



SMARTARRAYS[®]
ARRAY TECHNOLOGY FOR BUSINESS ANALYTICS

Working with SmartArrays

James G. Wheeler
SmartArrays, Inc.

June 2006

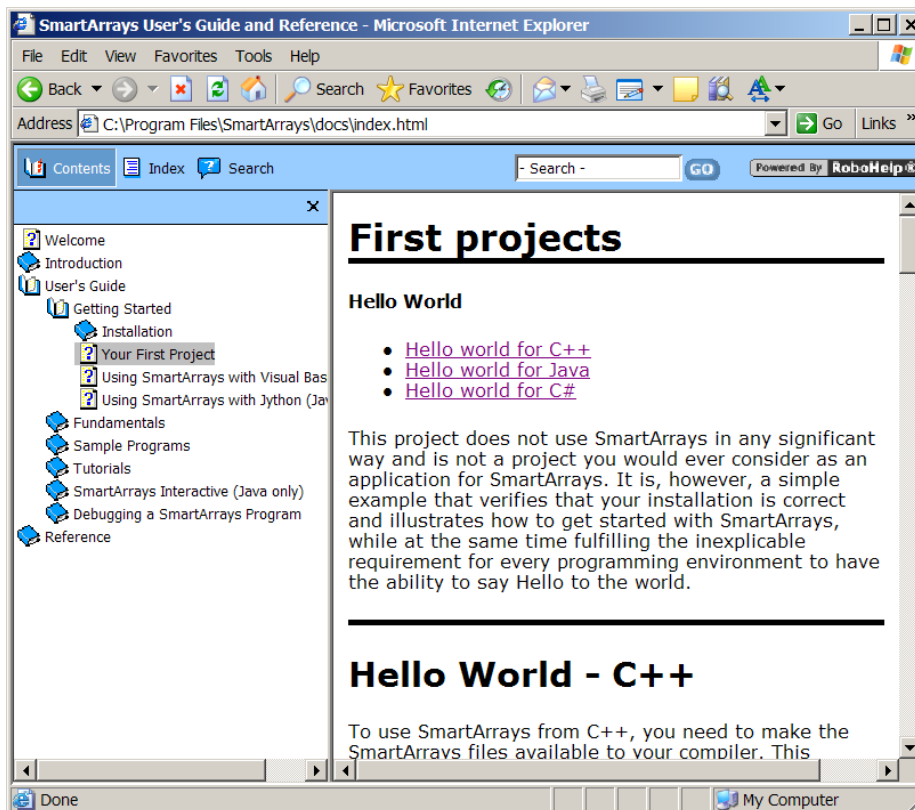
Working with SmartArrays – About this Tutorial

This tutorial is intended to help programmers rapidly develop a working understanding of SmartArrays. The associated short course is aimed at programmers who have at least a little experience with one of the languages supported by SmartArrays (Java, C#, or C++) and who have little or no prior experience with SmartArrays.

What you'll need to work through the tutorial on your own:

- A copy of the SmartArrays SDK. We have included limited-time evaluation versions of SmartArrays for Windows on the accompanying CD that you can use if you do not already have a licensed copy of the product.
- A development environment for the language of choice. The examples in this tutorial are written for C# and Visual Studio but Java or C++ can be used instead. If you do not already have a suitable development platform, we suggest that you download one of the following:
 - C#: Microsoft Visual C# Express Edition – beta versions currently available for free. at <http://lab.msdn.microsoft.com/express/vcsharp/>
 - Java: Eclipse (www.eclipse.org) or NetBeans (www.netbeans.org) are good IDEs for Java development
- The sample tutorial code and data used with the tutorial. These can be found on the CD.

The SmartArrays Manual is an HTML document that is installed on your computer when you install the SmartArrays SDK. It is also available online from the SmartArrays web site. It contains a Users Guide covering basic information on using SmartArrays and a complete Reference manual that describes each of the 250+ array operations.



Lesson 1: Set up a SmartArrays Project

For the classroom lectures we will use Visual Studio and C# simply because this is one of the most popular and easy to use languages that work with SmartArrays. If you prefer to work with other languages, the User Manual provides instructions for setting up projects in other environments. No matter which language you work with, there are two parts to setting up a project to work with SmartArrays.

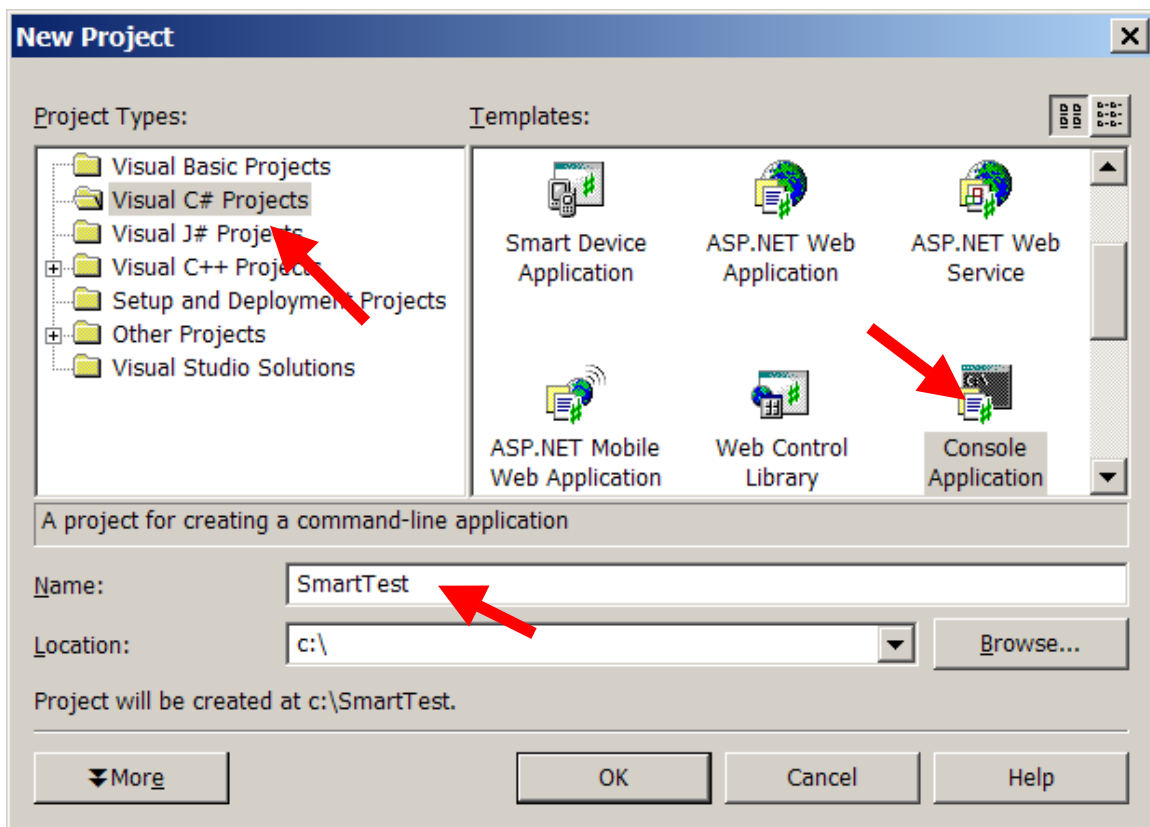
1. Add the SmartArrays library to the project. In .NET this means **add a reference** to the SmartArraysV4.dll to the project; in Java, **add the SmartArraysV4.jar** to your list of libraries; in C++, add **SmartArraysV4.lib** to the list of link libraries.
2. Add a reference to the **SmartArrays classes** in every source code file that works with SmartArrays objects. In C#, it's a **using** statement; in Java an **import** statement; and in C++ a **#include** statement.

You'll find details on Java or C++ project setup in the User Manual.

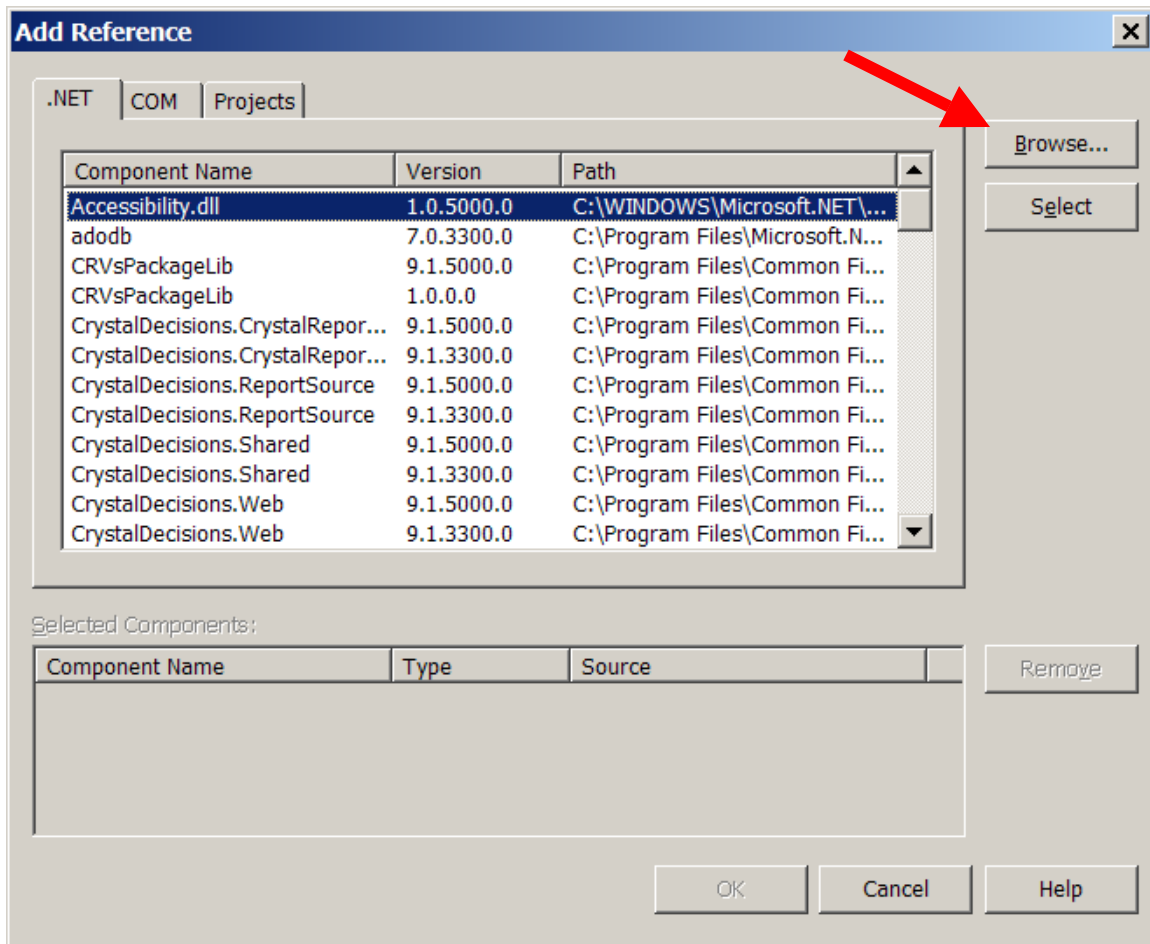
You'll note that there's nothing unusual about this process. These are the same things you have to do to work with any other class library.

Details: Using SmartArrays in a C# Project Visual Studio .NET Project

1. Start Visual Studio.NET. This example uses Visual Studio.NET 2003 but these steps are the same for the original VS.NET 2002 and VS.NET 2005. Create a new C# console application.

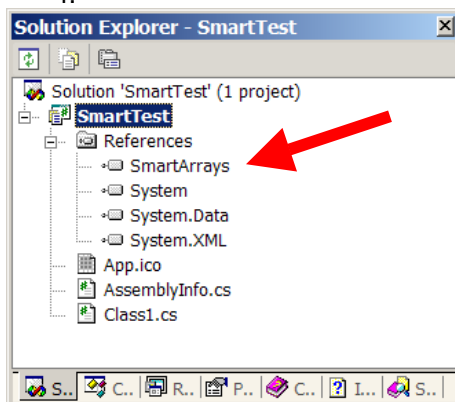


2. Locate the Solution Explorer window in Visual Studio. In the tree, under the SmartTest project, you will see a References folder. Right click on it and select Add Reference... The following dialog appears



3. In the Add Reference dialog, click the Browse... button and navigate to your Windows directory. Locate the file SmartArrays.dll and double click on it, then click OK to close the Add Reference dialog. "SmartArrays" will then appear in the list of references in Solution Explorer along with "System", "System.Data". etc.

4.



5. Now we can write some code. Unless you overrode this, Visual Studio created a source file named Class1.cs in your project. This contains the Main() function that is the entry point for a console (command-line interface) program. The code looks more or less like this:

```
using System;

namespace SmartTest
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //
        }
    }
}
```

6. You need to add the line
using SmartArrays;

to the top of the file below using System;. Then edit the Main() function to create SmArray objects and work with them. For example:

```
static void Main(string[] args)
{
    SmArray x = SmArray.sequence(100);
    System.Console.WriteLine(
        "SmartArrays says the sum from 0 to 99 is "
        + x.reduce(Sm.plus) );

    // Wait for user to press Enter (so the command window will
    // stay visible until you read its contents)
    System.Console.ReadLine();
}
```

Press F5 to compile and run your console application. You'll see this line in the console window.

```
SmartArrays says the sum from 0 to 99 is 4950
```

Congratulations! You've built and run your first C# program with SmartArrays. The next page has the entire source file.

```

using System;
using SmartArrays;

namespace Tutorial
{
    /// <summary>
    /// Sample Console Application for SmartArrays
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {

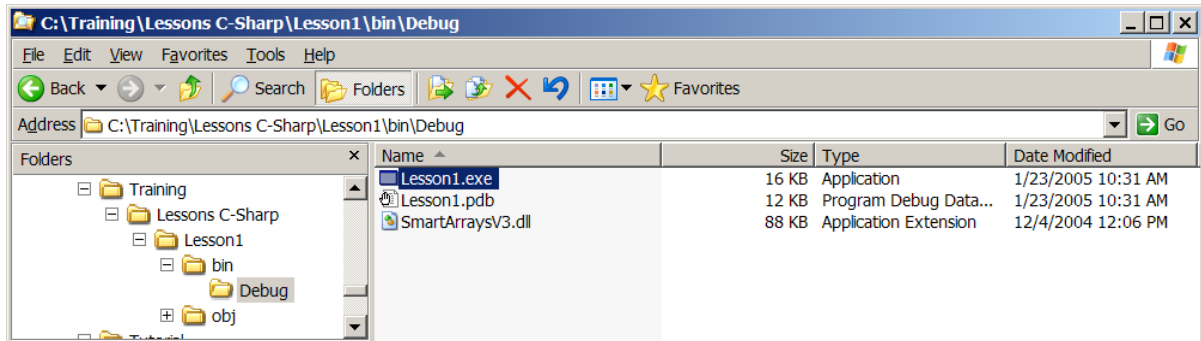
            SmArray x = SmArray.sequence(100);
            System.Console.WriteLine(
                "SmartArrays says the sum from 0 to 99 is "
                + x.reduce(Sm.plus) );

            // Wait for user to press Enter (so the command open will
            // stay visible until you read its contents)
            System.Console.ReadLine();

        }
    }
}

```

The C# compiler creates a program that's ready to run in the bin\debug subdirectory. Look for it in Windows Explorer. You can double-click on **Lesson1.exe** to run the program, which runs in a Console window (like DOS programs of long ago).



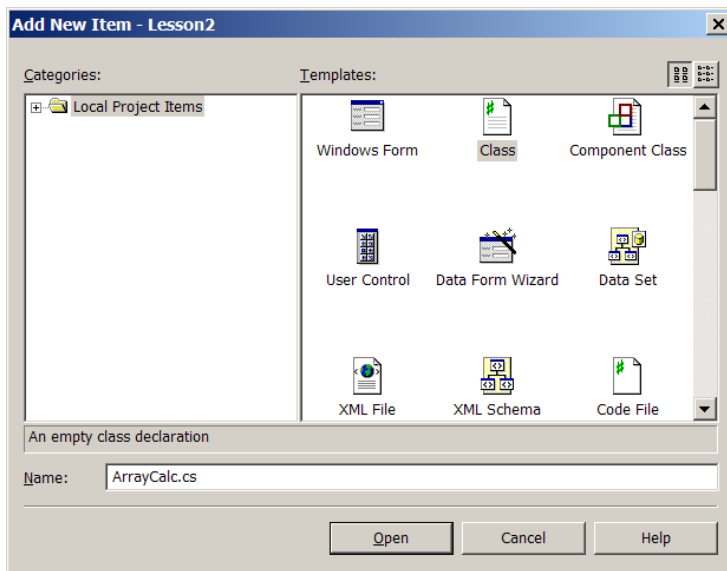
Notice that Visual Studio copied the SmartArrays .NET class library SmartArraysV4.dll to your project's bin\Debug directory. This is Visual Studio's default behavior, intended to help prevent your program from being separated from the library it needs. Making a local copy is optional, though not much to worry about because the file is not very large. See the Visual Studio manual for more details on this.

Lesson 2: Write a Bit of SmartArrays Code

In this lesson we will build on the simple “console” application we used in Lesson 1. The procedure for creating the project is the same as before. But this time we will create a new class that uses SmartArrays to generate some data and do some calculations on it. You will find the finished project in the Lesson 2 directory.

To begin, create another new console application and add the SmartArrays reference just like before.

To this project we will add a new class named ArrayCalc. In Visual Studio, use the Project>Add Class menu, and type “ArrayCalc.cs” as the class name:



Visual Studio will create the file, initialize it based on some assumptions, and open it in the editor like this:.

```
using System;

namespace Lesson2
{
    /// <summary>
    /// Summary description for ArrayCalc.
    /// </summary>
    public class ArrayCalc
    {
        public ArrayCalc()
        {
            //
            // TODO: Add constructor logic here
            //
        }
    }
}
```

Edit this file to add the SmartArrays classes to the ones you’ll be using by adding the line

```
using SmartArrays;
```

Now edit the class to add a member variable that is a SmartArrays object:

```
public class ArrayCalc
{
    protected SmArray _randomvalues; // member variable
```

Now we will add some public methods to the class. First is one to create a value for the member variable `_randomvalues`:

```
// Create some random data and hold it in a member variable of the class.
// The array will continue to exist as long as this instance of the class
// exists. When the class is finalized (garbage-collected), the array will
// be deleted.
public void generateRandomData()
{
    int count = 1000000; //
    int max = 50; // Use values in the range from 0..49

    // a million copies of 50
    _randomvalues = SmArray.scalar(max).reshapeBy(count);

    // generate uniform distribution of values 0..49
    _randomvalues = _randomvalues.roll();

    // roll() gets its name from rolling dice. The value in the array
    // is the number of sides of the die, so the above statement is like
    // rolling a million dice each with 50 sides.
}
```

Next is a method to display the first 20 values of the array. We don't want to see all 1,000,000 values!

```
public void showFirstTwenty()
{
    _randomvalues.takeBy(20).show();
}
```

Next is one to count the number of values in a range (\geq a minimum value and $<$ a maximum value)

```
public void countInRange( int min, int max )
{
    SmArray bitvector = _randomvalues.ge(min);
    bitvector = bitvector.and( _randomvalues.lt(max) );
    bitvector.reduce( Sm.plus ).show();
}
```

Don't worry about understanding this code yet, we'll get to that soon. The point is to show how to put together a class that uses arrays. Now let's add some code to the applications `Main()` function to call this class:

```
class Class1
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        // Create an instance of the ArrayCalc class - see ArrayCalc.cs
        ArrayCalc calc = new ArrayCalc();
```



```

    // Call its method that generates its member variable of random values
    calc.generateRandomData();

    // Call its method that displays the first part of the array
    calc.showFirstTwenty();
    System.Console.ReadLine();

    // How many of the numbers are between in the range 10..19?
    calc.countInRange(10,20);
    System.Console.ReadLine();

    // How many between 20 and 30?
    calc.countInRange(20,30);

    // Wait in the console window before closing
    System.Console.WriteLine( "Done. Press Enter to close window" );
    System.Console.ReadLine();
}
}

```

Start the program in the debugger by pressing F5 and press enter to move to each step:

```

C:\Training\Lessons C-Sharp\Lesson2\bin\Debug\Lesson2.exe
6 37 22 26 10 2 33 33 46 19 25 41 1 2 26 33 0 19 3 20

199606

200567

Done. Press Enter to close window
_

```

Lesson 3: What's Going On Here?

Some tutorials of new technology simply lead you through a series of complicated examples without explaining what's going on. It's like being taken by the hand, led into a dark forest, and left there with no map. I will try not to do that to you here.

Now that we've gotten SmartArrays working, let's step back and look at the architecture of SmartArrays. Effective use of SmartArrays requires an intuitive understanding of what happens when you create and work with array objects.

The SmartArrays Components

Whichever version of SmartArrays you're using, there are two major pieces:

- The SmartArrays array engine. This is a library file named "com_smartarrays_engineV4.dll" in Windows. In Unix or Linux it's has the extension ".so". The "engine" is identical whether you are programming in C#, Java, C++, etc.
- The SmartArrays class wrappers: These define the "Wrapper" classes that call the engine. They are written in the target language and are provided in the standard form for a class library of that language. In our sample C# project, they are packaged in a .NET "assembly" file named "SmartArraysV4.dll".

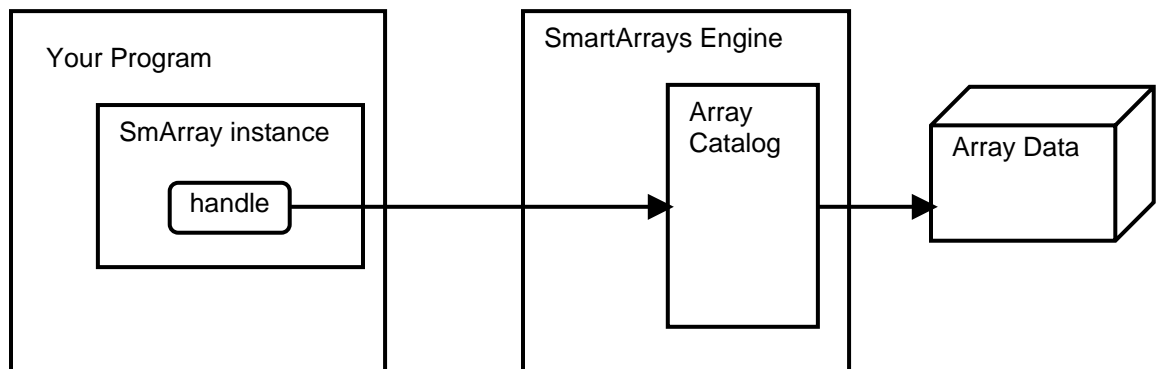
The wrapper class library defines the array class SmArray and a few other classes that you'll need to work with SmartArrays. We'll take a closer look at SmArray in the subsequent lesson.

Where Is the Array?

In the previous lesson we created an SmArray object as a member variable of a class. If we inspect the SmArray object in the debugger we'll just see a single number named **mHandle** in the object. So where's the array?

The explanation is that mHandle is a "handle" for an array – a number that identifies the array to the SmartArrays engine. The engine creates and maintains a catalog of all the arrays that are currently in use, using the handle to identify which one you are working with.

The actual data that holds the array is allocated by the engine, which keeps the memory so long as the array handle is in use. When an SmArray object is deleted, the engine releases the memory back to the operating system.



Array References

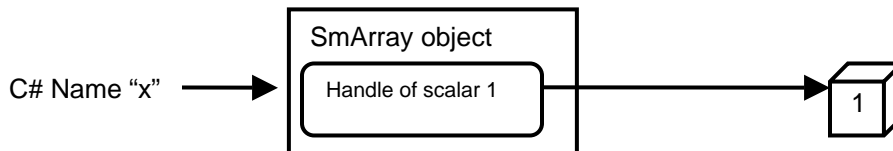
The handle in an SmArray is associated with a reference to an array. There can be more than one handle referring to the same physical array. The engine keeps track of **reference counts** on each array. Class SmArray defines a finalizer or destructor method that runs when that instance of the array is deleted. This method notifies the engine to release its reference to the array.

When the last reference to an array is released (its reference count goes to zero), the engine deletes the storage. This will happen when the SmArray object goes out of scope and is garbage collected, or when your program ends.

Lifespan of an SmArray

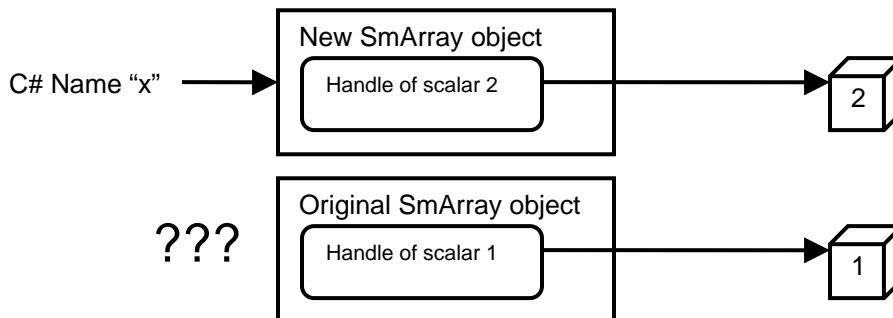
The life of an array is tied to the life of the SmArray object that holds its handle. That object can be bound to a name in your programming language.

```
SmArray x = SmArray.scalar( 1 ); // declares and initializes an array
```



Now let's assign a new array to the name "x":

```
x = SmArray.scalar( 2 ); // new SmArray object bound to the name
```



What happened to the first SmArray? There is no name bound to the object, so it's "garbage". In C# and Java the **garbage collector** periodically looks at all the objects it has created to see which ones no longer are bound to names. When it finds them it deletes them, but first it calls their destructor to tell them they are being demolished. At this point the SmArray object tells the engine that it's giving up the handle it contains. This tells the engine to delete the physical array.

In C++, destructors get called immediately when an object does out of scope. But in Java and the .NET languages, garbage collection runs on a separate thread. At some point the garbage collector will notice that the original SmArray object isn't anchored anywhere and delete it. It's important to understand the **latency** of garbage collection when you start using really huge arrays.

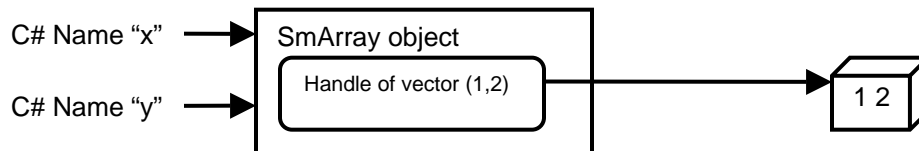
You may also need to learn about the **release()** method of class SmArray and also how to explicitly call the garbage collector, but we will not cover those topics here.

Watch Out in Java and C#:

Consider the following code.

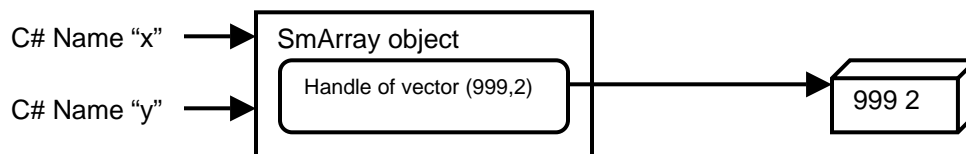
```
SmArray x = SmArray.vector(1,2);  
SmArray y = x;
```

What's the state of the arrays after these statements? In C# and Java this means that the two names "x" and "y" refer to the same SmArray object. There's only one SmArray object and that's all the engine knows about:



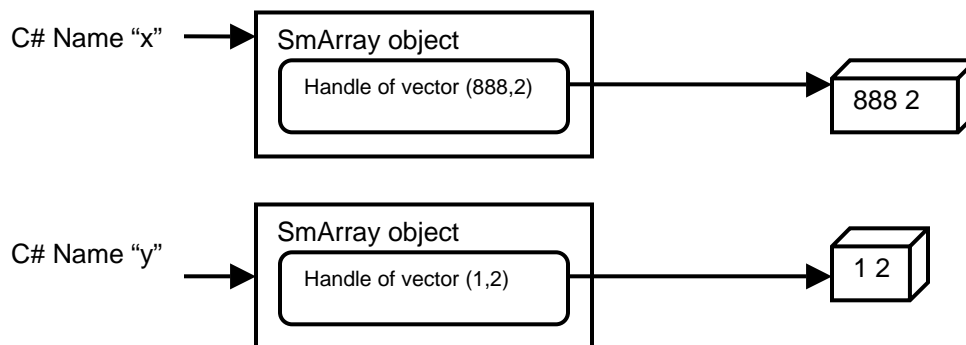
This is fine. C# and Java allow multiple names to refer to the same object. But **watch out** if you modify the values inside the SmArray without creating a new one:

```
x.setInt( 999, 0 ); // replace the value at position 0 with 999
```



As you can see, this changes the value inside the array for both **x** and **y** because they are the same SmArray object. If you assign one SmArray variable to another, and don't want them to refer to the same object, you can use the copy() method. This creates a new array inside the engine and a new SmArray object in your program that are distinct.

```
SmArray x = SmArray.vector(1,2);  
SmArray y = x.copy();  
x.setInt(888,0); // modifies x but not y
```



The sample project Lesson3 accompanying this tutorial shows this behavior in action if you want to experiment. This is not an issue in C++ because that language has the concept of a *copy constructor*, which allows the assignment $x = y$ to force the creation of a distinct object. But for garbage-collected, reference-passing, languages like Java and C# you need to be aware of this behavior.

Don't think, by the way, that this issue is unique to SmartArrays. It is how languages like C# and Java work and **any** object of any class has the same issues.

Lesson 4: The SmArray Class

SmartArrays is based on a small set of simple ideas, applied with strict consistency across the whole range of array operations. Once you've absorbed these concepts you will be able to use SmartArrays with ease because each of the 250+ methods will behave in ways you expect.

One Class for All Arrays

One essential concept is that every SmartArrays array, no matter what its size or what kind of data it contains, is an instance of the SmArray class. Once you declare a name in your program to be of type SmArray, you can assign any array to that name.

```
SmArray a;  
a = SmArray.sequence(10); // sequence of 10 numbers starting from zero.  
a = SmArray.vector("Now", "I", "am", "five", "strings");
```

Static Methods

A static method is a function of a class that can be called by referring to the class; you do not need an instance of the class. Static methods are the way to create an array from scratch. Both of the lines above are examples of initializing an array with a static method.

Examples of static methods for creating arrays:

- **scalar(value)** creates a scalar (defined below)
- **vector(value, value, value)** creates a vector from literal values
- **array(object[])** creates an array out of a native C#, Java, etc array
- **sequence()** creates an array from a series of numbers
- **empty()** creates an array initialized to an empty array.
- **fileRead()** creates an array with data from a file

The static methods all return new SmArray objects as their results.

Important: Although the class SmArray has a **new()** method, you are unlikely to ever use it. It will return a *null array* which will produce an error if you try to do anything with it. **SmArray.new()** exists only because a class needs a default constructor in certain situations such as creating a native array of SmArray objects(SmArray[]). In practice, arrays are created by the static methods above or, more typically, by the non-static methods described in the next section that return arrays as results.

Non-Static Methods

Non-static methods, also called "instance" methods, require an instance of the class SmArray. These methods are a command for the object to do something. An example of an instance method is **reverse()**, which produces an new array in reversed order.

```
SmArray v = SmArray.vector(100,200,300);  
v.reverse().show();
```

The last line above produces a line of output that shows the new array produced by reversing the original array, like this.

```
300 200 100
```

The output from display methods like `show()` appears in the program's output stream. From this point onward in this document, we will show this output as shaded text below the expression that produces it.

If we want to replace `v` with its reversed value, we assign the result of `reverse()` back to `v`:

```
v = v.reverse();  
v.show();  
300 200 100
```

Other array operations need two or more arrays to perform. For example to add two arrays together, we call the **plus()** method of an array and pass another as an argument

```
v.show();  
300 200 100  
v2.show();  
30 40 50;  
v.plus(v2).show();  
330 240 150
```

In statements like this, the array whose method is called (`v` in this case) is called the **subject array**. Any arrays passed inside the parentheses are called **parameters**.

Most of the array methods follow this syntax pattern. They operate on the subject array, with one or more other arrays as parameters, and return a new array as the result. This functional syntax makes it possible to string array operations together into a single expression, often accomplishing a huge amount of work in a single line of code.

If you have variables **a**, **b**, **c**, and **d**, what would you think this does?

```
SmArray e = a.plus(b).plus(c).plus(d);
```

You probably guessed right. This line of code starts with the array **a**, and adds array **b**, producing a new `SmArray` as the result. Since the result is itself an `SmArray`, you can call *its* `plus()` method with an argument of **c**, producing yet another array, whose `plus()` method is called with **d** as its argument, producing yet another array, which gets assigned to the variable **e**.

Operator Overloads (C++ and C# Only)

Some languages provide “operator overloads” – or the ability to redefine the behavior of common operators like “+” for a certain class. The `SmArray` classes for C# and C++ have these overloads, allowing you to write the last example above like this:

```
SmArray e = a+b+c+d;
```

This is just a shorthand way of writing arithmetic expressions – it does exactly the same thing as calling **plus()**. This feature is described in the manual but I will not use it in these lessons because I think the functional syntax helps you better visualize the computation being done. Also it's not available for all languages. Java, for example, simply doesn't permit operator overloading.

Lesson 5: Properties of Arrays

Since a single class `SmArray` is used to represent every possible array object, knowing that an object is an instance of class `SmArray` doesn't tell us much about it. To learn the properties of an array, we use various descriptive methods. This lesson will focus on the properties of an array and the methods used to inquire about them.

What Can We Store in Arrays?

An array created by `SmartArrays` has three essential characteristics:

- its **shape**
- its **type** - what sort of values it contains
- The values themselves

Shape of Arrays

Every array has a shape – its length along each of its dimensions. The **rank** of an array is the number of dimensions it has.

- A **scalar** is an array with no dimensions, just a single value. Its rank is zero.
- A **vector** is an array with one dimension, i.e. as rank of 1.
- A **matrix** has two dimensions
- Arrays can have 3 or more dimensions – the limit is 255 dimensions -- though arrays with rank higher than 3 are rare in practice.

The length of a dimension can be any value from 0 up to the system limit, assuming you have enough memory to hold the array values.

The shape of any array can be described as a vector of positive numbers, and the method **shape()** returns just such a *shape vector* for its subject array. For a scalar, **shape()** returns an empty vector because it has no dimensions.

For example, given some array `x`, we can determine its rank and its shape thus:

```
SmArray xrank = x.rank();
xrank.show();
2
SmArray xshape = x.shape();
xshape.show();
49212 12           It has 49,212 rows and 12 columns
```

The **rank()** and **shape()** methods return their results as `SmArray` objects. Often it is more useful to get the results as native types. For these cases there are the methods **getRank()**, which returns its result as an **int**, and **getShape()**, which returns an **int[]** native array.

```
int xrank = x.getRank();
int[] xshape = x.getShape();
```

Type of Arrays

An `SmArray` object can contain data values from a fixed set of types shown in the table below.

Numeric	Native Type	Size Per Item	Description
dtBoolean	bool	1 bit	True/false values;"bitmaps"
dtInt	int	4 bytes	A signed 32-bit integer
dtDouble	double	8 bytes	Double precision floating point
dtComplex	n/a	16 bytes	Pair of doubles representing the real and imaginary parts of a complex number.
Character	Native Types	Size Per Item	Description
dtByte	char (C++), byte (C# and Java)	1 byte	One byte character
dtChar	wchar (C++), char (C# and Java)	2 bytes	Unicode character
Other	Native Types	Size Per Item	Descriptions
dtString	string	4 bytes	Stored as a 4-byte string reference to the string table.
dtNested	--	4 bytes	An array as an item of an array
dtMixed	--	16 bytes	Used when an array contains a combination of the above types.

Some of the array methods work on all types of data, specifically the ones that select from or restructure arrays. Others – the arithmetic methods for example – require that the array be entirely numeric.

Using `showDebug()` Instead of `show()`

Up to now I've used the method `show()` to display the contents of an array. In practice, however, you will want to use `showDebug()` instead. `show()` is only suitable for small arrays; on large arrays it can take a long time to run. Furthermore, it does not tell you much about the array's shape or type. By contrast, `showDebug()` produces a single line of output that contains the array's shape, type, and the first few data values.

String Data and the String Table

SmartArrays provides both arrays of characters, where each item is a single character, and arrays of strings, which treat a character string as an atomic data value. Both have their uses, but strings have special properties that make them particularly valuable and fast in common situations.

Inside the SmartArrays engine is a **string table** that holds the set of all unique strings currently used in arrays. Every string item in an array is a **reference** to a specific string in this table, and each specific character sequence only occurs once in the table. The string in the array is represented indirectly by a 4-byte string ID which can be used to locate the character sequence in the string table.

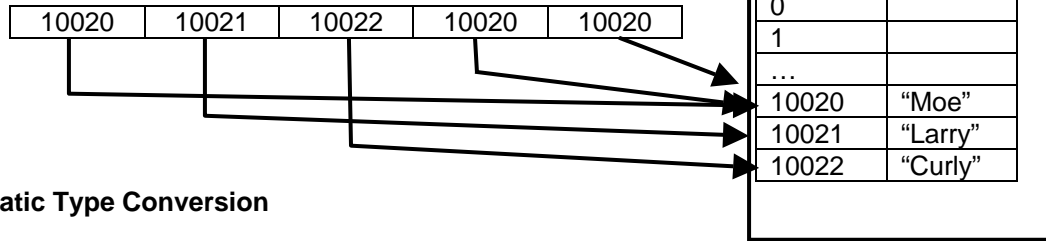
Representing strings in this way has a number of valuable properties. If, for example, the same string is used many times, in one array or multiple arrays, only one copy of the actual character sequence exists. All of the occurrences of that string will be represented by the same 4-byte string ID.

So, if we want to determine if two strings are equal, SmartArrays can do this just by comparing the 4-byte values. This is much faster than comparing every character in the two character sequences.

Suppose we create the following vector of strings, where

```
SmArray names = SmArray.vector( "Moe", "Larry", "Curly", "Moe", "Moe" );
```

The string vector itself holds 4-byte string references, which locate the actual characters in the string table.



Automatic Type Conversion

The type of data stored in an SmArray object is selected automatically based on the values that are stored in it or the process by which it was produced. The array engine will **promote** a data type where appropriate to produce a correct answer.

One example of promotion is arithmetic operations. If you add two integer valued arrays together, there's always the possibility that the result will be a number that doesn't fit in a 32-bit integer. SmartArrays tries to produce an array of 32-bit integers as the result but watches for overflows and, if one occurs, returns the result in floating-point type (dtDouble) instead.

If you "glue" two arrays together with one of the concatenation methods, the result array will be the most compact type that's capable of representing all the values in each array. Likewise, if you replace a value in an array with a new value that cannot be represented in the array's internal type, the array will be automatically promoted to a type that can hold the new value.

Examples:

```
SmArray intvec = SmArray.sequence(5);
intvec.showDebug();
I[5]0 1 2 3 4 5

SmArray doublevec = intvec.times(1.5);
doublevec.showDebug();
F[5]0 1.5 3.0 4.5 6.0 7.5

intvec = intvec.catenate( doublevec );
doublevec.showDebug();
F[10]0.0 1.0 2.0 3.0 4.0 0 1.5 3.0 ...

names.showDebug();
S[5] "Moe" "Larry" "Curly" "Moe" "Moe"

// Replace one of the strings with an integer
names.indexInto( SmArray.scalar(-5), 1 );

// Now the array is dtMixed type:
M[5] "Moe" -5 "Curly" "Moe" "Moe"
```

Lesson 6: Array Methods

There are more than 250 methods in SmartArrays, but their behavior is very uniform. If you understand a few general patterns and principles, you will find that you can pick up new methods easily because most of their behavior is what you expect.

Families of Methods

The SmartArrays Reference Manual groups methods into categories. All methods fall into two top level categories:

- **Native data methods** that take arguments or return results as native data.
- **Array methods** that work on SmArray objects and produce new SmArray objects as results.

The native data methods allow your other code to interact with SmartArrays and can be grouped into three categories:

Native Data Method Family	Examples
Create new arrays from native data	scalar(), vector(), array(), ...
Fetch descriptive information about an array	getType(), getShape(), getRank(), ...
Retrieve or replace data values in an array	getInt(), getInts(), setInt(), setInts(), ...

The Array methods that work on arrays and produce array results also fall into categories:

Array Data Method Family	Examples
Arithmetic	plus(), divide(), recip(), floor(), log(), ...
Trigonometric	sin(), cos(), arcsin(), ...
Comparison	lt(), gt(), ne(), match()
Logical	and(), or(), not()
Mathematical	roll(), matrixInverse(), decode()
"Mixed" (Metadata)	shape(), rank(), depth()
Selection and Replacement	row(), index(), compress(), pick(), pickInto()
Sorting and Searching	grade(), lookup(), member(), find()
Structure	catenate(), reshape(), ravel(), rowCat()
Composite	reduce(), inner()
Statistical	mean(), median(), standardDev()
Data and Time	dateToJulianDay()
Formatting, Conversion, and Files	format(), cast(), coerce(), toXML(), toFile()
Debugging and Tuning	show(), showDebug(), getTickCount()
String and Character	strReplace(), strTrim(), deb(), lower()...
XML	xmlParse(), xmlIndex()
File I/O and Memory-Mapped Files	toFile(), fileArray(), directory()...
Reflection	enumMethods(), apply()

Domain Rules for Array Methods

Some array methods – the ones that restructure data – do not care what the values of the data items are. Others do. Arithmetic only makes sense on numbers, so SmartArrays enforces that rule. If you try to do arithmetic on a character value, SmartArrays will produce an error. (See Exceptions, below).

```
SmArray.scalar(0).plus( SmArray.scalar( "one" )); // produces an SA_DOMAIN error
```

In other words, the string “one” is not in the domain of meaningful values for the plus() function. SmartArrays is strict about enforcing domain on every value in an array. For example, it is an error to divide by zero. If you divide two arrays and there’s a zero somewhere in the divisor, the **divide()** method will produce an SA_DOMAIN error.

Conformability Rules and Scalar Extension

Most array operations require the subject array and parameter arrays to “fit” together. If you try to add two 10-item vectors together, you will get a new array that contains the pairwise sums of the values. But if one vector has 10 items and the other has 9, the arrays don’t fit, and they can’t be added together. In cases where the shapes of arrays do not fit, SmartArrays will produce an SA_LENGTH error. If the arrays have an inappropriate rank, it will be an SA_RANK error.

Scalar arrays get more tolerance. In many operations, a scalar can be used in place of a conforming array. The scalar “extends” to conform to a higher-rank array. For example:

```
SmArray v = SmArray.( vector( 100, 200, 300 ) );
v.plus( SmArray.scalar(1) ).show();
101 201 301
```

Native Values as Implicit Scalar Arrays

You can pass a native string, number, or boolean value as the parameter of an array method in place of a scalar SmArray. This makes for more readable code. The array wrapper will create the SmArray scalar for you and pass that to the engine.

For example, the statement above could have been written:

```
v.plus( 1 ).show();
101 201 301
```

Composite Methods and the Sm Class

One class of array methods deserves special note. These are the **composite** methods that take array methods as arguments. How do we pass a method as an argument? Actually we don’t. Instead these methods take a special number that indicates which method we want. The class Sm is defined by the SmartArrays wrapper just like SmArray. It is a collection of named constants for each of the method codes. For example, **Sm.plus** is the code for the method **plus()**.

This lets us modify or combine the behavior of array methods in useful ways. For example, the method **reduce()** applies an arithmetic, comparison, or logical method between the values of a vector to produce a scalar:

```
SmArray v2 = SmArray.vector(3,4,5);
v2.show();
3 4 5

SmArray v2sum = v2.reduce( Sm.plus ); // i.e. 3+4+5
v2sum.show();
12
```

Lesson 7 - Essential Array Methods

SmartArrays provides hundreds of array methods, but of these many are not used very often. For example, I have personally never yet used hyperbolic arc-cosine in a program except when writing a test script for **acosh()**.

In practice, there are a few dozen methods that get used all the time. We will focus on those here.

Reshape Family

These are the methods for building or restructuring arrays. One of most common ways to build an array is to reshape the data of another array.

```
// Reshape a scalar into a big array
SmArray bigvector = SmArray.scalar(0).reshapeBy( 1000000 );
bigvector.showDebug(); // (don't use show() on anything this large!)
I[1000000] 0 0 0 0 0 0 0 0 0 0 ...
```

```
// Alternative form - the subject array is the shape
SmArray.scalar( 1000000 ).reshape(0).showDebug();
I[1000000] 0 0 0 0 0 0 0 0 0 0 ...
```

```
// If you supply more than one value, the data is repeated
SmArray v = SmArray.vector(1,2,3);
v.reshapeBy(10).show();
1 2 3 1 2 3 1 2 3 1
```

```
// You can reshape the data into a different rank
SmArray m = SmArray.sequence(12,1);
m.show();
1 2 3 4 5 6 7 8 9 10 11 12
```

```
m = m.reshapeBy(3,4); // 3 rows and 4 columns
m.show();
1 2 3 4
5 6 7 8
9 10 11 12
```

The **ravel()** method is the opposite of **reshape()** - it returns all the values as a vector

```
m.ravel().show(); // turn a matrix into a vector
1 2 3 4 5 6 7 8 9
```

ravel() also turns a scalar into a 1-item vector

```
SmArray.scalar(9).showDebug();
I[]9
SmArray.scalar(9).ravel().showDebug();
I[1]9
```

take() and **drop()** let you select leading or trailing pieces of arrays:

```
v = SmArray.sequence(10);
v.show();
0 1 2 3 4 5 6 7 8 9
```

```

    v.takeBy(4).show(); // take the first 4 values
0 1 2 3
    v.takeBy(-4).show(); // take the last 4 values
6 7 8 9

```

drop() gives you what's left after dropping off part of the array:

```

    v.dropBy(4).show();
4 5 6 7 8 9
    v.dropBy(-4).show();
0 1 2 3 4 5

```

Getters and Setters

These are the methods for extracting parts of an array into native variables (int, string, etc) and for storing native values back into an array. For example,

```

// Create a string array
SmArray prez = SmArray.vector( "Washington", "Jefferson", "Adams", "Monroe" );
prez.show();
Washington Jefferson Adams Monroe

// get the string at position 2 as a native string
string s2 = prez.getString(2);
System.Console.WriteLine( s2 );
Adams

// replace the string at position 2
prez.setString( "Adams(1)", 2 );
prez.show();
Washington Jefferson Adams(1) Monroe

// retrieve the entire array as string[] array
string[] prez_strings = prez.getStrings();
System.Console.WriteLine(
    "Length=" + prez_strings.Length + " "
    + prez_strings[0] + "... "
    + prez_strings[prez_strings.Length-1] );
Length=4 Washington...Monroe

// retrieve a part of an array as strings
prez_strings = prez.getStrings(1,2); // get 2 values starting at position 1
System.Console.WriteLine(
    "Length=" + prez_strings.Length + " "
    + prez_strings[0] + "... "
    + prez_strings[prez_strings.Length-1] );
Length=2 Jefferson...Adams(1)

// replace a series of values starting at a specific position
prez.setStrings( new String[]{ "Thos. Jefferson", "John Adams" }, 1 );
prez.show();
Washington Thos. Jefferson John Adams, Monroe

```

The examples above used the methods for retrieving or setting string data into an array. There are variants to get and set int, char, byte, double, and other native types. The full list is found in the SmartArrays Reference. All work the same way.

The “getter” methods always return the result of the type requested, no matter what the internal form of an array. If the array’s internal type is dtDouble, you can still call getInt(), which will return an

integer result provided that the double value is a whole number. If the double value cannot be converted to the requested type, the getter method will produce an SA_CANNOT_CONVERT error.

Similarly, “setter” methods will force the target array to a different type if necessary in order to replace the values being stored into the array.

Gluing Arrays Together – the Catenate Family

One of the most common things to do with two arrays is glue them together. The **catenate()** method and its variants handle this.

```
SmArray v1 = SmArray.vector( 10, 11, 12 );
SmArray v2 = SmArray.vector( 20, 21, 22 );

v1.catenate(v2).show();
10 11 12 20 21 22

// Make a 2-row matrix from two vectors:
SmArray m = v1.rowCat(v2);
m.show();
10 11 12
20 21 22

// Catenate a new row on to the matrix
m = m.rowCat( SmArray.vector(30,31,32) );
m.show();
10 11 12
20 21 22
30 31 32

// Make a 2-column matrix from two vectors:
m = v1.colCat(v2);
m.show();
10 20
11 21
12 22

m = m.colCat( SmArray.vector(30,31,32) );
m.show();
10 20 30
11 21 31
12 22 32

// catenate a leading column
m = SmArray.vector(0,1,2).colCat(m);
m.show();
0 10 20 30
1 11 21 31
2 12 22 32
```

Other methods for gluing arrays together include **catenateAxis()**, **laminare()**, **laminareAxis()**, and **append()**.

Arithmetic, Comparison, and Logical Methods

A few examples of arithmetic operations:

```
SmArray v1 = SmArray.vector(10, 20, 30 );
SmArray v2 = SmArray.vector( 5, 25, 30 );
```

```

// Subtract one array from another
v1.minus(v2).show();
5 -5 0

```

Find the larger of corresponding items:

```

v1.maximum(v2).show();
10 25 30

// Calculate the square root
v1.sqrt().show();
3.16227766017 4.472135955 5.4772255705

```

Comparison methods return dtBoolean (bit) arrays.

```

v1.eq(v2).show();
0 0 1

```

Logical methods work on boolean arrays and produce boolean results.

```

v1 = SmArray.sequence(10);
SmArray inrange = v1.gt(3).and(v1.lt(8));
inrange.show();
0 0 0 0 1 1 1 1 0 0

```

Compress

One of the most powerful data manipulation methods is **compress()**, which uses a boolean vector to select a subset of an array. Combined with the boolean operations, compress provides the means to select data based on logical criteria.

We'll look at **compress()** by first generating a boolean vector that marks the position of interesting values in a vector

```

SmArray v1 = SmArray.sequence(10);
SmArray inrange = v1.gt(3).and(v1.lt(8));
inrange.show();
0 0 0 0 1 1 1 1 0 0

```

Next, we'll use **compress()** to select a subset array of just the interesting values. Compress returns an array with values at the positions where the boolean vector is 1:

```

SmArray selected = inrange.compress(v1);
selected.show();
4 5 6 7

```

Compress lets you do operations on arrays that parallel database queries. Suppose we have three "columns" of data, each represented as a vector:

```

SmArray states = SmArray.vector( "AK", "AL", "AR", "AZ", "CA", "CT", "DE",
                                "FL", "GA", "HI" );

// population in millions
SmArray pop = SmArray.vector( .648, 4.5, 2.75, 5.58, 35.5, 4.55, .817,
                              17, 8.68, 1.28 );

// area in thousands of square miles
SmArray area= SmArray.vector( 663, 52, 53, 114, 164, 5.5, 2.5, 65.7, 59, 11);

```

If we had these in a database table named **states**, we could do an SQL query like this:


```

SELECT state
FROM states
WHERE pop > 5
      AND pop < 20
      AND area < 100

```

Here's how to do the same thing with SmartArrays, using boolean operations and **compress()**:

```

SmArray subset = pop.gt(5).and(pop.lt(20)).and(area.lt(100)).compress(states);
subset.show();
FL GA

```

Compress() can also operate on a matrix. Let's turn the three vectors into a 3-column matrix:

```

SmArray matrix = states.colCat(pop).colCat(area);
matrix.show();
AK .648 663
AL 4.5 52
AR 2.75 53
AZ 5.58 114
CA 35.5 164
CT 4.55 5.5
DE .817 2.5
FL 17 65.7
GA 8.68 59
HI 1.28 11

int POP = 1; // constant to remember which column is which
int AREA = 2;
SmArray bits = matrix.column(POP).gt(5)
               .and( matrix.column(POP).lt(20) )
               .and( matrix.column(AREA).lt(100) );
bits.show();
0 0 0 0 0 0 0 1 1 0

// apply bits to compress the axis 0 (rows)
bits.compressAxis( matrix, 0 ).show();
FL 17 65.7
GA 8.68 59

```

Index Selection

Another essential feature of SmartArrays is the ability to use subscripting to select values by their position in an array. The method `index()` is a general facility for doing this. It corresponds to the native array operation `array[index]` except that the index can be an entire array of values. The result is the **values** selected by the subscripts in the **index** array.

Let's look at some examples:

```

// Make a vector of sample data
SmArray v = SmArray.vector("aa", "bb", "cc", "dd", "ee", "ff", "gg");
v.show();
aa bb cc dd ee ff gg

// Select the values at positions 2, 3, and 6
SmArray inds = SmArray.vector(2,3,6);
v.index(inds).show();
cc dd gg

```

```

// The subscripts can be in any order and can be repeated:
v.index( SmArray.vector( 3,2,3,2 ) ).show();
dd cc dd cc

// Use indexInto() to replace values at specific positions
SmArray upper = v.index(inds).strUpper();
upper.show();
CC DD GG

v.indexInto( upper, inds );
v.show();
aa bb CC DD ee ff GG

// Indexing a matrix
SmArray m = SmArray.sequence(12,1).reshapeBy(3,4);
m.show();
1 2 3 4
5 6 7 8
9 10 11 12

// Select the matrix for rows (1,2) and columns(2,3):
m.index( SmArray.vector(1,2), SmArray.vector(2,3) ).show();
7 8
11 12

// Using row and column indexing on matrices
m.row(0).show();
1 2 3 4
m.column(0).show();
1 5 9

m.row( SmArray.vector(0,0,2,1) ).show();
1 2 3 4
1 2 3 4
9 10 11 12
5 6 7 8

// Replacing rows and columns
SmArray m2 = m.copy();
m2.rowInto( SmArray.vector( 5000, 6000, 7000, 8000 ), 0 );
m2.show();
5000 6000 7000 8000
5 6 7 8
9 10 11 12

// Scatter-point indexing: Use a matrix of subscripts where
// each column is the position on a dimension:
// Example:
inds = SmArray.vector(0,3)
    .rowCat(SmArray.vector(1,1))
    .rowCat(SmArray.vector(2,2));
inds.show();
0 3
1 1
2 2

// Select a vector of just the values at these positions
v = m.selectBySubscript(inds);
v.show();
4 6 11

// Replace the values at these positions:

```

```

    m.selectIntoBySubscript( v.plus(100), inds );
    m.show();
1  2  3  104
5  106 7  8
9  10  111 2

```

The method `selectUpdateBySubscript()` uses the indexing technique described above to aggregate data values in analytical cases like cross tabulation or histograms. It is a very important technique but we will omit it here in the interest of brevity.

The full vocabulary of SmartArrays indexing methods includes other techniques. These are described in the manual but the ones shown here are the most commonly used.

Searching, Relating, Sorting, and Grouping

We will return to the example of states and their populations and land areas

```

SmArray state = SmArray.vector( "AK", "AL", "AR", "AZ", "CA",
                                "CT", "DE", "FL", "GA", "HI" );
// population in millions
SmArray pop = SmArray.vector( .648, 4.5, 2.75, 5.58, 35.5, 4.55, .817,
                               17, 8.68, 1.28 );

// area in thousands of square miles
SmArray area= SmArray.vector( 663, 52, 53, 114, 164, 5.5, 2.5, 65.7, 59, 11 );

```

Suppose we want to focus on eastern states subset of the list of states. Given the list of state abbreviations in the subset, we can use **lookup()** to locate where they occur in **state**:

```

SmArray eastern_states = SmArray.vector("CT", "DE", "FL", "GA" );
// Where do the eastern states appear in the list?
SmArray i = state.lookup( eastern_states);
i.show();
5 6 7 8

```

In other words, "CT" is found at position 5, "DE" at position 6, and so on. We know that the vector **pop** is related to the vector **state**, so we know that the population of CT is found at position 5 in **pop**, and so on. We can use **index()** to select the values of population that correspond to these states.

```

// Using result of lookup to relate items of one vector to another
// Get the land areas for the eastern states
area.index(i).show();
5.5 2.5 65.7 59

```

A related method is `member()`, which produces a bitmap that maps the intersection of two sets:

```

// Which of the states are members of the set of eastern states?
state.member( eastern_states ).show();
0 0 0 0 0 1 1 1 1 0

```

The result of **member()** can be used to compress an equal-length vector:

```

// What is total population of the states in eastern_states?
state.member( eastern_states ).compress( pop ).reduce( Sm.plus ).show();
31.047

```

Sorting in SmartArrays depends on the methods `gradeUp()` and `gradeDown()`, which return the subscripts for putting an array into sorted order (not the sorted array itself). Return subscripts means

we can use the result with `index()` to arrange the subject array in sorted order, but we can also reorder a related array.

```
// gradeUp() for ascending order; gradeDown for descending order

// sort the population in ascending order
pop.index( pop.gradeUp() ).show();
0.648 0.817 1.28 2.75 4.5 4.55 5.58 8.68 17 35.5

// Works with strings, too
state.index( state.gradeDown() );
HI GA FL DE CT CA AZ AR AL AK

// rank the states according to population per square mile
SmArray ppsm = pop.times(1000000).divide( area.times(1000) );
state.colCat(ppsm).row( ppsm.gradeDown() ).show();
CT 827.3
DE 326.8
FL 258.7
CA 216.5
GA 147.1
HI 116.4
AL 86.5
AR 51.9
AZ 48.9
AK 0.97
```

Interval Mapping

`lookupInterval()` is a variant of `lookup()` that works with inexact matches. It is commonly used for grouping values into intervals or “buckets”. Suppose we want to group the states by land area into three sets: “small” states are less than 20,000 square miles, “medium” states are in the range 20,000 to 100,000 square miles, and “large” states are larger than 100,000 square miles.

```
SmArray intervals = SmArray.vector( 0, 20, 100 );
SmArray size_inds = intervals.lookupInterval( area );
2 1 1 2 2 0 0 1 1 0

// Show the states with their size classifications
SmArray size = SmArray.vector( "small", "medium", "large" );
state.colCat( size.index(size_inds) ).show();
AK large
AL medium
AR medium
AZ large
CA large
CT small
DE small
FL medium
GA medium
HI small
```

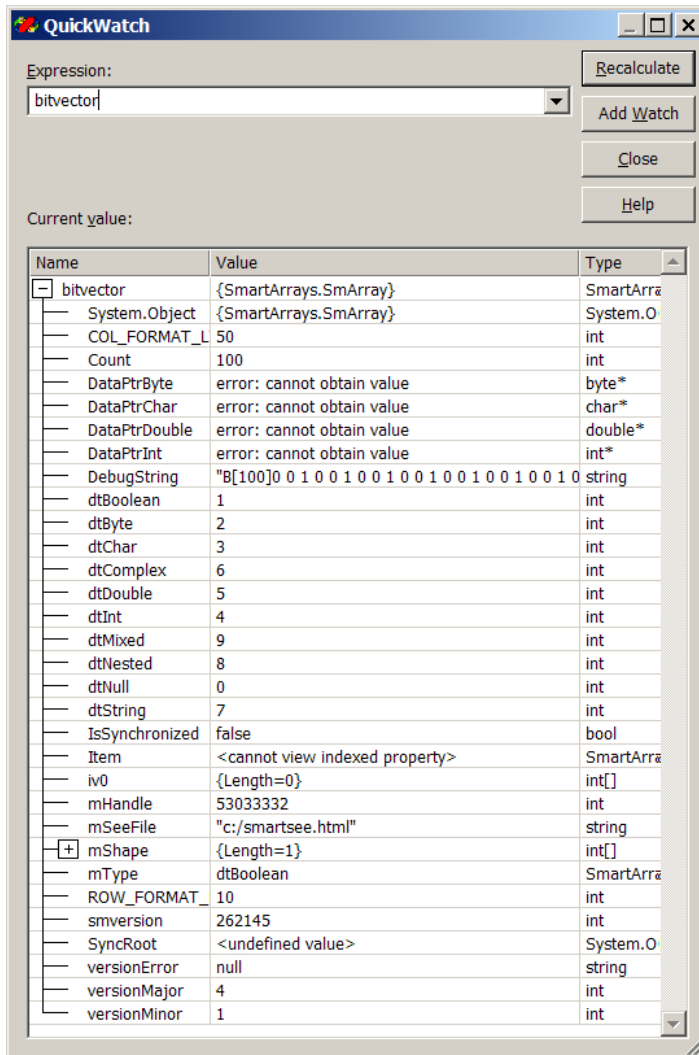
Composite Method Idioms

The uses of the composite methods are not always obvious from their names. The following table lists some of the most common applications of composite methods:

Operation	Method Used	Description
total	reduce(Sm.plus)	Totals a vector or rows of a matrix For bit vector, counts the number of 1s.
column total	reduceAxis(Sm.plus, 0)	Totals the columns of a matrix
all	reduce(Sm.and)	For bit vector, returns 1 if all bits are 1
any	reduce(Sm.or)	For bit vector, returns 1 if any bits are 1
matrix multiply	inner(Sm.plus, Sm.times)	“Dot” product of two vectors; matrix product of two matrices
running total	scan(Sm.plus)	Gives the cumulative total of a series of numbers
running product	scan(Sm.times)	Gives the cumulative product of a series of numbers (for example, calculate compound interest)

Lesson 8: Debugging and Exceptions

Debugging SmartArrays code is a bit different from debugging native C++, C#, or Java programs because the development environment does not understand what an SmArray object really is. If you inspect an SmArray variable named **bitvector** using Visual Studio's QuickWatch window, you'll see something like this:



The DebugString property shows you a summary description of the array, but the other attributes do not give you a great deal of information about this specific array. Note that the DebugString property is available only with SmartArrays for .NET. For Java and C++, keep reading.

The debugging and diagnostic methods provided with SmartArrays are the only way to see what's going on inside an array. We have been using the show() method in this tutorial, which will display the array on the output stream. However this does not tell you the shape or internal datatype of the array.

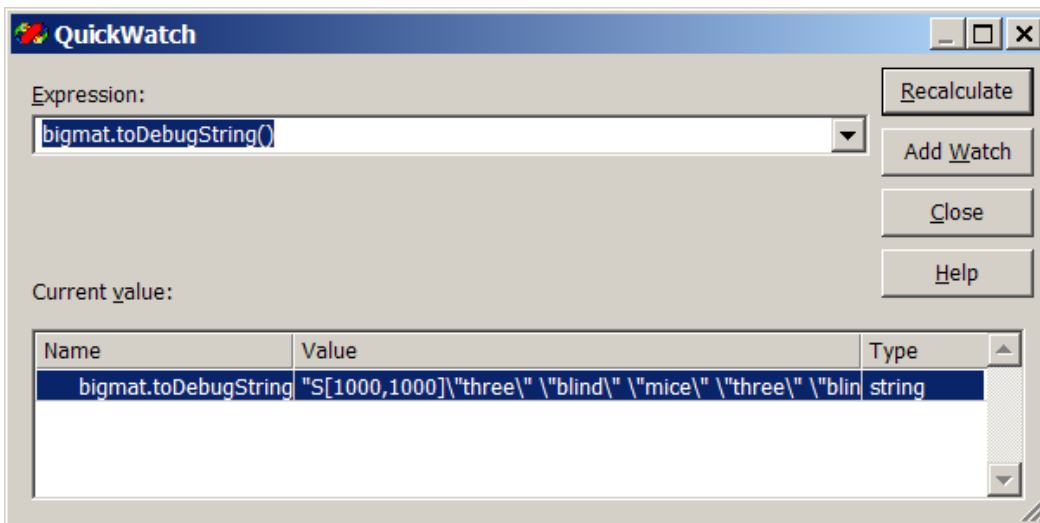
For inspecting an array in code that you are debugging, use **showDebug()**, which produces a one-line synopsis of the array. From it you can tell its internal datatype, shape, and see the first few values. For example:

```
SmArray.sequence(100000).showDebug();
I[100000]0 1 2 3 4 5 6 7 8 9 10 11 12 ....
```

This 'I' indicates that the array is Integer type, the [100000] indicates that it is a vector of length 100000. If it were a matrix or higher rank array, the lengths of all the dimensions would appear inside the brackets. Finally, it shows the first few data values so you can see if the array contains what you expect.

You can use showDebug() inside the expression window of your debugger to display the result on our output stream. Sometimes, particularly when debugging a web server application, you don't have an output stream to write to. In this case, use toDebugString(), which returns a string that contains the debugging information and displays it right in the debugger:

```
// big matrix of a million strings
SmArray bigmat = SmArray.vector("three", "blind", "mice").reshapeBy(1000,1000);
```



If you are logging array operations for diagnostic purposes, toDebugString() will provide a native string that you can easily store in a database, write to file, etc.

Exceptions

All SmartArrays methods employ rigorous checking of arrays when performing an array operation. If the array is unsuited for the operation in any way, an exception of type SmException will be thrown to the caller. Call the exception's **toString()** method to see the error. You can find a table of all SmArray exceptions in the User Manual.

Handling SmartArrays exceptions is like handling any other exception. Here's an example that deliberately tries to add unequal-length vectors:

```
SmArray v1 = SmArray.vector(1,2,3);
SmArray v2 = SmArray.vector(4,5);
SmArray v3;
try
{
    v3 = v1.plus(v2);
}
catch (SmException e )
```

```

{
    System.Console.WriteLine( "SmException: " + e.ToString() );
    System.Console.WriteLine( e.StackTrace );
}

```

This produces an error as expected. The code in the catch block displays the error message and also the call stack that illustrates where in the code the error occurred.

```

SmException:
SmException thrown at Lessons5plus.Debugging.runExamples(), Line 32
Error Code 202: SA_LENGTH - in {I[3]}.plus( {I[2]} )

    at SmartArrays.SmArray.checkRC(Int32 rc)
    at SmartArrays.SmArray.callOpcode(Int32 opcode, Int32[] args )
    at SmartArrays.SmArray.callOpcode(Int32 opcode, Int32 h1, Int32 h2 )
      at SmartArrays.SmArray.plus(SmArray arg2)
    at Lessons5plus.Debugging.runExamplices() in c:\training\lessons c-
sharp\lessons5plus\debuggins.cs:line 32

```

The text of the exception tells you where the error was thrown and provides an explanation of the context. SA_LENGTH is the general error produced when the sizes of arrays do not conform as required. In this case it indicates that the exception occurred in **plus()**, when the subject array is a length-3 integer vector -- **{ I[3] }** – and the argument's length is 2 – **{ I[2] }**.

Lesson 9: Database Data

The SmartArrays data adapter provides the means to build arrays directly from the result of database queries. Included in the SmartArrays wrapper are two classes:

- **SmDatabase** which represents a connection between SmartArrays and a database.
- **SmStatement** which represents a database statement

For detailed control of queries, such as fetching a result in pieces, you will want to use SmStatement, but for simple queries, SmDatabase is enough.

For these examples, we will use the MS Access database “partsales” which is installed as part of the SmartArrays sample programs. SmDatabase is designed to work with any database that can be accessed with ODBC, including enterprise databases like SQL Server and Oracle.

Here is the code provided with the sample C# program that accompanies this tutorial. It begins by establishing a connection to the database. Database operations should always be prepared for exceptions by enclosing them in a try-catch block.

```
// connection string to use with the sample Access database
// supplied with SmartArrays. You will need to have MS Access
// installed in order to use it.
string dbname = "c:\\Program Files\\SmartArrays\\Samples\\data\\partsales.mdb";
string connstr = "DBQ=" + dbname + ";DRIVER={Microsoft Access Driver (*.mdb)}" ;

SmDatabase db = new SmDatabase();
try
{
    db.connect( connstr );
}
catch ( SmException e )
{ // exception will describe ODBC error that caused the connection to fail
    System.Console.WriteLine( e.toString() );
    return;
}
```

At this point the database is connected and we can perform queries on it. The example shows the use of two SmDatabase methods: getColumn(), which describes the columns of a table, and queryVector(), which performs an SQL query and returns it as a SmArray. The data table is small, so we just pull the entire database into a memory-resident array:

```
SmArray result;
SmArray colnames;
try
{
    string table = "sales";
    colnames = db.getColumn( table ).column(4); // names of columns

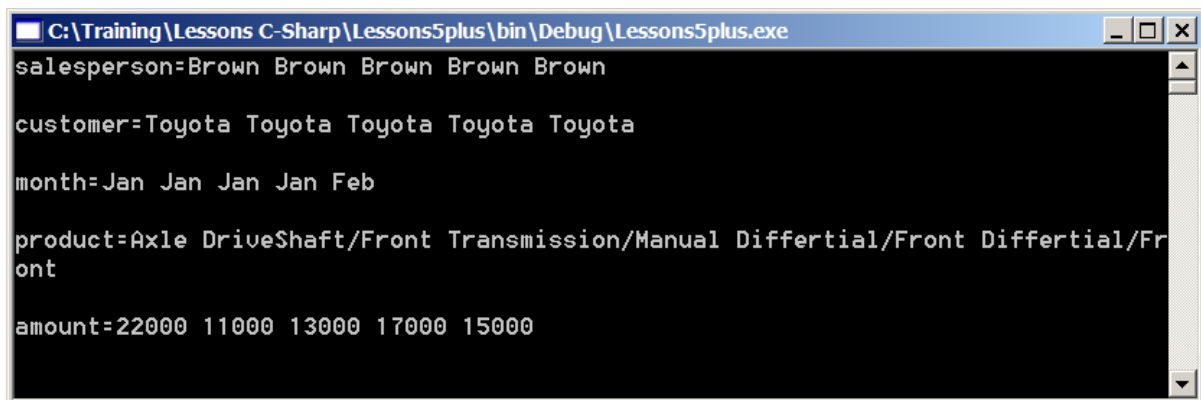
    // Query the database
    result = db.queryVector( "SELECT * FROM " + table );
}
catch ( SmException e )
{
    System.Console.WriteLine( e.toString() );
    return;
}
```

The result returned by **queryVector** is a vector, where each value is itself an SmArray vector. We have not used this kind of “nested” array (type = dtNested) thus far in this tutorial. In this case, all you need to know about a nested array is that you can use the method **pickBy()** to extract a specific item of the result.

Each of the items of the result is a vector, containing one entire column of data returned by the query. Each of the items in the string vector **colnames** is the name of the corresponding column. In the loop below, we will iterate through the items of these two arrays. For each column of the query result we will print out the column name and the first five values in that column.

```
// show the results
for( int i=0; i<colnames.getCount(); i++ )
{
    System.Console.WriteLine(
        colnames.getString(i)
        + "="
        + result.pickBy(i).takeBy(5).toString() );
}
```

The output of the above loop in the console window looks like this.



```
C:\Training\Lessons C-Sharp\Lessons5plus\bin\Debug\Lessons5plus.exe
salesperson=Brown Brown Brown Brown Brown
customer=Toyota Toyota Toyota Toyota Toyota
month=Jan Jan Jan Jan Feb
product=Axle DriveShaft/Front Transmission/Manual Differtial/Front Differtial/Front
amount=22000 11000 13000 17000 15000
```

The SmartArrays Database classes are useful for loading data from other sources besides database, thanks to the versatility offered by ODBC. You can use the classes to read flat files and even Excel spreadsheets. On the SmartArrays web site you can find a White Paper describing how to read Excel data using the ODBC adapter.

Note: The SmartArrays Data Management Toolkit provides a class SmDatabaseConnection that wraps the SmDatabase class with a standardized ISmDbConnection interface. We recommend use of this interface in serious database applications. See *Using the SmartArrays Data Management Toolkit*.

Conclusion

I hope you've found this introductory tutorial useful in learning SmartArrays. If you have any suggestions for improving this tutorial, or if there were parts that were not clear, please let me know. You can email me at james.wheeler@smartarrays.com.

I plan to develop a follow-on training program that will help you master the full power of SmartArrays in building fast and powerful applications. Here is an outline of the topics I plan to include. If there's something you'd like to see included, please let me know.

Next Steps:

The next level tutorial in this series is *Using the SmartArrays Data Management Toolkit*. You will find this in the SmartArrays/samples directory in the form of a PDF file and source code in both Java and C#.