Arrays
We have named variables ...

But each variable holds just one value, and we can’t really manage thousands of them.
Clumsy and Worse ...

Suppose we wanted to find the *longest five* and *shortest five* palindromes in the dictionary ... 

*Possible, but messy.*

Suppose we wanted to compute the median (middle value) length of dictionary words ... 

*Not even possible.*

*Possible, but messy and very slow.*
One name, many boxes

```java
int[] a = new int[6];
a[3] = 42;
```
Some details ...

\texttt{int \ [ \ ] \ a} does not create the boxes;
\texttt{new array[6]} does

The variable actually holds a reference (pointer) to the array.
When we create the array with \texttt{new}, we have to know how many elements (boxes) to create.
Array variables are references

int [ ] a = new int [ 4 ];

This is why we can change the items in a method. What we pass to the method is the reference.
Initialized arrays

int [ ] daysInMonth =
    new int [ ] { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

(Size comes from the initialization.)

Why did I make 13 months, and give the first zero days?
Arrays and loops: Better together

```java
int [ ] b = new int [ ] { 13, 11, 42, 26, 29 };
...
int max = b[0];  // after checking b.length
for (int i=0; i < b.length; ++i) {
    if (b[i] > max) {
        max = b[i];
    }
}
```
Make change

Desired behavior:

$ java Change 42.35 100

Making change for $57.65

57 dollars

  2 quarters

  1 dimes

  1 nickels
Making change ...

/**
 * Dispense a combination of coins or bills for a given total.
 * @param owed Total amount of change to make, in pennies.
 */
static void simpleChange(long owed) {
    System.out.format("Making change for $%d.%d\n",
                      owed / 100, owed % 100);

    ...
}

Strategy: Try dispensing highest value coin or bill, then next smaller, etc.,
Making change ...

/**
 * Dispense a combination of coins or bills for a given total.
 * @param owed  Total amount of change to make, in pennies.
 */
static void simpleChange(long owed) {
    System.out.format("Making change for $%d.%d\n",
                       owed / 100, owed % 100);

    if (owed >= 100) {
        System.out.format("%3d %s\n", owed / 100, "dollars");
        owed = owed % 100;
    }
    if (owed >= 25) {
        System.out.format("%3d %s\n", owed / 25, "quarters");
        owed = owed % 25;
    }
    if (owed >= 10) {
        System.out.format("%3d %s\n", owed / 10, "dimes");
        owed = owed % 10;
    }
    if (owed >= 5) {
        System.out.format("%3d %s\n", owed / 5, "nickels");
        owed = owed % 5;
    }
    if (owed >= 1) {
        System.out.format("%3d %s\n", owed, "pennies");
    }
}

Making change ...

```java
/**
 * Dispense a combination of coins or bills for a given total.
 * @param owed Total amount of change to make, in pennies.
 */
static void simpleChange(long owed) {
    System.out.format("Making change for $%d.%d\n",
                       owed / 100, owed % 100);
    if (owed >= 100) {
        System.out.format("%3d %s\n",
                           owed / 100, "dollars");
        owed = owed % 100;
    }
    if (owed >= 25) {
        System.out.format("%3d %s\n",
                           owed / 25, "quarters");
        owed = owed % 25;
    }
    if (owed >= 10) {
        System.out.format("%3d %s\n",
                           owed / 10, "dimes");
        owed = owed % 10;
    }
    if (owed >= 5) {
        System.out.format("%3d %s\n",
                           owed / 5, "nickels");
        owed = owed % 5;
    }
    if (owed >= 1) {
        System.out.format("%3d %s\n", owed, "pennies");
    }
}
```

Repeated pattern:

```java
if (owed >= 25) {
    System.out.format("%3d %s\n", owed / 25, "quarters");
    owed = owed % 25;
}
```
Making change …

/**
 * Dispense a combination of coins or bills for a given total.
 * @param owed Total amount of change to make, in pennies.
 */
static void makeChange(long owed) {
    System.out.format("Making change for $%d.%d\n", 
        owed / 100, owed % 100);
    String[ ] coinNames = { "dollars", "quarters", "dimes", 
        "nickels", "pennies" };
    int[ ] coinValues = { 100, 25, 10, 5, 1 };

    for (int i=0; i < coinValues.length; ++i) {
        int val = coinValues[i];
        String name = coinNames[i];
        if (owed >= val) {
            long giveCoins = owed / val;
            System.out.format("%3d %s\n", giveCoins, name);
            owed = owed % val;
        }
    }
}
Parallel arrays: coinValues, coinNames

coinValues

coinNames
String[ ] coinNames = { "dollars", "quarters", "dimes", "nickels", "pennies" };  
int[ ] coinValues = { 100, 25, 10, 5, 1 };  

for (int i=0; i < coinValues.length; ++i) {  
    int val = coinValues[i];  
    String name = coinNames[i];  
    if (owed >= val) {  
        long giveCoins = owed / val;  
        System.out.format("%3d %s\n", giveCoins, name);  
        owed = owed % val;  
    }  
}
Why is the new version better?

Less code, for a start
   Fewer chances for mistakes
   Less to read
   Less to change
And consider modifying it for Euros, Pesos, or Yuan
   Same program logic, different data values
Methods can change arrays

```java
static void swapItems(String[] stuff, int i, int k) {
    String tmp = stuff[i];
    stuff[i] = stuff[k];
    stuff[k] = tmp;
}

String[] greeAng = new String[] {
    "Yabba", "Dabba", "Doo"
};
swapItems(greeting, 1, 2);
```
Java.util.Arrays

Compare

Arrays.equals(a, b)

Sort

Arrays.sort(a)

Search

Arrays.binarySearch(a, key)

Copy, Fill, ...

Filling part of an array

Arrays don’t grow and shrink ... so we create one bigger than we need, and count how much we have used.
Example: Magic Square

```
  2  7  6
  9  5  1
  4  3  8
```

Horizontal sums: 15 15 15 15

Diagonal sums: 15 15 15 15
How would you check?

static boolean isMagic( int [][] square ) ... 

First: Pseudocode

Then: Java
Get sum of first row ...

```java
int magicSum = 0;
for (int col = 0; col < square.length; ++col) {
    magicSum += square[0][col];
}
```
Other rows the same sum?

```java
for (int row=1; row < square.length; ++row) {
    int rowSum = 0;
    for (int col=0; col < square.length; ++col) {
        rowSum += square[row][col];
    }
    if (rowSum != magicSum) { return false; }
}
```
Columns the same sum?

```java
for (int col=0; col < square.length; ++col) {
    int colSum = 0;
    for (int row=0; row < square.length; ++row) {
        colSum += square[row][col];
    }
    if (colSum != magicSum) { return false; }
}
```
Diagonals the same sum?

```java
int diagSum = 0;
for (int row=0; row < square.length; ++row) {
    int col = row; // left to right, downward
    diagSum += square[row][col];
}
if (diagSum != magicSum) { return false; }
```
Diagonal right to left

diagSum = 0;
for (int row=0; row < square.length; ++row) {
    int col = 2 - row; // diagonal from top right
    diagSum += square[row][col];
}
if (diagSum != magicSum) { return false; }
Some applications of matrices ...

Weather prediction, ecosystem simulation
Each element represents an area
At each time step, state of element changes depending on its neighbors

Medical and research imaging
3 dimensional grid of “voxels” representing a region in the head or body
Indexed Color Images

Note: PNG uses this scheme for some pixels, but not all.
## Masking: Parallel Arrays

### Indexed, masked image

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Color table

![Color table image]

### Transparency (alpha) mask

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Image processing is matrix processing

Image is matrix (grid) of pixels

Example operations:

- Composite (blend) images
- Blur image, or sharpen edges
- Scale image

How would you do these?

(just strategy or very rough pseudocode)
Blur

For each pixel in blurred image

Value is weighted average of surrounding pixels in old image
Unsharp Mask (edge sharpening)

1. Create a blurred image
2. Subtract blurred image from original
   (called “high pass filter”)
3. Add high-pass filtered image to original

Each step creates a new matrix of pixels from one or two other matrices, looping through all the pixels