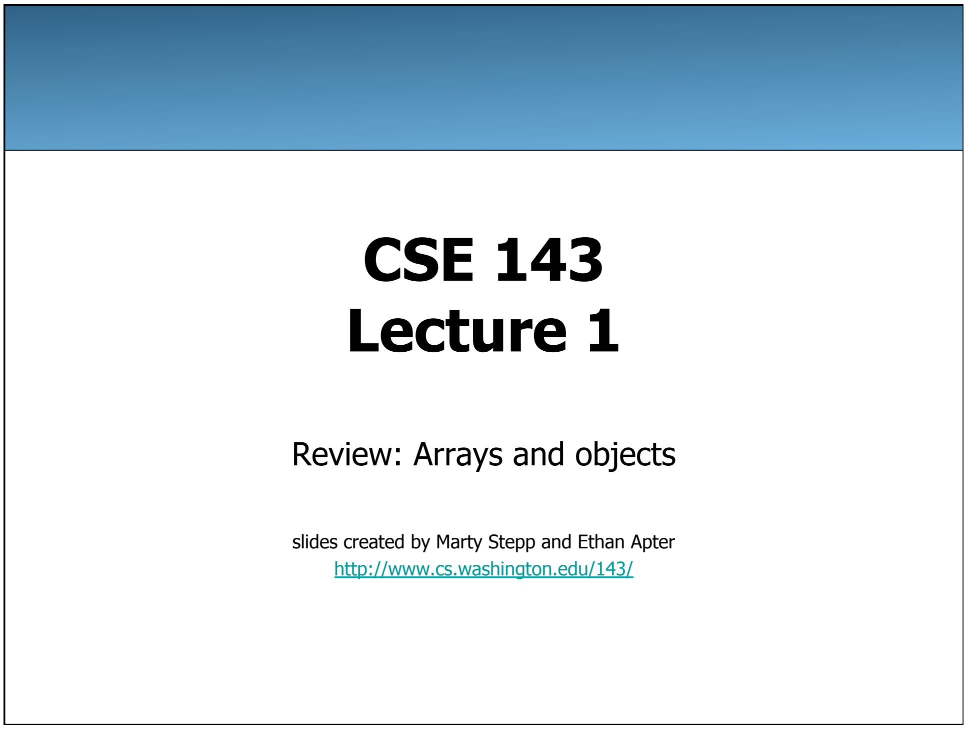
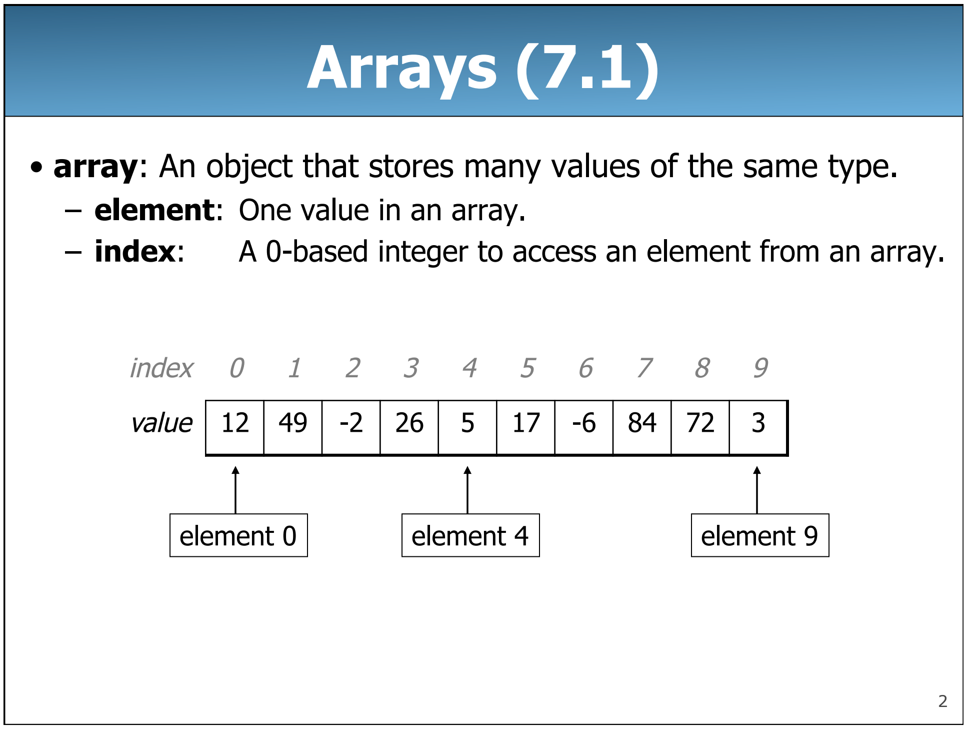


CSE 143
Lecture 1

Review: Arrays and objects

slides created by Marty Stepp and Ethan Apter
http://www.cs.washington.edu/143/



Arrays (7.1)

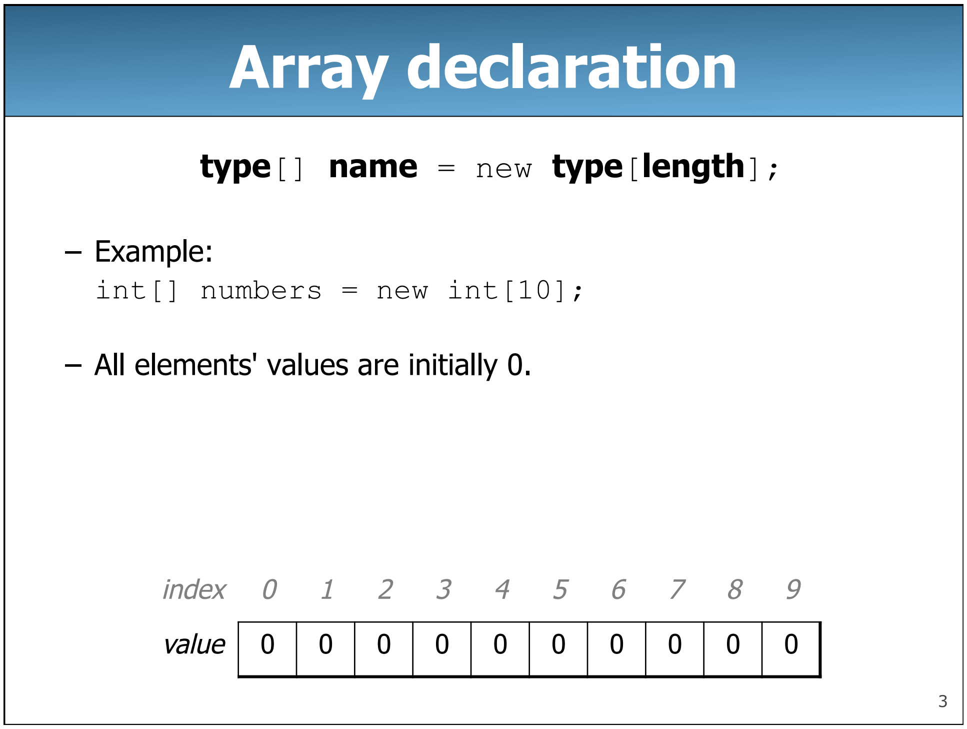
- **array**: An object that stores many values of the same type.
  - **element**: One value in an array.
  - **index**: A 0-based integer to access an element from an array.

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	12	49	-2	26	5	17	-6	84	72	3

element 0

element 4

element 9



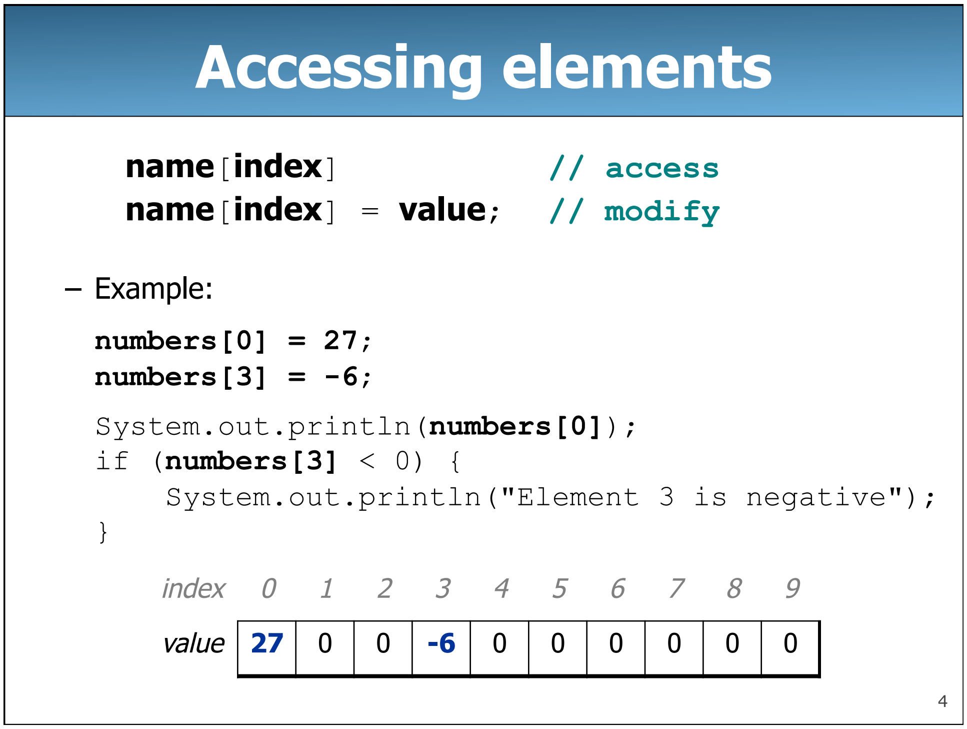
Array declaration

```
type[] name = new type[length];
```

- Example:
  ```
  int[] numbers = new int[10];
  ```

- All elements' values are initially 0.

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	0	0	0	0	0	0	0	0	0



Accessing elements

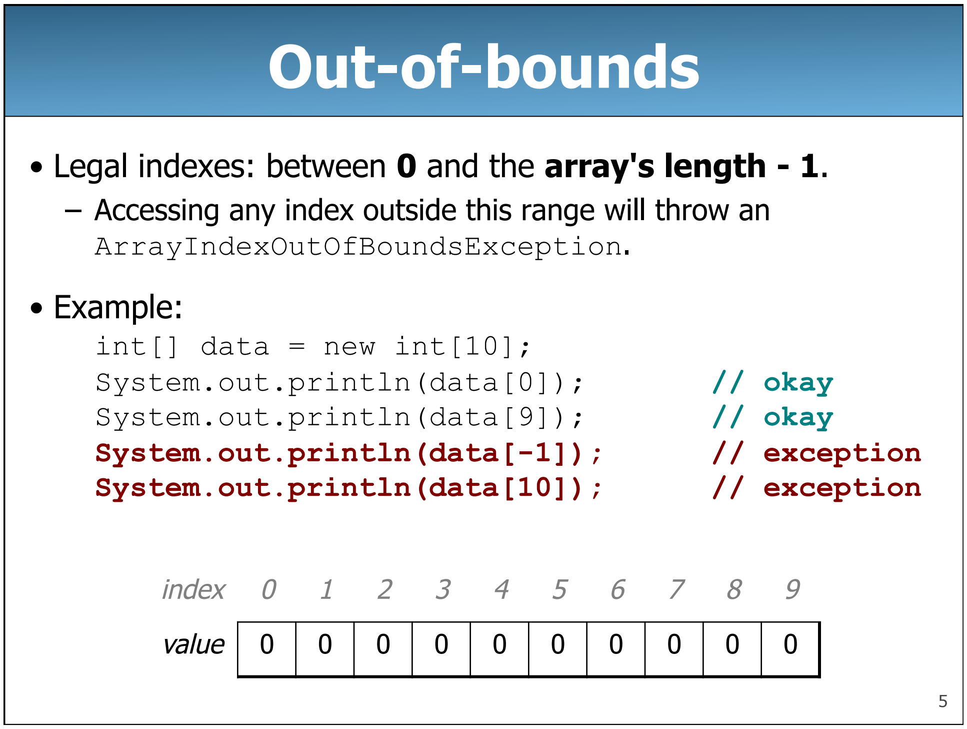
```
name[index]                 // access
name[index] = value;        // modify
```

- Example:

```
numbers[0] = 27;
numbers[3] = -6;

System.out.println(numbers[0]);
if (numbers[3] < 0) {
    System.out.println("Element 3 is negative");
}
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	**27**	0	0	**-6**	0	0	0	0	0	0

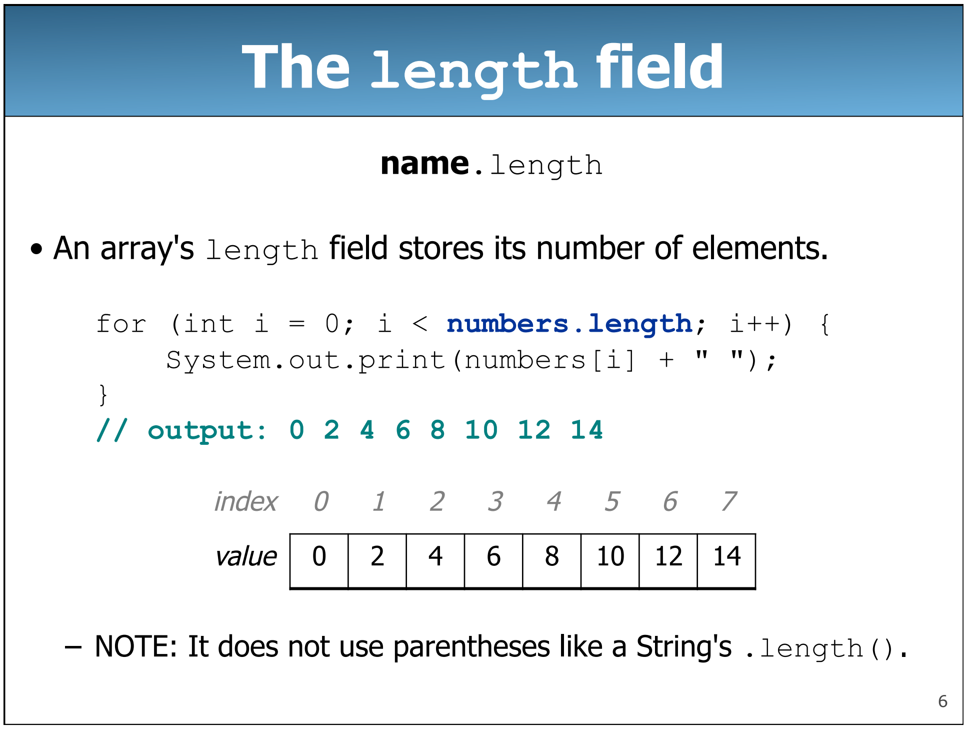


Out-of-bounds

- Legal indexes: between **0** and the **array's length - 1**.
  - Accessing any index outside this range will throw an `ArrayIndexOutOfBoundsException`.
- Example:

```
int[] data = new int[10];
System.out.println(data[0]);        // okay
System.out.println(data[9]);        // okay
System.out.println(data[-1]);       // exception
System.out.println(data[10]);       // exception
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	0	0	0	0	0	0	0	0	0



The `length` field

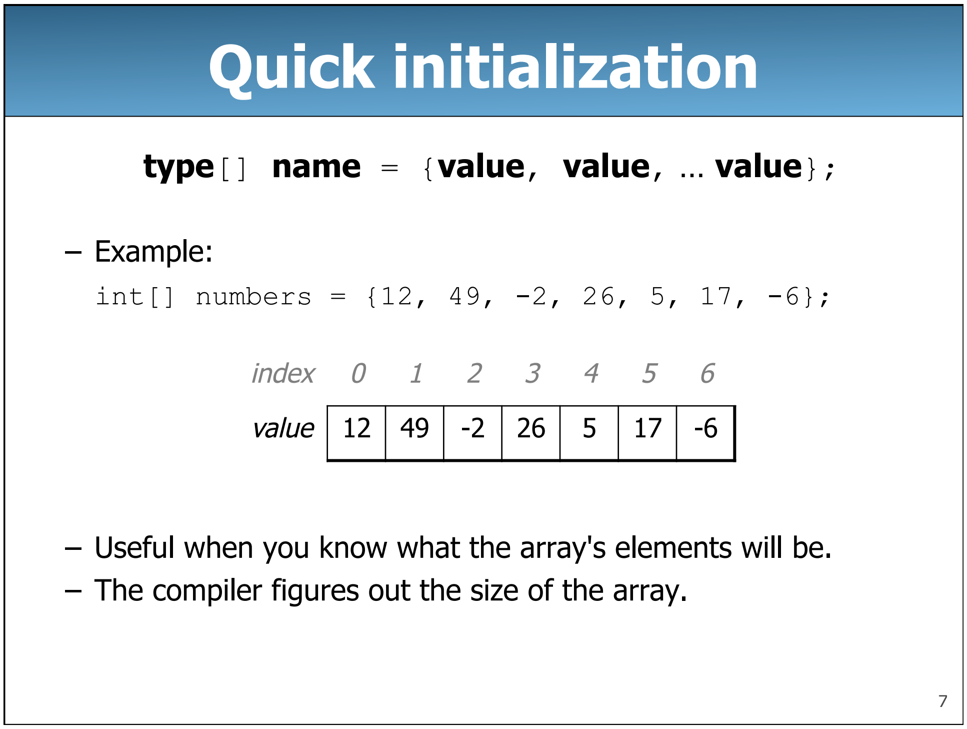
name`.length`

- An array's `length` field stores its number of elements.

```
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
// output: 0 2 4 6 8 10 12 14
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
value	0	2	4	6	8	10	12	14

  - NOTE: It does not use parentheses like a String's `.length()`.



Quick initialization

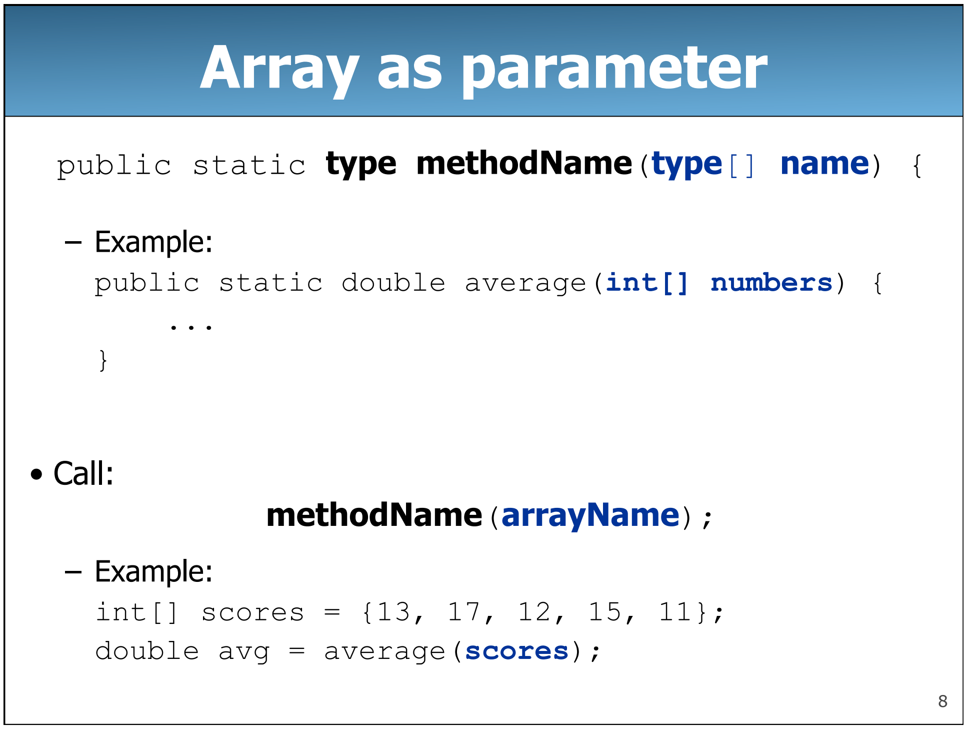
type`[]` **name** `=` `{`**value**`,` **value**`,` … **value**`};`

- Example:

```
int[] numbers = {12, 49, -2, 26, 5, 17, -6};
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
value	12	49	-2	26	5	17	-6

- Useful when you know what the array's elements will be.
- The compiler figures out the size of the array.



Array as parameter

`public static` **type** **methodName**`(`**type**`[]` **name**`) {`

- Example:

```
public static double average(int[] numbers) {
    ...
}
```

- Call:

methodName`(`**arrayName**`);`

- Example:

```
int[] scores = {13, 17, 12, 15, 11};
double avg = average(scores);
```

Array as return

```
public static type[] methodName(parameters) {
```

- Example:

```
public static int[] countDigits(int n) {
    int[] counts = new int[10];
    ...
    return counts;
}
```

- Call:

```
type[] name = methodName(parameters);
```

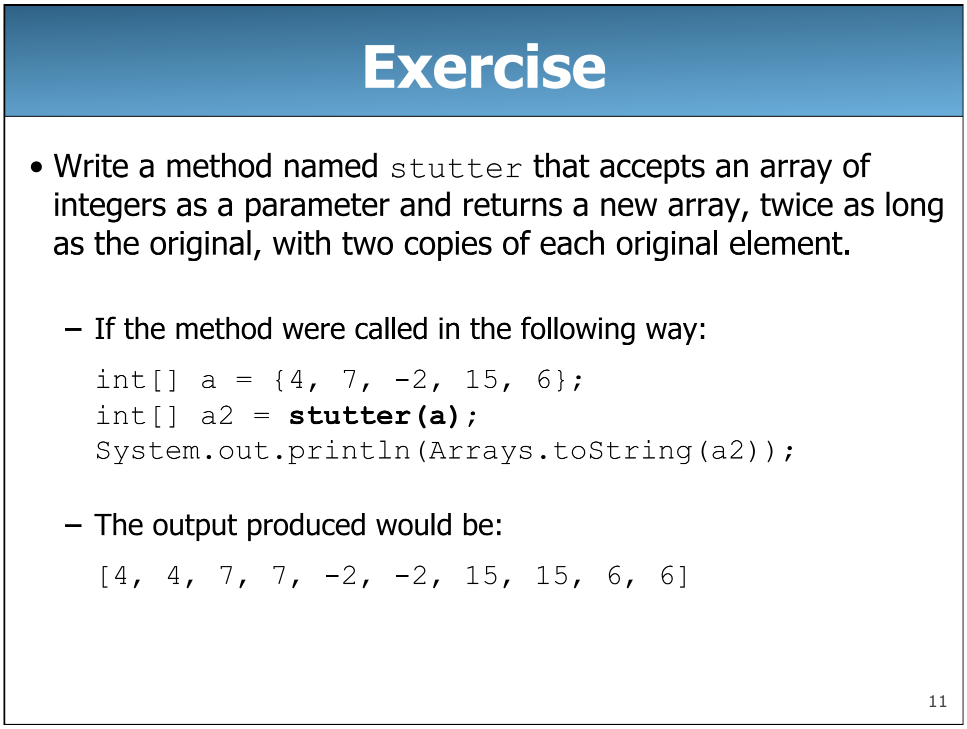
- Example:

```
int[] tally = countDigits(229231007);
System.out.println(Arrays.toString(tally));
```

The `Arrays` class

- Class `Arrays` in package `java.util` has useful static methods for manipulating arrays:

Method name	Description
`equals(`**array1**`, `**array2**`)`	returns `true` if the two arrays contain the same elements in the same order
`fill(`**array**`, `**value**`)`	sets every element in the array to have the given value
`sort(`**array**`)`	arranges the elements in the array into ascending order
`toString(`**array**`)`	returns a string representing the array, such as "`[10, 30, 17]`"



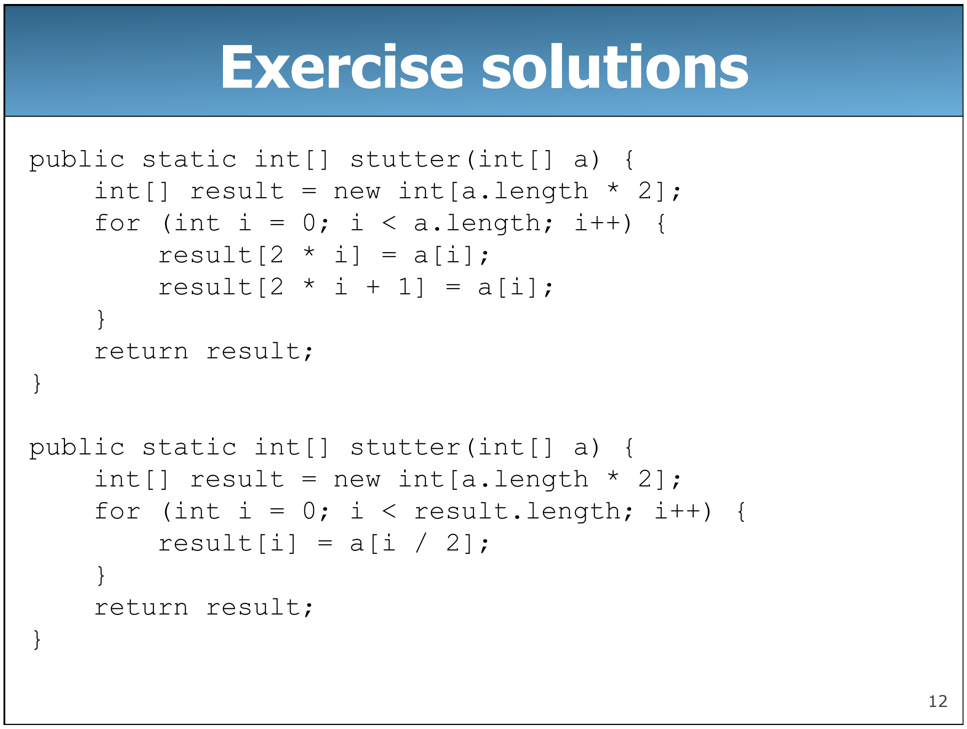
Exercise

- Write a method named `stutter` that accepts an array of integers as a parameter and returns a new array, twice as long as the original, with two copies of each original element.
  - If the method were called in the following way:

```
int[] a = {4, 7, -2, 15, 6};
int[] a2 = stutter(a);
System.out.println(Arrays.toString(a2));
```

  - The output produced would be:

```
[4, 4, 7, 7, -2, -2, 15, 15, 6, 6]
```



Exercise solutions

```
public static int[] stutter(int[] a) {
    int[] result = new int[a.length * 2];
    for (int i = 0; i < a.length; i++) {
        result[2 * i] = a[i];
        result[2 * i + 1] = a[i];
    }
    return result;
}

public static int[] stutter(int[] a) {
    int[] result = new int[a.length * 2];
    for (int i = 0; i < result.length; i++) {
        result[i] = a[i / 2];
    }
    return result;
}
```

Testing code

- Q: How can we tell if our `stutter` method works properly?
  - A: We must test it.

- Q: How do we test code?
  - A: Call the method several times and print/examine the results.

- Q: Can we test all possible usages of this method?
  Q: Can we prove that the `stutter` code has no bugs?
  - A: No; exhaustive testing is impractical/impossible for most code.
  - A: No; testing finds bugs but cannot prove the absence of bugs.

How to test code

- **test case**: Running a piece of code once on a given input.

- Q: Which cases should we choose to test?
  - *equivalence classes of input* : Think about kinds of inputs:
    - positive vs. negative numbers vs. 0; `null` (maybe)
    - unique values vs. duplicates (consecutive and non-consecutive)
    - an empty array; a 1-element array; a many-element array

- Q: What are some properties to look for in testing code?
  - *boundaries* : Hits cases close to a relevant boundary, e.g. the maximum allowed value, the first/last element in an array, etc.
  - *code coverage* : Hits all paths through code (`if`/`else`s, etc.)
  - *preconditions* : What does the method assume? Does the code ever violate those assumptions?

Exercise

- Write a short piece of code that tests the `stutter` method.
  - Decide on a group of test input cases.

  - For each test case:
    - Print the array's contents before and after stuttering.
    - Print whether the test was successful or failed.

Exercise solution 1

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 4, 5, 6};
    int[] a2 = stutter(a1);
    System.out.println(Arrays.toString(a2));
    ...
}
```

- Pros:
  - simple, short

- Cons:
  - must manually check output to see if it is correct
  - must copy/paste to create each test case (redundant)

Exercise solution 2

```
public static void main(String[] args) {
    test(new int[] {1, 2, 4, 5, 6, 8},
         new int[] {1, 1, 2, 2, 4, 4, 5, 5, 6, 6, 8, 8});
    test(new int[] {0, 0, 7, 9},
         new int[] {0, 0, 0, 0, 7, 7, 9, 9});
    test(new int[] {-50, 95, -9876},
         new int[] {-50, -50, 95, 95, -9876, -9876});
    test(new int[] {42}, new int[] {42, 42});
    test(new int[] {}, new int[] {});
}

public static void test(int[] a, int[] expected) {
    int[] a2 = stutter(a);
    System.out.print(Arrays.toString(a) + " -> " +
                     Arrays.toString(a2) + " : ");
    if (Arrays.equals(a2, expected)) {
        System.out.println("Pass");
    } else {
        System.out.println("FAIL!!!");
    }
}
```

Object: iPod

(ok, so it's an old iPod)

<http://www.takeitapart.net/photos/apple_1gen_ipod/>

Object: iPod

- Why did MP3 players, like the iPod, replace CD players?

  – MP3 players are more resistant to skipping

  – MP3 players are smaller

Both true for years before MP3 players became mainstream

  – MP3 players can store more music (a person's entire library!)
    - No need to swap CD's!

The important reason!

Object Motto

An object encapsulates state and behavior

An object encapsulates state and behavior

- **State**: what an object knows
  - Data, usually in the form of variables
  - MP3 player's state
    - Am I on or off?
    - Am I playing music?
    - What song am I playing?
    - How loud is my volume?
    - How many times have I played *I'm On A Boat*?
    - etc

An object encapsulates state and behavior

- **Behavior**: what an object does
  - Actions, usually in the form of methods
  - MP3 player's behavior
    - Turn on/off
    - Play music
    - Pause music
    - Increase volume
    - Increase bass
    - etc

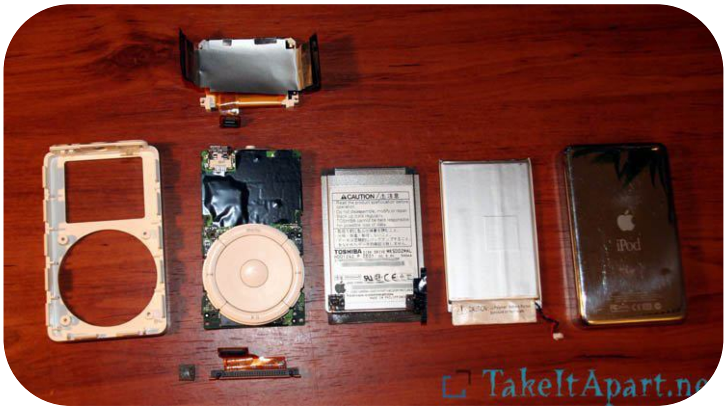
An object encapsulates state and behavior

How many of you know how to use this?

How many of you know how to *build* this?

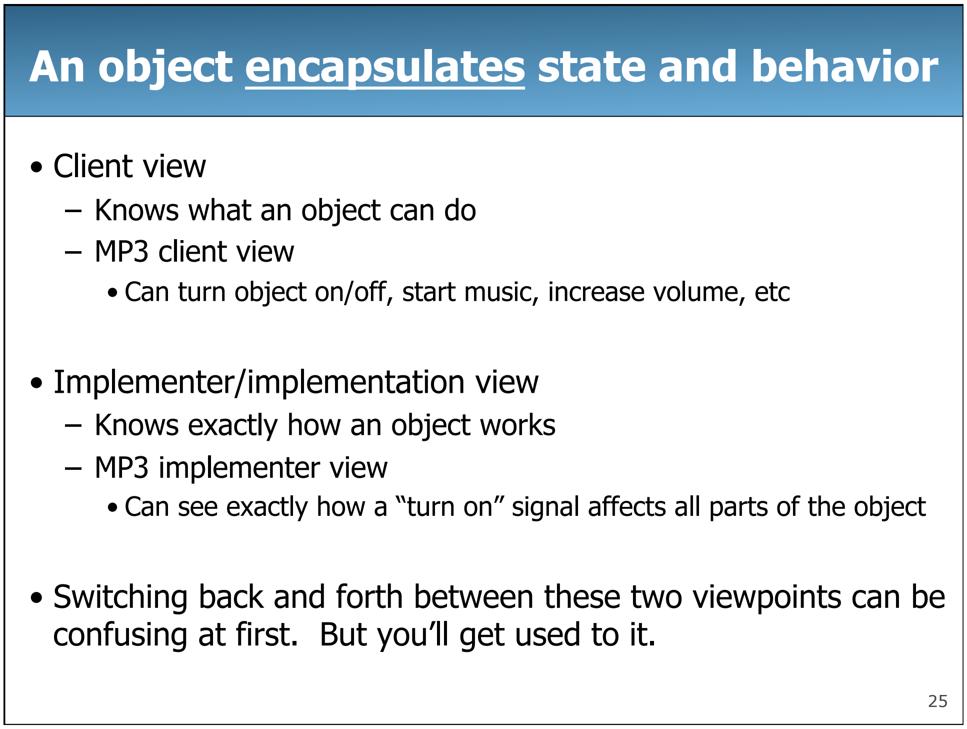
<http://www.takeitapart.net/photos/apple_1gen_ipod/>

An object encapsulates state and behavior



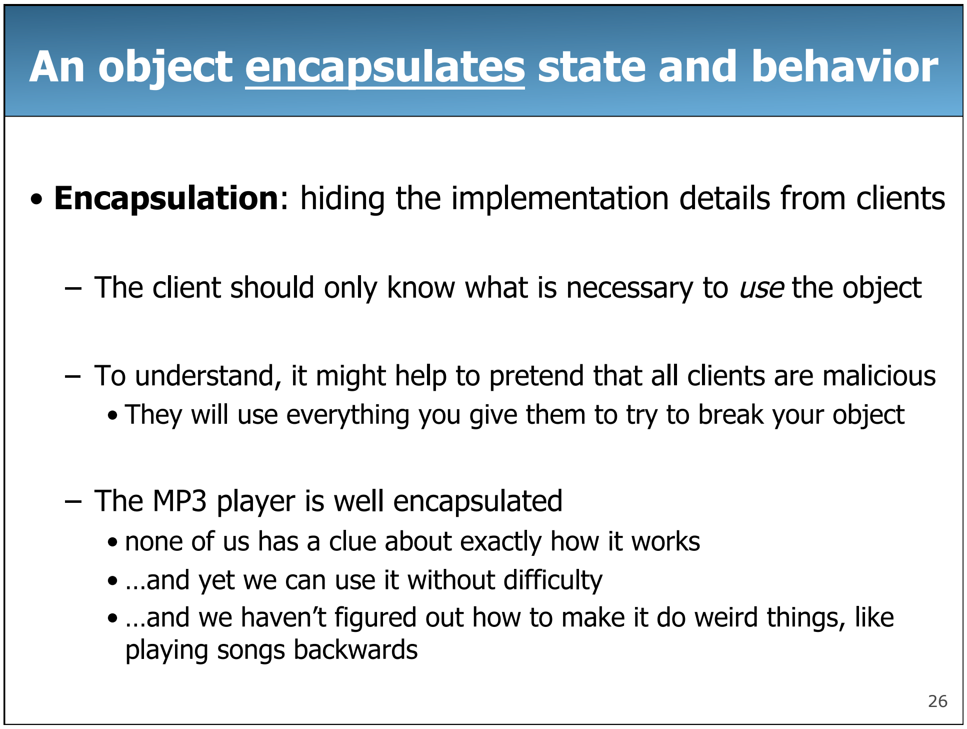
Poor iPod.

<http://www.takeitapart.net/photos/apple_1gen_ipod/>



An object encapsulates state and behavior

- Client view
  - Knows what an object can do
  - MP3 client view
    - Can turn object on/off, start music, increase volume, etc
- Implementer/implementation view
  - Knows exactly how an object works
  - MP3 implementer view
    - Can see exactly how a "turn on" signal affects all parts of the object
- Switching back and forth between these two viewpoints can be confusing at first. But you'll get used to it.



An object encapsulates state and behavior

- **Encapsulation**: hiding the implementation details from clients
  - The client should only know what is necessary to *use* the object
  - To understand, it might help to pretend that all clients are malicious
    - They will use everything you give them to try to break your object
  - The MP3 player is well encapsulated
    - none of us has a clue about exactly how it works
    - …and yet we can use it without difficulty
    - …and we haven't figured out how to make it do weird things, like playing songs backwards