` is guaranteed to be executed atomically—as if you had written `atomic when( expr ) S`.

Let's be precise about what we are guaranteeing, because what is at stake is the behavior not just of one `atomic` block, but the mutual behavior of a set of activities executing a set of `atomic` and `when` blocks:

> An X10 program in which all accesses (both reads and writes) of shared data appear in `atomic` or `when` blocks is guaranteed to use that data atomically, and *no races involving those accesses can occur.*
>
> Atomic sections at the same `Place` are mutually exclusive. That is, if one activity $A$ at a `Place p` is executing an `atomic` or a `when` block, then no other activity $B$ at `p` can also be executing an `atomic` or a `when` block concurrently. If such a $B$ does attempt to execute an `atomic` or a `when` block before $A$ finishes the active block, $B$ will be suspended until $A$ finishes its block.

If some accesses to shared data are not protected by `atomic` or by `when`, there are no guarantees: you are on your own, and whatever happens to you happens to you.

We'll look at a bunch of examples of these constructs in action in a moment. First, though, we want to say a few more words about their costs and correct use.

The first and most important thing is that the guarantees do not come for free. The statement `S` in these two constructs cannot be arbitrary. To begin with, `S` cannot spawn another activity, nor can it use statements like `finish` and `when` that might block an activity.[12] Finally, `S` cannot use `at`. There are rationales for all these restrictions. For example, allowing something like a `when` in `S` might cause `S` to deadlock the program, because it can get blocked at the `when`, and our guarantees say that *no other atomic construct can execute at* `S`*'s* `Place` *until* `S` *completes.* Please take our word that the others have similar rationales.

We need to say a little something about cost. The point of these constructs to serialize access to shared data in a predictable way. The choice was to serialize the atomic constructs themselves: at most one runs at a given time at a given `Place`. This flies in the face of our goal to exploit parallelism as much as we can. It can be very expensive for programs in which multiple activities need frequent access to volatile shared data. One activity executing an `atomic` block can stop 100 other activities cold who want to use other `atomic` blocks, *even if these other blocks share no data at all with the active block*.

A classic example where one has to be careful is when one activity is pushing small jobs onto a queue for some set of other activities to handle. If the jobs are many but small, the queue maintenance is at the heart of the matter and has to be handled carefully to minimize the overhead due to the serialization.

### 4.3.2 Efficient Atomic Expressions

X10 provides some less expensive ways to get some of the same functionality in some common situations. Our failing `for` loop in §4.3.1 above is an example. We can use

[12] You may nest an `atomic` block in `S`, but it is not clear when that might be desirable.

`atomic` to cure this race, as we did in Figure **??** in §1.3.2, page **??**, for a similar problem with our Monte Carlo code: [13]

```
1 public class AtomicInc {
2    public static def main(args: Array[String]) {
3       var n : Int = 0;
4       finish for (var m: Int = 0; m<2; m++) {
5          async for(var k: Int = 0; k<10000; k++) atomic n++;
6       }
7       Console.OUT.println("n is "+n);
8    }
9 }
```

Forcing the `n+=1` to be executed atomically guarantees that the sum is driven home to `n` every time the expression is evaluated, and guarantees that no value of `n` can be loaded by one thread while the other is processing it.

Several activities accessing a shared variable is such a common event that X10 provides a family of classes in the package `x10.utils.concurrent` that make executing the update atomically more efficient than using an `atomic` block.[14] For example, the class `x10.util.concurrent.AtomicInteger` can be used for our asynchronous increments: [15]

```
1 val n = new AtomicInteger(0);
2 finish for (var m: Int = 0; m<2; m++) {
3    async for(var k: Int = 0; k<10000; k++) n.getAndAdd(1);
4 }
5 Console.OUT.println("n is "+n.get());
```

**Line 1:** `n` is now a *reference* to an `AtomicInteger` rather than in `Int`. The initial value of `n` is `0`. There are several methods for updating that value, all guaranteed to be atomic:

> `n.set(k:Int)` sets the value to `k`.
>
> `n.getAndAdd(k:Int)` returns the current value of `n`, but before doing so adds `k` to the current value and stores the result as the new value of `n`.
>
> `n.addAndGet(k:Int)` adds `k` to the current value of `n`, stores the result as the new value of `n`, and returns the result.

[13] See file AtomicInc.x10

[14] The APIs are modeled on an analogous set of classes that are in the Java standard distribution in the package java.util.concurrent. You may find the documentation there useful.

[15] See file AtomicIntInc.x10

`n.compareAndSet(expected:Int, newValue:Int)` compares the current value of `n` with `expected`, and if the two are equal, sets `n`'s value to `newValue`. The return value is a `Boolean`: `true` if the expected and actual values were the same, `false` otherwise.

If you just need the current value—no update—you use `n.get()`.

**Line 3:** Since `addAndGet` returns the updated value of `n`, the call `n.addAndGet(1)` is precisely the equivalent of "`atomic n+=1`".

The timings on a dual core laptop showed the `atomic` version running nearly three times as slowly as the `AtomicInteger` version for the C++ runtime. Your experience may not be as dramatic, (and we'll show an example in a moment where a well-placed `Atomic` sometimes wins), but it pays to at least try `AtomicInteger` when you can.

The other classes in `x10.util.concurrent` are `AtomicBoolean`, `AtomicDouble`, `AtomicFloat`, `AtomicLong`, and `AtomicReference`. These classes have methods similar to those for `AtomicInteger`, with minor variations to accommodate the different types.

**Exercise** Our first example of an atomic update was back in our discussion of the Monte-Carlo computation of $pi$. Our final code, shown in Figure **??**, uses an atomic update in the method `countAtP` to protect the integrity of the final `count`: [16]

```
1  public static def countAtP(pId: Int, threads: Int, n: Long) {
2     var count: Long = 0;
3     finish for (j in 1..threads) {
4        val jj = j;
5        async {
6           val r = new Random(jj*Place.MAX_PLACES + pId);
7           val rand = () => r.nextDouble();
8           val jCount = countPoints(n, rand);
9           atomic count += jCount;
10       }
11    }
12    return count;
13 }
14 /**
```

Try replacing `count` with an `AtomicLong` here. You can use either `addAndGet` or `getAndAdd` in line 9 to update count, because the return value is ignored here:

17

```
async inCircle.addAndGet(countPoints(nPerThread, rand));
```

[16] See file MontePiCluster.x10

[17] See file MontePiAsync2.x10

There might not be much of an effect on the running time, but why not try it and see? We're not being coy here in not telling you what you wiil see. That will be very much dependent on hardware, operating system, and implementation parameters for your system.

To understand better the trade-offs of various ways to use `atomic` and the atomic helper classes, we are going to look at a number of ways to implement a histogram for an `Int`-valued data set. Recall that a histogram tracks the number of occurrences of each value in a data set. Here is a bar graph for the histogram of a collection of 9 integer values $\{1, 2, 2, 1, 3, 1, 3, 3, 4\}$.

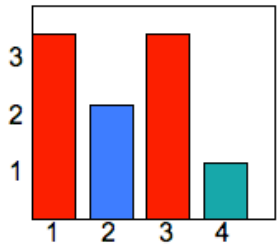


Figure 4.1: A Histogram For The Values $\{1, 2, 2, 1, 3, 1, 3, 3, 4\}$

The main loop for computing a histogram serially is very simple:

```
for (i in firstIn..lastIn) result(input(i))++;
```

The `result` array has one entry per possible input value, and initially all of these entries are 0. The `for` loop contains a new idiom: it is read: "for all integers `i` in the range `firstIn` to `lastIn` (including these end points) do...". In this case, what we want to do is for each entry `input(i)` in the `input` array, to increment the count in `result(input(i))` by 1. When the loop terminates, `result(k)` will be the number of times `k` appeared as `input(i)` for some `i`.

**Fine Point:** The syntax "`for(i in firstIn..lastIn)`" is identical in meaning to the older style "`for(var i:Int=firstIn; i<=lastIn; i++)`". So which to use? The main advantage of the new syntax is that `i` appears once instead of three times, which makes for a more readable loop for most people, and is a definite plus if you don't like to type. But seriously, if the natural name for the loop variable would be `cancerCellCount`, writing it once in all its glory is much better than collapsing it to `cccnt` to avoid a clumsy, overlong `for` loop in the older style.

Back to histograms! Assume that we are working at a single `Place`, but we can take advantage of multiple activities. An obvious approach for parallelizing this loop would be to partition the `input` array into disjoint slices, one slice per activity, and then combine the partial results in a shared `result` array. If there are `N` activities and the input array is indexed from 0, then:

```
val sliceSize = input.size/N;
finish {
  for(i in 0 .. (N-1)) {
    val sliceStart =  i*sliceSize;
    val sliceEnd = i==N-1 ? input.size-1 : sliceStart+sliceSize;
    async {
      val partial = new Array[Int](result.region, 0);
      for (i in sliceStart..sliceEnd) partial(this.input(i))++;
      this.foldIntoResult(partial);
    }
  }
}
```

Computing the histogram for each slice is done serially. The fun begins with folding the partial result into the final result, which we're going to realize as an instance field of "`this`". The field `this.result` is accessed concurrently by all of the `asyncs`, so we have an obvious data race to handle. There is more than one way to do so, and we have provided several in the classes for this chapter with "`Histogram`" in their name. The top-level class is `Histogram.x10`. It has a `main` that exercises all of the other classes:

**Make the whole folding operation atomic:** What might seem a drastic approach to the problem is to let "`atomic`" guard the whole operation of folding in the partial result:

```
atomic for (i in partial) this.result(i) += partial(i);
```

The effect is to serialize the folding of the partial results into `N` sequential operations whose cost is proportional to the `result.size`. See `AtomicFoldHistogram`.

**Make each update of `this.result` atomic:** At the other extreme, we could guard each addition:

```
for (i in partial) atomic this.result(i) += partial(i);
```

This is done in `AtomicHistogram`.

**Use AtomicIntegers for `this.result`:** The folding loop becomes

```
for (i in partial) this.result(i).addAndGet(partial(i));
```

See AtomicIntHistogram.

**Don't partition the input's indices, partition its range:** Another approach to the whole problem is to partition the range of values, rather than the input data set. You avoid synchronization problems that way: each `async` reads the whole input array, but just updates its own slice of the `results` array. You can find this in RangeSplitHistogram.

Why so many variations? There are two parameters that determine the problem size: the range of input values and the number of input values. We've set up the command line arguments for `Histogram`'s `main` to allow you to compare the performance of these various approaches. Our experience is that varying these parameters allowed us

to make each approach a winner at least once. We invite you to try it out yourself. Even the serial code performs best in some surprising in our case, because the operating system is allowing it to use both cores of our dual-core chip at one time effectively, even though this code is written serially. Bottom line: what to guard and how to guard it is not a one-size fits all decision insofar as the effect on performance is concerned. Be prepared to experiment!

### 4.3.3 `when`: Conditional Atomic Blocks

Next, let's implement a class that we'll call `AtomicCell`. Think of it as a *very* small buffer. It can hold one value, which, for simplicity, we'll make an `Int`. Since it only holds one `Int`, an `AtomicCell` can be either *empty* or *full*. You can put a value into it, if it's empty; otherwise, you have to wait until someone else empties it. You can take a value out of it, if it's full; otherwise, you have to wait until someone else fills it. As we said: a very small buffer.

Those clauses "wait until someone else empties it" and "wait until someone else fills it" are best coded using "`when`".

```
 1  class AtomicCell {
 2    var full : Boolean;
 3    var contents : Int;
 4    def this(init:Int) {full = true; contents = init;}
 5    def fill(newVal:Int) {
 6      when(!full) {
 7        full = true; contents = newVal;
 8      }
 9    }
10    def empty(): Int {
11      when(full) { full = false; return contents; }
12    }
```

**Lines 8-12:** When `fill` is called, the first thing that happens is that the expression `!full` is evaluated atomically. If the result is `true`, execution immediately continues to the assignments on line 10. The whole process, test and assignments, is carried out as one atomic statement. No other activity is able to act at this `Place` from the moment the evaluation of the test expression `!full` begins until the two assignments are both complete.

Of course, it might happen that on entry `!full` is `false`, in which case the executing activity will be suspended. It is probably best, once again, to think of there being a scheduler who puts `!full` on a "watch list" that it checks from time to time. When the scheduler discovers that it has become `true`, it *immediately and atomically* evaluates the assignments. There is no gap between the scheduler determines that `!full` is true and when it evaluates the assignments.

**Lines 13-15:** `empty` is the mirror image of `fill`: the test is whether `full` is `true`, and if so the cell is emptied and what had been its contents are returned.

You can exercise this code with:

```
public static def main(Array[String](1)){
   val c = new AtomicCell[Int](0);
   async for([i] in 1 .. 10) c.fill(10*i);
   async for([j] in 1..10) {
      Console.OUT.println("value "+j" is " + c.empty());
   }
}
```

The full source is in AtomicCell.x10.

There are two activities, one of which puts ten numbers into the cell, and the other takes the first nine of them out again, leaving the cell full:

> The cell `c` starts out full, with 0 as its `contents`.
>
> On lines 3 and 4, two activities are spawned. The scheduler will, as likely as not, give the first one (on line 3) the first shot at running. Since a full cell can't be added to, this activity will be blocked at the `when` in `fill()`.
>
> At that point, the scheduler may allow the root activity to run to completion, or, on checking that state of `c.full`, allow the second `async` a chance to run. If it happens to favor the main activity, then almost instantaneously that activity, which, having spawned the two `asyncs` already, now has nothing else to do and will terminate. The scheduler is left with only one activity it can run: the second `async`, the emptier. One way or another the emptier gets to run.
>
> It immediately retrieves `c`'s contents and sets `c.full` to false. It is then either interrupted by the scheduler or tries to empty `c` again, which fails: there's nothing there, so it is suspended.
>
> But now the first `async` can proceed again... From this time on, once the root activity has terminated, the scheduler has only two activities to worry about, and at any given moment, only one of them is not blocked. So the two alternately run to completion.

There is an important point here: notice that when the root activity—the one executing `main()`—died, the other two activities remained alive. This is critical: the root activity for an X10 program *always runs inside a* `finish` *that is executed by the X10 runtime.* From the X10 scheduler's point of view, therefore, the root activity is nothing special: it is just an `async` like any other. The `asyncs` spawned by the root are all monitored by that `finish`, which returns control to the runtime only when those activities have all terminated. This is exactly the same as the way that a `finish` in your code returns

to control to an activity when the activities spawned inside the `finish`'s body have all terminated

There are two guarantees that make `when(expr) S` statements work:

**Timeliness:** If the expression `expr` becomes true and stays true for long enough, eventually `S` will get executed.

**Correctness:** Noticing `expr` is true and evaluating `S` are done atomically.

If we didn't have both of these, `AtomicCell` wouldn't work. The correctness guarantee says that if the scheduler every gets around to seeing whether a blocked `when` should be executed, it does the job of checking and executing it correctly (in this case, meaning atomically). But this guarantee is only worth something if the runtime guarantees that the scheduler is *fair*, that it will check the blocked `when` in a timely manner.

If you think about these guarantees for a moment, you will realize that to exploit them well requires some discipline on your part. There are two governing principles, the first of which is:

> If the statement `S` in `when(expr) S` does something (like remove some data from a buffer) that *might* affect the value of `expr`, make sure that all changes to whatever variables affect `expr` are also done as part of `S`.
>
> Furthermore, any changes to the variables affecting `expr` that occur elsewhere in the program should be done so that the effect on `expr` is atomic and consistent with its semantics as the test in this and any other `when` blocks in which it is the test.

This is sometimes described as "assuring that the test for a `when` is *stable*".

Here's the second principle, which is actually a variation on the first:

> Never forget that if you are going to allow an activity to be blocked trying to enter a `when`, you had better have at least one other activity around that eventually can free the blocked activity.

Here is an extreme example of violating the first principle with an unstable test. First we write a function that computes a `Boolean`:

```
private var flag: Boolean = true;
private def alternate() {
   val oldFlag = flag;  flag = !flag;  return oldFlag;
}
```

Evaluating `alternate()` alternates between `true` and `false`. And for the very unstable `when`:

```
when(alternate()) {
 pleaseDoSomethingAboutThis();
}
```

Suppose there are two activities with such `whens` in them. No implementation of the X10 scheduler could hope to guarantee that both of these two activities both get their chance to run. You can imagine a sequence like

1. First activity is checked and sees the value to be `true`, so it gets to run. The next call to `alternate` will return `false`.

2. Next, the second activity is checked, and because this call to `alternate` returns `false`, it continues to be blocked. Of course, its call to `alternate` flipped the `flag` again.

3. The first activity arrives at the `when` again. Good news: its call to `alternate()` is going to return `true`.

4. ... back to step 2 ...

In short, unless the first activity terminates or gets interrupted for some other reason, the second activity might never be run, no matter how fair the scheduler tries to be. Notice that there is no activity around to help our indefinitely blocked second activity.

In addition to be careful about what you test, there are also some good practices for engineering the body of a `when` (and the bodies of atomic blocks more generally).

`atomic` blocks are expensive. When X10 executes an `atomic` block at a `Place p`, no other `atomic` block can execute at `p` at the same time. That can seriously cut down the concurrency in your program—as you might expect from a device that's intended to limit concurrency.

`when` blocks are even more expensive. The test expression will be evaluated atomically, repeatedly, whenever X10's scheduler thinks it's a good idea. This can take lots of time, especially if the expression takes time to evaluate. And, since it's atomic, all other potentially active `atomic` and `when` code will be idled while `p` is evaluated, regardless of what the `whens`' tests might yield.

X10 does provide some lower level primitives: locks, latches and monitors are available in the `x10.lang` package. We won't say anything more about these tools here: they have been around for decades now, and most texts on concurrency in operating systems explain them in detail, since that is where the constructs were first introduced. Here are some rules of thumb about how to use `atomic` and `when`:

- Do as little work as possible (but—need we say?—no less) in an `atomic` or `when` block.
- `atomic` is cheaper than `when`, and generally using classes like `AtomicInteger` is cheaper than either.

- Don't be afraid, though, to use `atomic` and `when`, especially in early versions of a program. The more efficient mechanisms for concurrency control, like locks, exact their own price in readability and robustness, so wait until you are sure you have a problem before you switch to them.
- The more activities there are at a `Place`, the more expensive `atomic` and `when` tend to get at that `Place`.
- The issue isn't `atomic` and `when` by themselves—they don't take all that much time to do. The issue is the loss of concurrency when you have several activities with many `atomic` and `when` blocks trying to run concurrently at the same `Place`.

`atomic` and `when` are excellent tools for getting your concurrent program started. They're easy to use, and they're very general. If your program works fine with them, you win. If it's too slow, switching to other concurrency-control devices is possible.

### 4.3.4 Putting It All Together: Implementing Queues

An example that expands on our original `AtomicCell` provides an excellent chance to see the considerations that can arise in deciding whether to use `atomic` or `when`—or neither. Instead of a single cell, we assume that we are given a buffer holding some number of items. For the moment, we'll assume that the buffer, once allocated, cannot be enlarged. The way we want to use the buffer is as a "queue". Items arrive at one end and are removed from the other. In other words, "first in, first out."

The usual drill is to add at the end and remove from the beginning of the buffer. Here is some serial code to do the job as simply as possible:

```
public def addLast(t: T) { // add instance of T to queue end
   if (next < buffer.size) {
      buffer(next++) = t;
   }
   else { /* you've got a problem: out of space */  }
}
public def removeFirst(): T { // remove and return first item
   if (next > first) {
      return buffer(first++);
   }
   else { /* you've got a problem: nothing left to remove */}
}
```

We're assuming here that we have a *very, very* big array as the buffer, and that, at any moment, the queue occupies the slice in that array from `first` to `next-1`, so it has size `next-first`. We'll worry about the two "problems" in due course.

The naive solution to parallelizing this code is simply to make the body of each method a single atomic statement. Our take on the naive solution is NaiveQueue.x10. The

effect is to serialize the accesses to the buffer. On the surface, it might appear that one could do better, because adding an item does not involve `first`. The problem is that the two operations share two variables: `next` and `buffer`. To see why that causes us grief, let's try to improve `addLast` by getting rid of the `atomic` and using an `AtomicInteger`, rather than an `Int` for `next`:

```
1    public def addLast(t: T) { // add instance of T to queue end
2       buffer(next.getAndAdd(1)) = t;
3    }
```

From the *adder*'s point of view, each addition is guaranteed to get a new value for `last`, and `size` will be incremented exactly once for each addition, so all is as it should be. Even if an adder is interrupted before executing line 3, it does not matter, in the sense that we don't lose any updates, we may just be slow in acquiring them. As for error handling, if you try to add an item to a full buffer, the X10 runtime will see the out-of-range array index and abort the call—see §3.4 for an explanation of what happens then. This is crude, but effective, since you expect overflow to happen rarely, if at all.

This all may look good from an adder's point of view, but a remover is not apt to be so happy.

> Suppose that an adder gets interrupted before its store of `t` into `buffer` completes in line 2, but after the call to `getAndAdd`. Let `nt` be the index where `t` is eventually to be stored in the `buffer` array.
>
> Now, along comes a remover. It will see `next.get() > first`, thanks to the completed call to `getAndAdd`. But, if it has the bad luck that `first == nt` now (which it easily could: this could be the first addition to the queue, followed by the first removal), the remover will return `buffer(nt)`, even though `t` has not yet been stored there. The caller will therefore get whatever garbage value we used to initialize `buffer`'s entries.

This suggests a way to get the remover back into the game: block on getting garbage until the store of the real thing has completed:

```
1    public def removeFirst():T { // remove and return first item
2       val firstNow = first.getAndAdd(1); // reserve our slot
3       when(buffer(firstNow) != GARBAGE) {
4             return buffer(firstNow);
5        }
6    }
```

This is fine, so long as you *know* that eventually enough elements will be added to unblock the removing activity. As for `GARBAGE`, if `T` is a class, `null` will usually fill the bill. For structs, you have to make the call.

We are not quite done yet. Using `when` has created a pit into which we can fall if we are not careful. Suppose that we have two activities $A$ and $B$. Suppose that *both of them add and remove items from the queue.* Consider the following scenario:

> $A$ needs an item off the queue, so calls `removeFirst`. The queue is empty, so it is suspended at the `when` until the queue end reaches the slot it is trying to read.
>
> There is still hope for $A$, because $B$ is still active. But suppose we have some bad luck and $B$ also needs an item off the queue before it has any items to put on it. $B$ calls `removeFirst` and now both activities are blocked: a classic deadlock.

This is not an unrealistic situation: suppose that $A$ and $B$ are servers that have incoming streams of job requests. So long as each can handle the incoming jobs, no queuing is done. But if either server gets too busy, it pushes jobs it can't handle off onto the queue. Later, if one becomes idle, it can go to the queue to ask for work to. Two producers, both also consumers.

Bottom line: we violated one of our two governing principles: we've allowed ourselves to create a `when` that, potentially, can block *all* of the activities that affect the variables it tests.

Here the cure is simple: no producer can consume from the queue it is feeding.

There is more to be gained by looking at this problem carefully. In the code samples for this chapter, we'ver provided several different implementations and a driver to exercise them.

## 4.4 Asynchronous Error Recovery

We've been pretty blasé so far about what happens when things go wrong in one of our activities and we need to bail out: to throw an exception. We covered the basics of exceptions in serial code in §3.4. The time has come to face up to handling exceptions in distributed code.

Distributed computing and multi-threading add a level of complexity to the problem of dealing with exceptions. On the one hand, there is a natural parent/child relationship between an activity and those activities that it spawns. On the other hand, the parent activity normally continues in parallel with its children, and such a parent cannot serve to catch any exceptions thrown by its children. Let's look at an example:

> Suppose that our program requires data from a number of sources: databases, perhaps, or the Web, or even just plain files. The list of inputs is in an array `sources: Array[String](1)`. The entries might by URLs, SQL queries, or file paths, or even a mixture of all three. Let's make life easy and assume that the data from each of these sources can be captured as an instance of the class `Stuff`. The key loop, guarded against trouble, could look like
>
> ```
> try {
> ```

```
   for(var n: Int = 0; n<sources.size; n++) {
      async stuff(n) = at(getPlace(sources(n)))
                           readSource(sources(n));
   }
}
catch(e: Exception) { ... }
```

When an exception is thrown in an activity that is not handled by one of members of that activity's call chain, X10 terminates the activity. The question is: what happens then? There may be more than one activity with the same parent alive. Several of them might throw exceptions, and if so, the problems almost certainly will occur at different times. One of two bad things might happen, both of which the X10's runtime, as we shall see, uses the `finish` construct to avoid:

**Ignorance is bliss:** The runtime simply terminates the failed `asyncs` and `stuff(n)`, for those `n`'s, winds up with whatever garbage in it that was there before the attempt to read the source. The exceptions are effectively ignored, their information lost.

**A race to the bottom:** Once the `async` is terminated, the exception is immediately fed back to its parent, the spawning activity, which does... something. But what, exactly? If the parent had continued alongside the `asyncs`, as it would if it is programmed as shown above, who knows where control would be in the parent when the exception was thrown? Even assuming that we did somehow manage to handle the first exception, what do we do if a second or third exception is thrown? Just ignore them? Interrupt the parent again? It looks like a classic race situation: what happens depends on who gets where when.

How does `finish` help? Let's stick a `finish` in front of the `for` and see what happens:

```
1 try {
2    finish {
3       for(var n: Int = 0; n<sources.size; n++) {
4          async stuff(n) = at(getPlace(sources(n)))
5                              readSource(sources(n));
6       }
7    }
8 }
9 catch(e: Exception) { ... }
```

Once control enters the `for` loop, all of the `asyncs` will be spawned and control will reach the end of the `finish` that starts at line 2 and ends at line 7. At that point, this activity will request the X10 run-time's process scheduler to block it until all the `asyncs` are done. If they all completed normally, no problem: the scheduler will restart

the activity at the statement following the `catch` clause at line 9. In other words, execution leaves the `try` in exactly the same way it does for *serial* code that has to catch exceptions.

Suppose, though, that one (or more!) of the `asyncs` is terminated because an exception was thrown, but not caught inside the `async`. As part of cleaning up after now dead `async`, the X10 scheduler will save that exception information somewhere. Later, when the last of the `for` loop's `asyncs` is done, the scheduler will see the saved exceptions, and instead of returning control normally, it will execute code that appears to the activity being resumed to be an exception thrown inside the `finish`. This exception contains all the information the main activity needs to see what went wrong in its `asyncs`. The `try` statement in line 1 that surrounds the `finish` will see the exception, just as it would see any exception thrown in serial code that it surrounds. The `try` statement will thus give control to the `catch` block on line 9, which says that it is willing to handle *any* sort of `Exception`.

For a working example, take a look at TwoFiles.x10. You'll see that what the scheduler throws is an instance of `x10.lang.MultipleExceptions`. It has as instance data an array containing all of the exceptions that the process scheduler has saved. That is how the `catch` can figure out what actually happened, but not necessarily to *whom* it happened: which `async` failed may have to be encoded in its exception's message.

Suppose an activity blows up, but it's immediate parent is *not* blocked inside a `finish`. How does X10 handle that? Because the parent is in no position to deal with the problem, the parent also has to be aborted. The upshot is that the scheduler tries the activity's grandparent, great-grandparent, and so on, until it finds an activity $B$ that *is* blocked in a `finish`. The scheduler will give $B$ the opportunity to cope by waking it up (so rudely!) with a thrown exception, as we described above. Of course, $B$ may not have a `try` statement that catches this sort of exception, in which case, $B$ will also be terminated, and the scheduler will continue to work back through $B$'s ancestors, until some activity *does* handle the exception. In the worst case, the root activity will be reached that started the application by kicking off its `main()`. X10's startup regime guarantees that that activity is always blocked in a `finish`, and that it is willing to field all exceptions. So, as the saying goes, the buck is guaranteed to stop there.

This is a good point to reiterate that the activities spawned by an activity $A$ can outlive $A$, *with the exception of the root activity that executes* `main()`. At any moment in time, the activity running `main()` is thus the ultimate ancestor of all of the program's live activities. It is helpful in this regard to keep in mind the distinction between *local* termination and *global* termination of an activity. An activity is terminates locally when that activity has finished executing the body of the `async` that gave it birth. It terminates globally when it has terminated locally, and in addition, all of the activities that it has spawned have also terminated *globally*.

A good way, therefore, to think about the activities for a progam that are alive at any given moment is that they form a tree, rooted at the `main()` activity, with two sorts of nodes:

"live" nodes: nodes that are still executing, and

> "zombie" nodes: nodes that have terminated locally (*i.e.*have finished their execution), but that still have descendants which are "live".

When a leaf node activity terminates, it is removed from the tree, and when a zombie node no longer has any immediate children, it is removed from the tree.

You may have been saying to yourself that there is another (better?) solution to dealing with exceptions that we forgot: *never write an* `async` *that doesn't catch its own exceptions.* Then we know that all `asyncs` terminate "normally." In simple situations, like the one in our example, you may be able to operate that way. It just means adding error information as an instance field in `Stuff`, so that when you go to use some `Stuff`, you will know that that `Stuff` is okay. You might well have to add this sort of information for other reasons anyway, in which case the added work is minimal. If, at some point in the code, you are going to want to check whether any of the `asyncs` failed, though, you still are going to need a `finish` somewhere inside of which all of these `asyncs` are being spawned—how else might you know that the store into `stuff(n)` has completed? So it is not clear that you've saved yourself much by moving the information about whatever problems there are out of the `Exception` and into the application data. One could even argue that—particularly in large, complex programs—putting the error information in the data may make it easier to overlook errors that need to be handled in a timely way.

That's enough for the moment about concurrency *per se*. We'll look next in more detail at distributing computations—the `at` statement revisited. Then we'll put everthing together in a chapter on local and distributed arrays.

# 5 There and Back Again: Computing with `at`

We've seen how an activity may shift execution to another `Place` temporarily by executing an "`at`" statement or expression. Such a shift is potentially expensive, because it requires, at a minimum,

> a message to the remote `Place` to kick off the execution there;
>
> as part of that kick-off, copying to the remote `Place` all data used in the operation;
>
> a message from the remote `Place` back home when the remote operation is complete; and
>
> if a value was being computed, it must be copied back.

The moral is that `at` must be used with care to minimize these costs, particularly any unnecessary copying of data. That is one of the two things this chapter is about. The other is the question of how to pass to a remote `Place` a reference to some object so that the remote `Place`'s computation can modify the object.

## 5.1 Hidden Treasure: Unexpected Copies

The first thing you need to do is to be aware of what *actually* gets copied, as opposed to what you see named explicitly in the body of the `at`. Of course, there are simple situations where what you see is what you get:

```
val a = 123;
val b = 456;
val c =  at(someOtherPlace) a + b;
```

No mysteries here: `a` and `b` are `Int`s, 4 bytes apiece, and those 8 bytes get copied. Things get a little nastier if you have arrays:

```
val bigArray = new Array[Int](HUGE_INT);
// code here to initialize bigArray...
for(p in Place.places()) at (p) doYourSlice(bigArray);
```

Here every Place gets to work with some slice of bigArray. But what gets copied? All of bigArray to each Place: 4 bytes per Int times HUGE_INT. Here's a simple example you can run to see that we're telling you the truth: [1]

```
1  public class ArrayCopy {
2    public static def main(args: Array[String](1)) {
3      val a = [1,2,3];
4      Console.OUT.println("initial a is "+a);
5      at(here.next()) {
6        Console.OUT.println("in at, a is "+a);
7        a(1) = 4;
8        Console.OUT.println("after assignment in at, a is "+a);
9      }
10     Console.OUT.println("back home, a is "+a);
11   }
12 }
```

[1] See file ArrayCopy.x10

# 6 Fancy Types

X10's types are a lot more powerful than we've seen so far. They can do a lot for you that types in Java and C++ can't do, if you want.

Now, most of what types do for you is to prevent mistakes. Here's a mistake:

```
var total : Int = 0;
total += "3";
```

`total` is an integer. You can't add a `String` value into it. This is a bit tricky, because you can add a `String` *to* it: `0 + "3"` evaluates to the string `"03"`. In some languages, you *can* add a string into an integer if the string contains a number, so adding `"3"` into `total` would behave just like adding `3` into `total`.

A good type system will catch this mistake as early as possible, and tell you about it in as useful a way as possible. For X10, this means that it'll catch the mistake when you try to compile your program – or even sooner, if you're using the programming environment X10DT – and it'll print out a message telling you that the code doesn't fit together right.

The other thing that well-designed types do is try not to bother you too much. This is a bit tricky. The ultimate purpose of types *is* to bother you – to get in your face when you're about to goof up. At the very least, you have to do some work in order to get any value of them. You have to write down some types in your program, enough to tell X10 what you mean, so it can catch when you don't do what you meant. But X10 often lets you leave types off, so they're not *too* much in your face.

## 6.1 Constrained Types

Here's a perfectly innocuous bit of code that looks just fine: [1]

```
1 val a : Array[Int] = [1,2,3];
2 Console.OUT.println("a(1)=" + a(1));
```

[1] See file ArrayRef.x10

But when you go to compile it, you get a warning message:

```
Generated a dynamic check for the method call.
```

(Or, if you are using `STATIC_CHECKS`, it won't compile at all.)

What's wrong here is that `Array[Int]` covers arrays of many different ranks – one-dimensional ones like `[1,2,3]`, two-dimensional, ten-dimensional, or whatever you like. When `a` was declared to have type `Array[Int]`, we told X10 to forget what rank `a` had.[2]

So, when X10 goes to use `a`, it doesn't know how many subscripts it takes. Giving it one subscript, in `a(1)`, might be right, or it might be wrong. X10 doesn't know. So, with dynamic checks on, it sticks in a test to find out.

And, since X10 doesn't know, it will do the same thing for `a(1,2)`. That will fail at runtime, since `a` is one-dimensional.

It would be nice if X10 could handle this sensibly – if X10 could know that `a` was one-dimensional, that one subscript was right, and two were wrong.

Well, it can do that — that, and a whole lot more. The mechanism it uses is called *constrained types*. Constrained types track things that the compiler knows about your data, like the ranks of arrays, or that particular variables aren't **`null`**.

### Tracking Nulls

One common problem in programs (in most object-oriented languages) is that some parts of code assume that some objects aren't **`null`**, but other parts of code don't know about this assumption and set them to **`null`**. This can cause `NullPointerException`s and considerable premature gray hair. For example,

[3]

```
1 val x : Person = lookUp("Kim Geep");
2 Console.OUT.println("Kim's phone number is " + x.phoneNumber());
```

If Kim isn't in the database, `x.phoneNumber()` will throw an exception.

If you want to protect yourself against this in X10, you can. Declare `x` with a type that says "`x` is a `Person`, but not **`null`**". Using this type obligates you to convince X10 that whatever value you give `x` is not null. Three good ways to convince X10 are: (1) use a constructor call, since constructors never return **`null`**. (2) use a method declared to return a non-null value, though you will have to convince X10 that method is defined

[2] This is why you should use `val a <: Array[Int]`... whenever you can – `<:` in a `val` doesn't lose track of this kind of thing.

[3] See file NeedsNullCheck.x10

properly. (3) Check the value at runtime, with a type-cast ("as") operation, and make sure it's not **null** there.[4]

For our sample program, we choose a version of database lookup that is guaranteed never to return **null**. If lookUpOrCreate(name) doesn't find name in the database, it creates a new record for that name. This approach doesn't make sense for all applications[5]. (In many cases, if some crucial datum really is missing, there's no way to figure it out, and the best thing that can be done is to report the error and ask a human for help. Throwing a NullPointerException is a greatly inferior choice.)

The code that uses non-null Persons is very much like the code that used possibly-**null** ones: [6]

```
1 val x : Person{self!=null} = lookUpOrCreate("Kim Geep");
2 Console.OUT.println("Kim's phone number is " + x.phoneNumber());
```

The code of lookUpOrCreate keeps track of the fact that its values are never **null**. The database db is a hash-table of non-**null**Persons. lookUpOrCreate itself returns a non-null Person. The constructor call at line 6 always produces a non-**null** value, and X10 knows that, so it works out fine too. [7]

```
1 static val db = new HashMap[String, Person{self!=null}]();
2 static def lookUpOrCreate(name:String) : Person{self!=null} = {
3   if( db.containsKey(name)) {
4     return db.getOrThrow(name) ;
5   }
6   val p = new Person();
7   db.put(name, p);
8   return p;
9 }
```

### 6.1.1 self, this, and all that

Now, it's time to learn what that self!=**null** means. You already know != and **null**, but self is new.

When we're using a type, we can think of it as a test, asking if some thing under consideration is a member of the type or not. So, for the type Int, imagine asking if **true** is a member (no), or 18 is (yes). With types that come from classes and such, like

[4] Method 3, checking values at runtime, is pretty familiar from defensive programming in Java or C++. X10's types, if used this way, have the advantage that you can't *forget* to do the check – if you forget, your program won't compile.

[5] Not even for this one – how do we really know the phone number of a newly-created Person?

[6] See file HasNullCheck.x10

[7] See file HasNullCheck.x10

`Int`, we don't need to have a name for the thing under consideration. When we write a constraint, we do need a name for it – and that name is a special X10 keyword, `self`.

So, we can write some other constraints. Like this one, which says that the variable `three` is an `Int` which is equal to 3. [8]

```
1 val three : Int{self==3} = 3;
```

Of course, if you try to set it to something that *isn't* three, it won't work. This doesn't compile: [9]

```
1 val four : Int{self==3} = 4;
```

Not-equals constraints are sometimes useful for excluding a few problem values. For example, if you're defining a reciprocal function, you might want to constrain the input to avoid dividing by zero: [10]

```
1 static def recip(n:Int{self!=0}) = 1.0/n;
```

This isn't a terrible idea, though it does require you to prove to X10 that numbers are non-zero when you take their reciprocals. Here are a few ways that you can do it: [11]

```
1 static def useRecip(m:Int, n:Int{self!=0}) {
2   val a = recip(n);
3   val three : Int{self==3} = 3;
4   val b = recip(three);
5   val c = recip(m as Int{self!=0});
6 }
```

**Line 2:** Use an argument which has been declared to be the right type. That's an ordinary way to call methods anyhow. Here, the right type is `Int{self!=0}`. Since `n` has that type, it's fine.

**Line 4:** Use an argument which has a *more specific* type. You're used to doing this too, when you have a method that takes a `Person` as an argument and you give it a subclass, like `Dentist`.

Here we're doing the same thing, in a slightly different way. The type of `three` is `Int{self==3}`. That's a subtype of `Int{self!=0}`, because every number that *is*

[8] See file ConstraintExamples.x10
[9] See file ConstraintExamples.x10
[10] See file ConstraintExamples.x10
[11] See file ConstraintExamples.x10

three *isn't* zero. The official word for this is *subtyping by strengthening*. `self==3` is logically *stronger* than `self!=0` — also pronounced "`self==3` implies `self!=0`". A constrained type with a stronger constraint is always a subtype of a type with the same base and a weaker one.

(As usual, we didn't have to write the type on line 3. We just wrote it to make the example clearer.)

**Line 5:** Use a *cast*. At some point, you're going to have a value like `m`, which you don't know in advance is not zero, and you're going to have to tell X10 that it's not zero. The way you say "it's not zero", here, is to use a type cast `m as Int{self!=0}`. This is just the same as any other time you have a value of one type and need it to be another type.

Of course, `m` might be zero. If `m` is zero, this cast fails – `m as Int{self!=0}` first makes sure that `m` is an integer (which it can do at compile-time), and then that it's not zero (which has to be done at runtime). If `m` is zero, this throws an exception, the same as any other attempt to cast a value to some type it isn't.

This isn't much of an improvement over getting a division-by-zero error. It might even be worse: the division by zero error is obviously about division by zero, but the cast error is a bit further removed from the problem.

The improvement comes earlier, when you try to write this:

```
val c = recip(m);
```

This is wrong, because `m` could be zero. X10 will give you a warning or an error here (depending on the `STATIC_CHECKS` compiler flag – see §**??**).

At this point, you have been alerted to the problem. It's your responsibility as an intelligent programmer to figure out the right thing and do it. Testing `m` and printing a comprehensible error message if it's zero, while untraditional in some circles, would not be out of the question.

### `self` in nested types

If you have a constrained type that looks like `T{c}`, where `T` is some type and `c` some constraint, then `self` inside `c` means *"the value of type T that we're wondering whether it might be a member of* `T{c}` *or not"*. So, in `Int{self!=0}`, `self` is the `Int` that we're saying had better not be zero.

That's a simple rule. But it has some implications that might be brainhurty. If you have a complicated type, there might be two `self`s inside it that mean different things – and are even of different types.

For example, `Array[U]{self!=null}` means "A non-`null` array of `U`'s". The array itself isn't `null`. The `U`'s inside of it might be `null`, depending on what `U` is. `a:Array[String]{self!=null}` makes `a` be a non-`null` array of strings, but `a` might have a `null` inside of it.

Similarly, `String{self!=null}` is the type of strings that aren't `null`.

So, we can use `String{self!=null}` for `U`. `Array[String{self!=null}]{self!=null}` is a non-`null` array of non-`null` strings. The first `self` refers to the `String` that isn't null. The second one refers to the `Array` that isn't `null`.

It's easy to write an utterly incomprehensible type expression using this. If you find yourself tempted to do so, look at the `type` declaration (§6.2). A couple of definitions and you can write this: [12]

```
var a : ArrayNotNull[StringNotNull];
```

which is much easier to read.

### 6.1.2 `val` variables instead of `self`

Another way of making constrained types in `val` declarations easier to read is to use the name of the variable being declared instead of `self`. So, you can write [13]

```
val n : Int{n != 0} = 3;
```

instead of [14]

```
val n : Int{self != 0} = 3;
```

This isn't quite a circular definition. It's saying that `n` is an `Int` and `n!=0`.

There are a few restrictions on this.

- It only works on `val`s. `var`s can never appear in constraints, as we see in §6.1.4.
- It only works for the variable being declared. For example, you can declare that the whole array of strings is non-`null`:

  ```
  val a : Array[String]{a!=null};
  ```

  which you might prefer to using `self` to say the same thing.

  ```
  val a : Array[String]{self!=null};
  ```

[12] See file ConstraintExamples.x10
[13] See file ConstraintExamples.x10
[14] See file ConstraintExamples.x10

But the entries in the array don't have names, so, to make them be non-`null`, you have to write

```
val b : Array[String{self!=null}];
```

If you want both the array and each entry to be non-`null` (which you often do), you can write:

```
val c : Array[String{self!=null}]{a!=null};
```

which at least uses `self` for only one thing, unlike the equivalent

```
val c : Array[String{self!=null}]{self!=null};
```

### 6.1.3 Guards

Guards are constraint-like expressions that control what can be done with a class or method. We've seen them in §2.3.1.

A guard on a class can be specific about the generic parameters. For example, if we've got an interface `Flavored`: [15]

```
1 interface Flavored {
2   def flavor():Int;
3 }
```

We can demand that a generic type variable implement `Flavored`: [16]

```
1 class Tasting[X] {X <: Flavored} {
2    def taste(x:X) = x.flavor();
3 }
```

The clause {`X` <: `Flavored`} is a guard.

A method can have a guard too. For example, consider a method to find the biggest element in an array of `Int`s. What's it supposed to do if the array is empty and has no biggest element? One way to deal with the problem is to forbid it entirely, and to require that the array not be empty. It could be done with a guard, like this: [17]

[15] See file ConstraintExamples.x10
[16] See file ConstraintExamples.x10
[17] See file ConstraintExamples.x10

```
def max(a:Array[Int]){a.size != 0} {
```

The guard {a.size != 0} insists that the array not be empty. It could just as well be phrased as a constrained type: [18]

```
def min(a:Array[Int]{a.size!=0}) {
```

Use whichever way you like more.

### 6.1.4 Legal Constraints

It would be very nice if you could write any sort of Boolean expression in your constraints. In fact, it would be way, way *too* nice. You could write constraints which no computer program could understand.

**Fine Point:** Basic theory of computability says that there is no computer program that (1) always halts, and (2) can accept any nullary Boolean function `f` as input, and tell whether `f()==true`. (Remember that `f()` might run forever, but the tester would have to halt in finite time, so it can't just run `f()` and see what happens. It has to look at the code of `f`, and any computability theory textbook will explain why that cannot work.)

So, if we allowed expressions liike `f()` in constraints, X10 would have some trouble. Does this compile?

```
val a : Int{f()} = 1;
```

Well, if `f()==true`, it does compile, and otherwise it doesn't. So the X10 compiler would have to solve that unsolvable problem in order to do basic type checking.

So, to keep type-checking deciable — and fast — only a scant handful of things are allowed in constraints.

### 6.1.5 Constraints can be...

**Equalities:** As we've seen, we can have equalities in constraints, like `Int{self==3}`.

**Non-Equalities:** As we've seen, we can say that two things have to be different, like `String{self!=null}`. (We can't use other kinds of inequalities, like `Int{self` $>0$`}`.)

**Type Equalities and Non-Equalities:** We can require that two types either be the asme (`T==U`) or be different (`T!=U`), too. These aren't used very often.

[18] See file ConstraintExamples.x10

**Subtyping:** We can require that one type `T` be a subtype of another type `U`. This *is* useful, especially for the guard of a class. We've seen it used already, back in §2.3.1 and §6.1.3.

**Property Method Invocations:** "Property methods" (§**??**) are very limited sorts of methods – they can say just the sorts of things that belong in constraints.

**Conjuctions:** You can put several constraints together, but only with `&&`. See §6.1.5 for details.

**Zero Test:** You can test to see if a type has a default value, with `T haszero`. See §**??** for some examples.

**`true` and `false`:** A true constraint is always true; a false constraint never is.

### Expressions Allowed In Constraints

When you compare expressions for equality, or invoke property methods, you re restricted to only a few sorts of expressions. You can write `n==0`, but not `n == m+1`.

**Constants:** `3` and `"fish"` and **`null`** and so on.

**`vals`:** You can use `val` variables. Not `var`s, which change too much for X10's constraints to understand.

**`this`:** If the constraint is at a point in the program where **`this`** makes sense, like in a method, you can use **`this`** in a constraint.

**`here`:** The same goes for `here`.

**`self`:** If you are constraining a *type*, you can use `self`. If you are writing a guard, there's no `self` to be found.

**Properties of `self`:** If `p` is a property field of `self`, then you can write `self.p` in a constraint. Just `p` alone can mean `self.p`, if that is the only choice.

**`val` fields of anything else:** You can extract `val` fields of anything else.

**Property Method Invocations:** You can use a property method in a constraint expression, if the property method's actual expansion works there too.

### Having your cake and eating it too

If you want `n` to be neither zero nor one, you can write it in one of three ways.

- `val n : Int{ n!=0, n!=1 }`, using a comma to separate multiple constraints. Commas don't mean "and" anywhere but constraints.
- `val n : Int{ n!=0 && n!=1 }`, using the regular `&&` operator to mean "and".

- `val n : Int{ n!= 0} {n!=1}`, putting one constraint on top of another.

We usually use commas, but it doesn't matter.

(You can't say "or" or "not" in constraints at all. `x||y` and `!x` are perfectly good anywhere else, but they're not allowed in constraints.)

#### Unrelated Constraints

There's no requirement that the constraint involves `self`, or that a constraint on a variable `v` has anything to do with `v`. [19]

```
1 val a = 1;
2 val b = 2;
3 val v : Int{a!=b} = 3;
4 //ERROR: val d : Int{a==b} = 4;
```

The constraint on line 3 says that `v` is an integer such that the `val`s `a` and `b` are different. Well, looking at the code, `a` and `b` are indeed different. So the constraint `a!=b` is always true, no matter what `Int` value `v` has. In other words, `v` can be any `Int` value.

On the other claw, if the constraint *isn't* true, then *no* value works. You can't even use the type `Int{a==b}` on line 4 — or the type `Int{false}` anywhere. No matter what value you assign to `d`, it won't make `a==b`, and won't make `false` be true.

These unrelated constraints are a way of enforcing that certain facts are true at *compile time*. `assert` statements, like

```
assert a != b;
```

can check that `a` and `b` are different at *runtime*. But if you need to know that they're different at *compile time*, you'll need to put the fact into a constraint.

(Why might you need to know that? Some other constraint might use it. If you have a method `recipDiff(x,y)= 1.0/(x−y)`, and you're calling `recipDiff(a,b)`, you might need to persuade X10 that `a!=b`.)

### 6.1.6 Using Properties

The *properties* of a class or struct are the values that X10 keeps the closest track of at compile time. They can be used in constraints on a class. Let's do a concrete example: points and triangles in plane geometry. We'll call the points `Pt` to keep them from getting confused with `x10.array.Point`. Here it is: [20]

[19] See file ConstraintExamples.x10
[20] See file Geometry.x10

```
1 struct Pt(x:Double, y:Double) {
2   public operator this - (that:Pt) = Pt(this.x-that.x, this.y-that.y);
3   public def len() = Math.sqrt(x*x-y*y);
4 }
```

**line 1:** `x` and `y` are properties of `Pt`. **(B: Do we discuss this elsewhere? :B)** Making them properties like this, and not giving any other constructor, means that X10 will automatically give us the obvious constructor `Pt(xx,yy)`.

**line 2:** This line defines a binary operator of subtraction on `Pts`. We're treating them as two-dimensional vectors. This isn't essential, but it does make using them very slick.

**line 3:** This is a perfectly ordinary method, giving the length of a vector.

The code for triangles will use a constraint to ensure that all triangles are non-degenerate — that they all have three distinct points. [21]

```
1 struct Triangle(p:Pt, q:Pt, r:Pt) {p != q, p != r, q != r} {
2   public def area() {
3     val a = (p-q).len();
4     val b = (q-r).len();
5     val c = (r-p).len();
6     val s = (a + b + c)/2;
7     val area = Math.sqrt(s * (s-a) * (s-b) * (s-c) );
8     return area;
9   }
10 }
```

Triangles have three `Pts`, which we're phrasing as properties. The `Triangle` class has a guard on line 1, saying that `p`, `q`, and `r` are all different. (Obviously this approach is going to get pretty troublesome by the time we get to heptagons, where we'd have to write 7 variables and 28 non-equalities, and won't work at all if we're trying to use arbitrary $n$-gons. But it's a very slick approach when it does work.)

**Fine Point:** We rather sneakily made `Pt` a struct rather than a class. This matters because == means different things for structs and objects. For structs, `p==q` is true if `p.x==q.x` and `p.y==q.y`. For classes, `p==q` is true if `p` and `q` are the same object. If `Pt` had been an object, we could have a `Triangle` with three corners each made by a call to `new Pt(0,0)` – three `Pts` in the same place, but different objects.

[21] See file Geometry.x10

### 6.1.7 Nulls and Constraints

Sometimes you want to forbid `null` values in your object types. You can do this with the constraint {`self` != `null`}.

If you don't have that constraint on an object type, though, `null` is allowed. And that goes even if you do have other constraints – *any* other constraints. The point of this is that it's useful to have `null` as a default value for object types, so you get to have it unless you specifically say you don't.

So, the following program is fine: [22]

```
1 class Trunk(length:int){}
2 class Elephant(trunk:Trunk){}
3 def example() {
4  val dumbo: Elephant{self.trunk.length == 3} = null;
5  assert e == null;
6  val jumbo: Elephant{self.trunk.length ==11} = null;
7 }
```

This makes for some surprises about constraints. At least, they're surprising if you are under the mistaken impression that =='s in constraints mean the same thing they mean everywhere else — they don't! The difference is that, in a constraint, `a.f==b` and `a.f!=b` both mean `true` if `a==null`. Outside of a constraint, it means "null pointer exception", neither true nor false.

One way you could get confused, if you don't keep this special meaning in mind, is that `dumbo : Elephant{self.trunk.length == 3}` does **not** mean that `dumbo.trunk.length` is three! It means, "**If** `dumbo!=null` and `dumbo.trunk!=null`, **then** `dumbo.trunk.length` is three." Or, to phrase it another way, "If `dumbo.trunk.length` means anything at all, what it means is three."

Another way to get confused is to notice that that `dumbo` and `jumbo` have types that look like they couldn't possibly have any values in common. One is for short-trunked elephants, the other for long-trunked ones, and there's no way that the same elephant's trunk could be both three and eleven feet long.

Well, of course there's no such elephant. `null` isn't an `Elephant`. It's a non-value that can be used in place of an `Elephant`. And `null.trunk.length` isn't three, or eleven, or any other number – it's an error.

All you need to remember about this is three things:

- `null` is allowed in constrained object types, unless you specifically say it's not.
- Constraints on fields get fudged so that they're true for `null`, and don't throw null pointer exceptions.

[22] See file ConstraintsAndNulls.x10

- You need to check for `null`s when you use constrained object types – the same way you need to when you use ordinary object types. (Unless you constrained them to never be `null`.)

### 6.1.8 Constraints and Subtyping

### 6.1.9 STATIC_CHECKS

### 6.1.10 Incompleteness

### 6.1.11 Why Generics Lose Constraints At Runtime

## 6.2 Type Declarations

**(B: do this – note non-null type decl for niceifying Array[Stringself!=null]self!=null :B)**

## 6.3 Type Inference

**(B: do this! :B)**

## 6.4 Generics

**(B: do this! :B)**

## 6.5 Default Values

**(B: do this! :B)**

## 6.6 Common Ancestors of Types

**(B: do this! :B)**

## 6.7 When Types Don't Work

**(B: do this! :B)**