One way to get an array is to input it from the user - e.g.:

```java
int[] stuff = cin.nextIntArray();
```

Another way is to "hard code" it into a program declaration step - e.g.:

```java
int[] stuff = { 3, 15, -6, 201 };
```

This is known as "hard coding" because the initial value of the array named `stuff` is "nailed down hard". Anytime the program runs for any user, `stuff` will always start off with 4 elements with values 3 and 15 and -6 and 201.

HOWEVER, using `nextIntArray`, each time the program runs for a user, that user gets to choose the starting size and element values for `stuff`. 
A third way to get an array is to make an empty array and then add elements to it one at a time. First you do:

```java
int[] stuff = new int[0];
```

The `new int[0]` part creates a new array of integers having `0` many elements. This is like a defective credit card wallet with a nice leather binders, but they left out all the sleeves.

In standard Java, it is not possible to add any elements to the end of any array. However, the easy library uses a trick named "append" to get the same effect. For example

```java
stuff = append(stuff,31);
```

will change the array named `stuff` so that it is now `{31}`. Then we could do:

```java
stuff = append(stuff,7);
```

And `stuff` will now be the array `{31, 7}`.
The "append" trick will seem to be more work than it is worth. For example the single line of code:

```java
int[] mylist = { 17,5,21,200 };
```

Would require all these steps using the "append" trick:

```java
int[] mylist = new int[0];
mylist = append(mylist,17);
mylist = append(mylist,5);
mylist = append(mylist,21);
mylist = append(mylist,200);
```

However, read more pages to see where the "append" trick might be better.
Here is another hard coded array example:

```java
int[] alist = { 3, 10,17,24,31,38,45 };
```

And the append trick would require:

```java
int alist = new int[0];
alist = append(alist,3);
alist = append(alist,10);
alist = append(alist,17);
alist = append(alist,24);
alist = append(alist,31);
alist = append(alist,38);
alist = append(alist,45);
```

Ah - but there is a way to do THIS append trick with ONLY THREE lines of code. Can you guess what it is ? (see next page)
The hard coded:

```c
int[] alist = { 3, 10, 17, 24, 31, 38, 45 };
```

can be redone using append in just THREE LINES:

```c
int alist = int[0];
for(int k=3;k<=45;k=k+7)
    alist = append(alist,k);
```

Note that the `for` loop has only one command in its body and thus the sequence braces `{ }` are NOT needed.
The larger hard coded array:

```cpp
int[] alist = { 3, 10, 17, 24, 31, 38, 45, 52, 59, 66,
               73, 80, 87, 94, 101, 108, 115, 122, 129, 136,
               143, 150, 157, 164, 171, 178, 185, 192 };```

can be redone using append without the programmer having to figure out all the arithmetic:

```cpp
int alist = int[0];
for(int k=3;k<=192;k=k+7)
    alist = append(alist,k);
```
Suppose we want an array with the arithmetic sequence 3, 10, 17, 24, ... in it but we want the user to be able to choose how many elements are in the array.

This CANNOT be hard coded at all. Hard coding an array always requires the array ALWAYS has the same size - as shown in its declaration command.

However, the append trick now comes to the rescue:

```java
out.print("How many elements: ");
int size = cin.nextInt();
int[] alist = new int[0];
int value = 3;
for(int k=1;k<=size;++k) {
    alist = append(alist,value);
    value = value + 7;
}
```

NOTE: There are two loop variables, value is the arithmetic series loop variable, and k is the counting-how-many-elements-to-append loop variable.

NOTE: the for loop braces are needed back again.
Here is the code from the previous page:

```java
out.print("How many elements: ");
int size = cin.nextInt();
int[] alist = new int[0];
int value = 3;
for(int k=1;k<=size;++k) {
    alist = append(alist,value);
    value = value + 7;
}
```

Here is another way to do the same thing:

```java
out.print("How many elements: ");
int size = cin.nextInt();
int[] alist = new int[0];
for(int k=1;k<=size;++k) {
    int value = 3 + 7*(k-1);
    alist = append(alist,value);
}
```

**NOTE:** The variable named `value` is no longer "advanced". Instead it is re-computed for each repetition using the loop variable `k`.

**NOTE:** The tricky stuff with `k-1` is needed to get the first value to work out right. Remember, it was supposed to be 3. But `k` starts off being 1 and so `k-1` starts off being 0 and so `3+7*0` is 3 +0 is 3 is the right value.
Here is the last version from the previous page:

```java
out.print("How many elements: ");
int size = cin.nextInt();
int[] alist = new int[0];
for(int k=1;k<=size;++k) {
    int value = 3 + 7*(k-1);
    alist = append(alist,value);
}
```

Here a trick is used to get rid of the $k-1$

```java
out.print("How many elements: ");
int size = cin.nextInt();
int[] alist = new int[0];
for(int k=1;k<=size;++k) {
    int value = 3 + 7*(k-1);
    alist = append(alist,value);
}
```

SEE if you can figure out how the trick works.
The previous examples used an arithmetic sequence as the source of the element values.

You can get a series of element values lots of OTHER WAYS.

In particular, elements of one array can be used to provide element values for another array.

Suppose an array mystuff of integers is given (maybe the user typed it in - we don't care).
We want a new array named morestuff each of whose elements is larger by 10 than the corresponding elements of mystuff.
Thus if mystuff happened to be \{ 3 , 1 , 8 , 2 \}
We would want morestuff to become \{ 13, 11, 18, 12 \}

The code is on the next page.
Suppose an array `mystuff` of integers is given (maybe the user typed it in - we don't care).
We want a new array named `morestuff` each of whose elements is larger by 10 than the corresponding elements of `mystuff`.
Thus if `mystuff` happened to be \{3, 1, 8, 2\}
We would want `morestuff` to become \{13, 11, 18, 12\}

Here is the code - notice the use of a `for-in` loop to get a series of values named `item` extracted from the array named `mystuff`. Then the loop variable `value` is used to `recompute` an element `value` to add to the newly constructed array `morestuff`.

```java
int[] morestuff = new int[0];
for (int item : mystuff) {
    int value = item + 10;
    morestuff = append(morestuff, value);
}
```
As another example, suppose an array named `somelist` is given and we wish to construct another array `backlist` consisting of the same elements BUT in the reverse order:

For example, if `somelist` were `{ 3, 1, 9, 23, 7 }
We would want `backlist` to become `{ 7, 23, 9, 1, 3 }

Note that loop variable `k` sequences through the possible subscript values in REVERSE order:

```java
int[] backlist = new int[0];
for (k = somelist.length - 1; k >= 0; k = k - 1) {
    int value = somelist[k];
    backlist = append(backlist, value);
}
```

Try to figure out why the `-1` and the `>=0` are used.
Can you rewrite this code to use 3 lines of code instead of 4 (not counting the lonely `}` line)???
The easy library append trick is easy to understand and use but is INCREDABLY INEFFICIENT.

It is MUCH better to use an array initialized to the desired final size and just assign the desired values to its subscripted elements.

Here is simple "append style" code to fill an array with 10 numbers 2,4,6,...

```java
int[] stuff = new int[0];
for(int k=2;k<=10;k=k+2)
    stuff = append(stuff,k);
```

Here is the most efficient want to do this:

```java
int[] stuff = new stuff[10]; // starts off with ten 0's
int k = 2+n*2;
stuff[n] = k;
for(int n=0;n<10;++n) {
    // counting from 0 trick
    // k progresses: 2,4,6,... as before
    // stick value directly into array
}
```

NOTE: This requires that we know how many elements are in the new array BEFORE we begin computing those elements.