Software Development (cs2500)

Lectures 13 & 15: Arrays and Array Lists

M.R.C. van Dongen

October 21, 2013

Introduction

- Arrays are a special data type in `Java`.
- Arrays are objects that contain other things.
- There are two kinds of arrays:
  1. Arrays containing primitive data type values;
  2. Arrays containing object reference values;
- The type of the array determines what values are in it.
- Before you can use an array you must create it (it's an object).
  - When doing this, you must specify the array's length.
  - The length remains fixed.
- You can put things into the array.
- You can retrieve things from the array.
- You can only access arrays with index values:
  - Only `int` index values are allowed.
  - They must be non-negative;
  - They must be smaller than the length of the array.

Initialisation

Java

```
final int[] numbers = new int[ 10 ];
System.out.println( "length of numbers: " + numbers.length );

final String[] words = new String[ 5 ];
System.out.println( "length of words: " + words.length );
```

Initialisation

Java

```
final int[] numbers = new int[ 10 ];
System.out.println( "length of numbers: " + numbers.length );

final String[] words = new String[ 5 ];
System.out.println( "length of words: " + words.length );
```

Getting Stuff from the Array

- An array is best viewed as a tray/sequence with cups.
- Each cup has a number: 0, 1, …
- The cups contain what's in the array.
- The number of cups is the length of the array.
- Let `array` be a `Java` array.
- Then `array[ i ]` returns what's in the `i`th cup of `array`.

Getting Stuff from the Array

- □ An array is best viewed as a tray/sequence with cups.
- □ Each cup has a number: 0, 1, ...
- □ The cups contain what's in the array.
- □ The number of cups is the length of the array.
- □ Let `array` be a `Java` array.
- □ Then `array[ i ]` returns what's in the `i`th cup of `array`.

Java

```
final int[] numbers = new int[ 10 ];
...
System.out.println( "The first value is " + numbers[ 0 ] );
System.out.println( "The last value is " + numbers[ 9 ] );
```

Getting Stuff from the Array

- An array is best viewed as a tray/sequence with cups.
- Each cup has a number: 0, 1, ...
- The cups contain what's in the array.
- The number of cups is the length of the array.
- Let `array` be a `Java` array.
- Then `array[ i ]` returns what's in the ith cup of `array`.

Java

```
final int[] numbers = new int[ 10 ];
...
System.out.println( "The first value is " + numbers[ 0 ] );
System.out.println( "The last value is " + numbers[ 9 ] );
```

Writing Stuff to the Array

- The notation `array[ index ]` works just as with getting.
- Cups in the arrays work just like variables, so
  - `array[ index ] = value` assigns a `value` to the “`indexth`” cup.

Writing Stuff to the Array

- The notation `array[ index ]` works just as with getting.
- Cups in the arrays work just like variables, so
  - `array[ index ] = value` assigns a `value` to the “`indexth`” cup.

Java

```
final int[] numbers = new int[ 10 ];

numbers[ 0 ] = 1;
numbers[ 9 ] = 42;
System.out.println( numbers[ 0 ] + " == 1" );
System.out.println( numbers[ 9 ] + " == 42" );
```

Writing Stuff to the Array

- The notation `array[ index ]` works just as with getting.
- Cups in the arrays work just like variables, so
  - `array[ index ] = value` assigns a `value` to the “`index`th” cup.

Java

```
final int[] numbers = new int[ 10 ];

numbers[ 0 ] = 1;
numbers[ 9 ] = 42;
System.out.println( numbers[ 0 ] + " == 1" );
System.out.println( numbers[ 9 ] + " == 42" );
```

Writing Stuff to the Array

- The notation `array[ index ]` works just as with getting.
- Cups in the arrays work just like variables, so
  - `array[ index ] = value` assigns a `value` to the “`indexth`” cup.

Java

```java
final int[] numbers = new int[ 10 ];

numbers[ 0 ] = 1;
numbers[ 9 ] = 42;
System.out.println( numbers[ 0 ] + " == 1" );
System.out.println( numbers[ 9 ] + " == 42" );
```

Writing Stuff to the Array

- □ The notation `array[ index ]` works just as with getting.
- □ Cups in the arrays work just like variables, so
  - □ `array[ index ] = value` assigns a `value` to the “`index`th” cup.

Java

```
final int[] numbers = new int[ 10 ];

numbers[ 0 ] = 1;
numbers[ 9 ] = 42;
System.out.println( numbers[ 0 ] + " == 1" );
System.out.println( numbers[ 9 ] + " == 42" );
```

Default Values

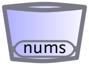
- When you create an array, the array’s content is initialised.
- Each cup in the array is filled with the same value.
- The value depends on the type of the array.

Numeric	`0;`
`boolean`	`false;`
`char`	`'\u0000';`
Object	`null.`

Arrays with Primitive Type Values

Java

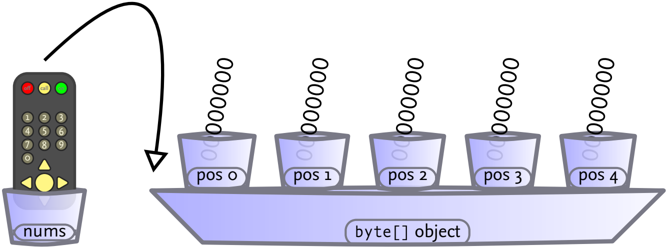
```
byte[] nums = new byte[ 5 ];
nums[ 1 ] = 4;
nums[ 4 ] = 17;
```



Arrays with Primitive Type Values

Java

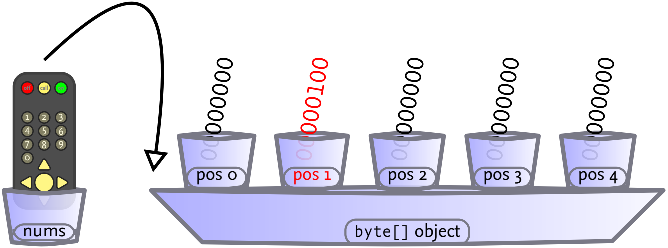
```
byte[] nums = new byte[ 5 ];
nums[ 1 ] = 4;
nums[ 4 ] = 17;
```



Arrays with Primitive Type Values

Java

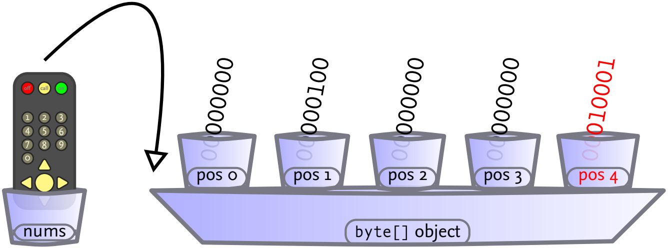
```
byte[] nums = new byte[ 5 ];
nums[ 1 ] = 4;
nums[ 4 ] = 17;
```



Arrays with Primitive Type Values

Java

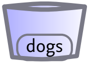
```
byte[] nums = new byte[ 5 ];
nums[ 1 ] = 4;
nums[ 4 ] = 17;
```



Arrays with Objects

Java

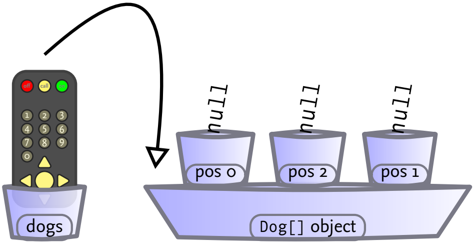
```
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( );
```



Arrays with Objects

Java

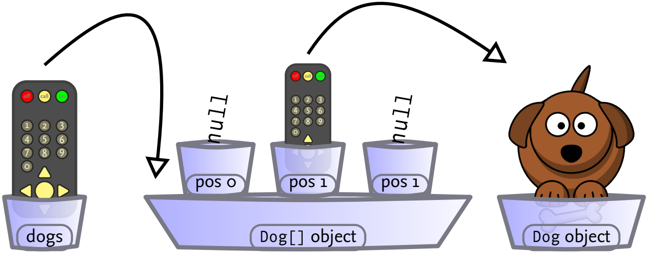
```java
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( );
```



Arrays with Objects

Java

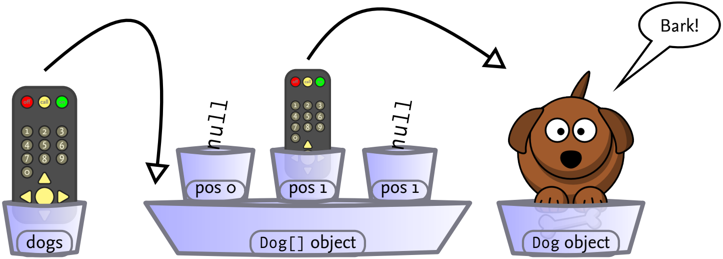
```
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( );
```



Arrays with Objects

Java

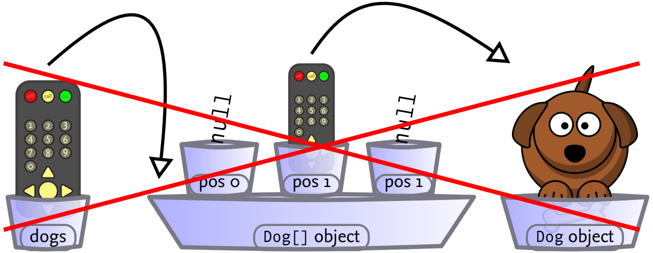
```java
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( );
```



Arrays with Objects

Java

```
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( ); // Run-time error!
```



Arrays do Not Grow

- □ The `length` attribute of a `Java` array is `final`.
- □ So you cannot assign values to ⟨`array`⟩`.length`.
- □ The minimum size of any array is `0`.
- □ The maximum size of any array is `Integer.MAX_VALUE`.

Partially Filled Arrays

- You must fill the array before you can use it.
- You usually start filling at the bottom (index 0).
- Then fill the next position (index 1).
- And so on.
- You need a counter to keep track of the current index.

Java

```
final Scanner scanner = new Scanner( System.in );
final int[] values = new int[ scanner.nextInt( ) ];

int size = 0;
int next = 0;
while ((size != values.length) && (next >= 0)) {
    System.err.println( "Next value (negative value to stop): " );
    next = scanner.next( );
    if (next >= 0) {
        values[ size++ ] = next;
    }
}

final double percentage = 100.0 * size / values.length );
System.out.println( "Percentage filled is " + percentage );
```

Partially Filled Arrays

- □ You must fill the array before you can use it.
- □ You usually start filling at the bottom (index 0).
- □ Then fill the next position (index 1).
- □ And so on.
- □ You need a counter to keep track of the current index.

Java

```java
final Scanner scanner = new Scanner( System.in );
final int[] values = new int[ scanner.nextInt( ) ];

int size = 0;
int next = 0; // We need this to enter the loop.
while ((size != values.length) && (next >= 0)) {
    System.err.println( "Next value (negative value to stop): " );
    next = scanner.next( );
    if (next >= 0) {
        values[ size++ ] = next;
    }
}

final double percentage = 100.0 * size / values.length );
System.out.println( "Percentage filled is " + percentage );
```

Common Errors

Index too Large

Don’t Try This at Home

```
int[] values = new int[ 10 ];

values[ 10 ] = 1;
```

Common Errors

Index too Small

Don’t Try This at Home

```
int[] values = new int[ 10 ];

values[ -1 ] = 1;
```

Common Errors

Uninitialised Values

Don’t Try This at Home

```
String[] words = new String[ 10 ];

if (words[ 0 ].equals( "yes" )) {
    System.out.println( "This isn’t printed." );
} else {
    System.out.println( "This also isn’t printed." );
}
```

Representing Bank Accounts

- Consider a bank account application.
- Each account has an owner and a balance.
  - We could represent the owners using a `String` array;
  - We could represent the balance using a `double` array.

Parallel Array Implementation

Java

```
public class AccountManager {
    private final String[] owners;
    private final double[] balances;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        owners = new String[ size ];
        balances = new double[ size ];
        for (int index = 0; index != size; index++) {
            owners[ index ] = scanner.next( );
            balances[ index ] = scanner.nextDouble( );
        }
    }

    ...
}
```

Parallel Array Implementation

Java

```
public class AccountManager {
    private final String[] owners;
    private final double[] balances;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        owners = new String[ size ];
        balances = new double[ size ];
        for (int index = 0; index != size; index++) {
            owners[ index ] = scanner.next( );
            balances[ index ] = scanner.nextDouble( );
        }
    }

    ...
}
```

Parallel Array Implementation

Java

```
public class AccountManager {
    private final String[] owners;
    private final double[] balances;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        this.owners = new String[ size ];
        this.balances = new double[ size ];
        for (int index = 0; index != size; index++) {
            owners[ index ] = scanner.next( );
            balances[ index ] = scanner.nextDouble( );
        }
    }

    ...
}
```

Parallel Array Implementation

Java

```
public class AccountManager {
    private final String[] owners;
    private final double[] balances;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        owners = new String[ size ];
        balances = new double[ size ];
        for (int index = 0; index != size; index++) {
            owners[ index ] = scanner.next( );
            balances[ index ] = scanner.nextDouble( );
        }
    }

    ...
}
```

Class-Based Implementation

Java

```
public class AccountManager {
  private final Account[] accounts;

  public AccountManager( final int size ) {
    final Scanner scanner = new Scanner( System.in );
    accounts = new Acount[ size ];
    for (int index = 0; index != size; index++) {
      final String owner = scanner.next( );
      final double balance = scanner.nextDouble( );
      accounts[ index ] = new Account( owner, balance );
    }
  }

  ...
}
```

Class-Based Implementation

Java

```
public class AccountManager {
    private final Account[] accounts;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        accounts = new Acount[ size ];
        for (int index = 0; index != size; index++) {
            final String owner = scanner.next( );
            final double balance = scanner.nextDouble( );
            accounts[ index ] = new Account( owner, balance );
        }
    }

    ...
}
```

Class-Based Implementation

Java

```
public class AccountManager {
    private final Account[] accounts;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        this.accounts = new Acount[ size ];
        for (int index = 0; index != size; index++) {
            final String owner = scanner.next( );
            final double balance = scanner.nextDouble( );
            accounts[ index ] = new Account( owner, balance );
        }
    }

    ...
}
```

Class-Based Implementation

Java

```
public class AccountManager {
    private final Account[] accounts;

    public AccountManager( final int size ) {
        final Scanner scanner = new Scanner( System.in );
        accounts = new Acount[ size ];
        for (int index = 0; index != size; index++) {
            final String owner = scanner.next( );
            final double balance = scanner.nextDouble( );
            accounts[ index ] = new Account( owner, balance );
        }
    }

    ...
}
```

Comparison

Stability The parallel array implementation has an “unstable” API:

- If addresses are also needed, we must pass pass one more array.

Security The parallel array implementation is not safe:

- Parallel array clients need access to all arrays:
  - `withdraw( owners, balances, nr, amount );`
  - This gives the client access to all account details.
  - They can even modify the array.
  - It violates encapsulation.
- `Account` clients only see relevant details.
  - `withdraw( accounts[ nr ], amount );`
  - (We must trust the implementation of `withdraw( )`.)
- Better if `withdraw` is (trusted) instance method.
  - `accounts[ nr ].withdraw( amount );`

Comparison

Stability The parallel array implementation has an “unstable” API:

- If addresses are also needed, we must pass pass one more array.

Security The parallel array implementation is not safe:

- Parallel array clients need access to all arrays:
  - `withdraw( owners, balances, nr, amount );`
  - This gives the client access to all account details.
  - They can even modify the array.
  - It violates encapsulation.
- `Account` clients only see relevant details.
  - `withdraw( accounts[ nr ], amount );`
  - (We must trust the implementation of `withdraw( )`.)
- Better if `withdraw` is (trusted) instance method.
  - `accounts[ nr ].withdraw( amount );`

Comparison

Stability The parallel array implementation has an “unstable” API:

- If addresses are also needed, we must pass pass one more array.

Security The parallel array implementation is not safe:

- Parallel array clients need access to all arrays:
  - `withdraw( owners, balances, nr, amount );`
  - This gives the client access to all account details.
  - They can even modify the array.
  - It violates encapsulation.
- `Account` clients only see relevant details.
  - `withdraw( accounts[ nr ], amount );`
  - (We must trust the implementation of `withdraw( )`.)
- Better if `withdraw` is (trusted) instance method.
  - `accounts[ nr ].withdraw( amount );`

Comparison

Stability The parallel array implementation has an “unstable” API:

- If addresses are also needed, we must pass pass one more array.

Security The parallel array implementation is not safe:

- Parallel array clients need access to all arrays:
    - `withdraw( owners, balances, nr, amount );`
    - This gives the client access to all account details.
    - They can even modify the array.
    - It violates encapsulation.
- `Account` clients only see relevant details.
    - `withdraw( accounts[ nr ], amount );`
    - (We must trust the implementation of `withdraw( )`.)
- Better if `withdraw` is (trusted) instance method.
    - `accounts[ nr ].withdraw( amount );`

Comparison

Stability The parallel array implementation has an “unstable” API:

- If addresses are also needed, we must pass pass one more array.

Security The parallel array implementation is not safe:

- Parallel array clients need access to all arrays:
  - `withdraw( owners, balances, nr, amount );`
  - This gives the client access to all account details.
  - They can even modify the array.
  - It violates encapsulation.
- `Account` clients only see relevant details.
  - `withdraw( accounts[ nr ], amount );`
  - (We must trust the implementation of `withdraw( )`.)
- Better if `withdraw` is (trusted) instance method.
  - `accounts[ nr ].withdraw( amount );`

Array Algorithms are Everywhere

- Sorting & Searching (whole book dedicated to it [Knuth 1998]);
- Representing characters in a game;
- Retrieving/changing pixel colours;
- ...

Linear Search

Input Array with values.[1]

- □ We make no assumptions about order of the values.

Question Does the array has a certain value?

- □ Usually you're looking for a specific value.
- □ You may also be looking for more general properties.

Output Depends on two cases:

array has such value Any index that has such value;

array doesn't have it A negative value.

[1] The word value may also mean object reference value.

The Algorithm

Java

```
public static int linearSearch( final int[] array ) {
    int index = 0;


    while ((index < array.length) && (!satisfies( array[ index ] ))) {
        index++;


    }




    return (index < array.length) ? index : -1;
}

...
private static boolean satisfies( final int number ) {
    return number == 42;
}
```

The Algorithm

Java

```
public static int linearSearch( final int[] array ) {
    int index = 0;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index
    while ((index < array.length) && (!satisfies( array[ index ] ))) {
        index++;
        // index <= array.length and
        // !satisfies( array[ prev ] ) for 0 <= prev < index.
    }
    // (index <= array.length) and
    // (!satisfies( array[ prev ] ) for 0 <= prev < index) and
    // ((index >= array.length) || (satisfies( array[ index ] )))

    return (index < array.length) ? index : -1);
}

...
private static boolean satisfies( final int number ) {
    return number == 42;
}
```

The Algorithm

Java

```
public static int linearSearch( final int[] array ) {
    int index = 0;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index
    while ((index < array.length) && (!satisfies( array[ index ] ))) {
        index++;
        // index <= array.length and
        // !satisfies( array[ prev ] ) for 0 <= prev < index.
    }
    // (index <= array.length) and
    // (!satisfies( array[ prev ] ) for 0 <= prev < index) and
    // ((index >= array.length) || (satisfies( array[ index ] )))

    return (index < array.length) ? index : -1);
}

...
private static boolean satisfies( final int number ) {
    return number == 42;
}
```

The Algorithm

Java

```
public static int linearSearch( final int[] array ) {
    int index = 0;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index
    while ((index < array.length) && (!satisfies( array[ index ] ))) {
        index++;
        // index <= array.length and
        // !satisfies( array[ prev ] ) for 0 <= prev < index.
    }
    // (index <= array.length) and
    // (!satisfies( array[ prev ] ) for 0 <= prev < index) and
    // ((index >= array.length) || (satisfies( array[ index ] )))

    return (index < array.length) ? index : -1);
}

...
private static boolean satisfies( final int number ) {
    return number == 42;
}
```

The Algorithm

Distinguishing Cases: `index < array.length || index == array.length`

Java

```
public static int linearSearch( final int[] array ) {
    int index = 0;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index
    while ((index < array.length) && (!satisfies( array[ index ] ))) {
        index++;
        // index <= array.length and
        // !satisfies( array[ prev ] ) for 0 <= prev < index.
    }
    // (index <= array.length) and
    // (!satisfies( array[ prev ] ) for 0 <= prev < index) and
    // ((index >= array.length) || (satisfies( array[ index ] )))

    return (index < array.length) ? index : -1);
}

...
private static boolean satisfies( final int number ) {
    return number == 42;
}
```

The Algorithm

Distinguishing Cases: `index < array.length || index == array.length`

Java

```
public static int linearSearch( final int[] array ) {
    int index = 0;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index
    while ((index < array.length) && (!satisfies( array[ index ] ))) {
        index++;
        // index <= array.length and
        // !satisfies( array[ prev ] ) for 0 <= prev < index.
    }
    // (index <= array.length) and
    // (!satisfies( array[ prev ] ) for 0 <= prev < index) and
    // ((index >= array.length) || (satisfies( array[ index ] )))

    return (index < array.length) ? index : -1);
}

...
private static boolean satisfies( final int number ) {
    return number == 42;
}
```

The Algorithm

Distinguishing Cases: index < array.length || index == array.length

Java

```
public static int linearSearch( final int[] array ) {
    int index = 0;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index
    while ((index < array.length) && (!satisfies( array[ index ] ))) {
        index++;
        // index <= array.length and
        // !satisfies( array[ prev ] ) for 0 <= prev < index.
    }
    // (index <= array.length) and
    // (!satisfies( array[ prev ] ) for 0 <= prev < index) and
    // ((index >= array.length) || (satisfies( array[ index ] )))

    return (index < array.length) ? index : -1);
}

...
private static boolean satisfies( final int number ) {
    return number == 42;
}
```

Binary Search

Input *Ordered* array `keys` of keys.

Indices `lo` and `hi` of first and last key.

Question Does “`keys[lo..hi]`” has a certain value?

- Usually you’re looking for a specific value.
- You may also be looking for more general properties.

Output Depends on two cases:

array has such value Any index that has such value;
array doesn’t have it A negative value.

Dictionary Search

1. If `hi < lo` then `key` is not in `keys`.
2. Else assign `(lo + hi) / 2` to `mid`.

   Splits `keys` into three parts:

   (I) Keys before position `mid`. Keys are in `keys[ lo..mid - 1 ]`.
   (II) Keys after position `mid`. Keys are in `keys[ mid + 1..hi ]`.
   (III) Key `keys[ mid ]`.

   (I) and (II) approximately half the size of `keys[ lo..hi ]`.
3. There are three possibilities:

   (I) If `keys[ mid ] > key`, search in `keys[ lo..mid - 1 ]`.
   (II) If `keys[ mid ] < key`, search in `keys[ mid + 1..hi ]`.
   (III) Else `key` is in `keys` at position `mid`.

The Algorithm

We Assume Input are ints Sorted from Small to Large

Java

```
private static int binarySearch( final int[] keys, final int key ) {
    return binarySearch( keys, key, 0, keys.length - 1 );
}

private static int binarySearch( final int[] keys, final int key, int lo, int hi ) {
   boolean found = false;

   while ((lo <= hi) && (!found)) {
      final int mid = (lo + hi) / 2;

      if (key < keys[ mid ]) {
         hi = mid - 1;
      } else if (key > keys[ mid ]) {
         lo = mid + 1;
      } else {
         lo = mid;
         found = true;
      }
   }

   return (found) ? lo : -1;
}
```

Array Lists

- Java arrays have several disadvantages.
  - They can’t grow;
  - They can’t shrink;
  - There are no immutable arrays.
- This is why Java introduced classes similar to arrays.
- One of these classes is the ArrayList class.
- An ArrayList can do more than an array.
- You need a different notation to use an ArrayList.

Creation

Creates an Empty ArrayList for Strings

- ArrayList is a *generic class.*
- Generic classes are parameterised over one or more *object* types.
- You write the types inside angular brackets.
  - Use commas as separators.
- Java SE 7 introduced the diamond notation for the constructor.

Java

```java
final ArrayList<String> names = new ArrayList<>( );
```

Creation

Creates an Empty ArrayList for Strings

- ArrayList is a *generic class*.
- Generic classes are parameterised over one or more *object* types.
- You write the types inside angular brackets.
  - Use commas as separators.
- Java SE 7 introduced the diamond notation for the constructor.

Java

```
final ArrayList<String> names = new ArrayList<>( );
```

Creation

Creates an Empty `ArrayList` for `Strings`

- `ArrayList` is a *generic class.*
- Generic classes are parameterised over one or more *object* types.
- You write the types inside angular brackets.
  - Use commas as separators.
- `Java SE 7` introduced the diamond notation for the constructor.

Java

```
final ArrayList<String> names = new ArrayList<String>( );
```

Creation

Creates an Empty ArrayList for Strings

- □ ArrayList is a *generic class*.
- □ Generic classes are parameterised over one or more *object* types.
- □ You write the types inside angular brackets.
  - □ Use commas as separators.
- □ Java SE 7 introduced the diamond notation for the constructor.

Java

```
final ArrayList<String> names = new ArrayList<>( );
```

Creation

Creates an Empty HashMap from String to String (Not Studied Yet)

- ArrayList is a *generic class.*
- Generic classes are parameterised over one or more *object* types.
- You write the types inside angular brackets.
  - Use commas as separators.
- Java SE 7 introduced the diamond notation for the constructor.

Java

```
final ArrayList<String> names = new ArrayList<>( );
final HashMap<String,String> map = new HashMap<String,String>( );
```

Creation

Creates an Empty HashMap from String to String (Not Studied Yet)

- ArrayList is a *generic class.*
- Generic classes are parameterised over one or more *object* types.
- You write the types inside angular brackets.
  - Use commas as separators.
- Java SE 7 introduced the diamond notation for the constructor.

Java

```
final ArrayList<String> names = new ArrayList<>( );
final HashMap<String,String> map = new HashMap<>( );
```

Using

- Add a value to the end of an `ArrayList`:

Java

```
names.add( "Java Joe" );
```

- Read the value at a given position:

Java

```
final String firstName = names.get( 0 );
```

- Change the value at a given position:

Java

```
names.set( 0, "Java Jane" );
```

Using

- Add a value to the end of an `ArrayList`:

Java

```
names.add( "Java Joe" );
```

- Read the value at a given position:

Java

```
final String firstName = names.get( 0 );
```

- Change the value at a given position:

Java

```
names.set( 0, "Java Jane" );
```

Using

- □ Add a value to the end of an `ArrayList`:

Java

```java
names.add( "Java Joe" );
```

- □ Read the value at a given position:

Java

```java
final String firstName = names.get( 0 );
```

- □ Change the value at a given position:

Java

```java
names.set( 0, "Java Jane" );
```

Using (Continued)

- Get the size:

Java

```
final int size = names.size( );
```

- Remove the value at a certain position:

Java

```
names.delete( 0 );
```

- Remove first object if present; this returns `true` if successful:

Java

```
if (names.delete( "Elvis" )) {
    // Uses deep equality
    System.out.println( "Elvis has Left" );
}
```

Using (Continued)

- □ Get the size:

Java

```
final int size = names.size( );
```

- □ Remove the value at a certain position:

Java

```
names.delete( 0 );
```

- □ Remove first object if present; this returns `true` if successful:

Java

```
if (names.delete( "Elvis" )) {
    // Uses deep equality
    System.out.println( "Elvis has Left" );
}
```

Using (Continued)

- Get the size:

Java

```
final int size = names.size( );
```

- Remove the value at a certain position:

Java

```
names.delete( 0 );
```

- Remove first object if present; this returns `true` if successful:

Java

```
if (names.delete( "Elvis" )) {
    // Uses deep equality
    System.out.println( "Elvis has Left" );
}
```

Wrapper Classes

- Generic classes are parameterised over object types only.
- Means you cannot put primitive type values in them.
- Fortunately, `Java` has a *wrapper class* for each primitive type.

`Integer` For `int`s:

- `final Integer iObject = new Integer( 42 );`
- `final int val = iObject.intValue( );`

`Double` For `double`s:

- `final Double dObject = new Double( 3.14 );`
- `final double val = dObject.doubleValue( );`

`Boolean` For `boolean`s:

- `final Boolean bObject = new Boolean( true );`
- `final boolean val = bObject.booleanValue( );`

....

Autoboxing and Unboxing

- Writing code to convert to and from wrapper classes is tedious.
    - You must type more.
    - In increases the code size.
- That's why Java automates (some) conversions.
    - Automatic conversion to the wrapper class is called *autoboxing*.
    - Automatic conversion from the wrapper class is called *unboxing*.
- The conversion is done at runtime.

Autoboxing

- Let `val` be an value with primitive type type.
  - If you use `val` and `Java` expects an object, `Java` will autobox `val`.
- The type of `val` determines the wrapper class:
  - `int` ↦ `Integer`;
  - `double` ↦ `Double`;
  - `boolean` ↦ `Boolean`;
  - ...

Autoboxing (First Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );

objects.add( new Integer( 0 ) );
objects.add( 42 );
objects.add( 3.14 );
```

Autoboxing (First Example)

Java

```java
final ArrayList<Object> objects = new ArrayList<Object>( );

objects.add( new Integer( 0 ) );
objects.add( 42 );
objects.add( 3.14 );
```

Autoboxing (First Example)

Java

```java
final ArrayList<Object> objects = new ArrayList<Object>( );

objects.add( new Integer( 0 ) ); // Adds object
objects.add( 42 );
objects.add( 3.14 );
```

Autoboxing (First Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );

objects.add( new Integer( 0 ) ); // Adds object: Grand
objects.add( 42 );
objects.add( 3.14 );
```

Autoboxing (First Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );

objects.add( new Integer( 0 ) ); // Adds object: Grand
objects.add( 42 );
objects.add( 3.14 );
```

Autoboxing (First Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );

objects.add( new Integer( 0 ) ); // Adds object: Grand
objects.add( 42 );   // autoboxing: adds object
objects.add( 3.14 );
```

Autoboxing (First Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );

objects.add( new Integer( 0 ) ); // Adds object: Grand
objects.add( 42 );   // autoboxing: adds object: Grand
objects.add( 3.14 );
```

Autoboxing (First Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );

objects.add( new Integer( 0 ) ); // Adds object: Grand
objects.add( 42 );   // autoboxing: adds object: Grand
objects.add( 3.14 );
```

Autoboxing (First Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );

objects.add( new Integer( 0 ) ); // Adds object: Grand
objects.add( 42 );    // autoboxing: adds object: Grand
objects.add( 3.14 ); // autoboxing: adds object
```

Autoboxing (First Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );

objects.add( new Integer( 0 ) ); // Adds object: Grand
objects.add( 42 );   // autoboxing: adds object: Grand
objects.add( 3.14 ); // autoboxing: adds object: Grand
```

Autoboxing (Second Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );

ints.add( new Integer( 0 ) );
ints.add( 42 );
ints.add( 3.14 );
```

Autoboxing (Second Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );

ints.add( new Integer( 0 ) );
ints.add( 42 );
ints.add( 3.14 );
```

Autoboxing (Second Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );

ints.add( new Integer( 0 ) ); // Adds object
ints.add( 42 );
ints.add( 3.14 );
```

Autoboxing (Second Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );

ints.add( new Integer( 0 ) ); // Adds object: Grand
ints.add( 42 );
ints.add( 3.14 );
```

Autoboxing (Second Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );

ints.add( new Integer( 0 ) ); // Adds object
ints.add( 42 );
ints.add( 3.14 );
```

Autoboxing (Second Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );

ints.add( new Integer( 0 ) ); // Adds object
ints.add( 42 );   // autoboxing: adds Integer object
ints.add( 3.14 );
```

Autoboxing (Second Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );

ints.add( new Integer( 0 ) ); // Adds object
ints.add( 42 );   // autoboxing: adds Integer object: Grand
ints.add( 3.14 );
```

Autoboxing (Second Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );

ints.add( new Integer( 0 ) ); // Adds object
ints.add( 42 );   // autoboxing: adds Integer object: Grand
ints.add( 3.14 );
```

Autoboxing (Second Example)

Java

```java
final ArrayList<Integer> ints = new ArrayList<Integer>( );

ints.add( new Integer( 0 ) ); // Adds object
ints.add( 42 );   // autoboxing: adds Integer object: Grand
ints.add( 3.14 ); // autoboxing: adds Double object
```

Autoboxing (Second Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );

ints.add( new Integer( 0 ) ); // Adds object
ints.add( 42 );   // autoboxing: adds Integer object: Grand
ints.add( 3.14 ); // autoboxing: adds Double object: Compile time error
```

Unboxing

- Unboxing turns wrapper class objects to primitive type values.
- The wrapper class type determines the primitive type.
    - `Integer` ↦ `int`;
    - `Double` ↦ `double`;
    - `Boolean` ↦ `boolean`;
    - ...
- The conversion is done at runtime.

Unboxing (First Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 42 );

final int lucky = ints.get( 0 );
System.out.println( lucky );
```

Unboxing (First Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 42 );

final int lucky = ints.get( 0 );
System.out.println( lucky );
```

Unboxing (First Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 42 );

final int lucky = ints.get( 0 ); // unboxing
System.out.println( lucky );
```

Unboxing (First Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 42 );

final int lucky = ints.get( 0 ); // unboxing: Integer -> int
System.out.println( lucky );
```

Unboxing (First Example)

Java

```
final ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 42 );

final int lucky = ints.get( 0 ); // unboxing: Integer -> int (Grand)
System.out.println( lucky );
```

Unboxing (Second Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );
objects.add( 42 );
objects.add( 3.14 );

final int lucky = (Integer)objects.get( 0 );
System.out.println( lucky );

final int disaster = (Integer)objects.get( 1 );
System.out.println( disaster );
```

Unboxing (Second Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );
objects.add( 42 );
objects.add( 3.14 );

final int lucky = (Integer)objects.get( 0 );
System.out.println( lucky );

final int disaster = (Integer)objects.get( 1 );
System.out.println( disaster );
```

Unboxing (Second Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );
objects.add( 42 );
objects.add( 3.14 );

final int lucky = (Integer)objects.get( 0 ); // unboxing
System.out.println( lucky );

final int disaster = (Integer)objects.get( 1 );
System.out.println( disaster );
```

Unboxing (Second Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );
objects.add( 42 );
objects.add( 3.14 );

final int lucky = (Integer)objects.get( 0 ); // unboxing: Integer -> int
System.out.println( lucky );

final int disaster = (Integer)objects.get( 1 );
System.out.println( disaster );
```

Unboxing (Second Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );
objects.add( 42 );
objects.add( 3.14 );

final int lucky = (Integer)objects.get( 0 ); // unboxing: Integer -> int (Grand)
System.out.println( lucky );

final int disaster = (Integer)objects.get( 1 );
System.out.println( disaster );
```

Unboxing (Second Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );
objects.add( 42 );
objects.add( 3.14 );

final int lucky = (Integer)objects.get( 0 ); // unboxing: Integer -> int (Grand)
System.out.println( lucky );

final int disaster = (Integer)objects.get( 1 );
System.out.println( disaster );
```

Unboxing (Second Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );
objects.add( 42 );
objects.add( 3.14 );

final int lucky = (Integer)objects.get( 0 ); // unboxing: Integer -> int (Grand)
System.out.println( lucky );

final int disaster = (Integer)objects.get( 1 ); // unboxing
System.out.println( disaster );
```

Unboxing (Second Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );
objects.add( 42 );
objects.add( 3.14 );

final int lucky = (Integer)objects.get( 0 ); // unboxing: Integer -> int (Grand)
System.out.println( lucky );

final int disaster = (Integer)objects.get( 1 ); // unboxing: Double -> Integer
System.out.println( disaster );
```

Unboxing (Second Example)

Java

```
final ArrayList<Object> objects = new ArrayList<Object>( );
objects.add( 42 );
objects.add( 3.14 );

final int lucky = (Integer)objects.get( 0 ); // unboxing: Integer -> int (Grand)
System.out.println( lucky );

final int disaster = (Integer)objects.get( 1 ); // unboxing: Double -> Integer (Runtime Error)
System.out.println( disaster );
```

Caching

- □ `Java` *caches* a limited number of wrapper class values.
- □ Guarantees shallow equality for small number of boxed values.
  - □ If `o1.equals( o2 )` then `o1 == o2`.
- □ For example, `new Integer( 0 ) == new Integer( 0 )`.
- □ In general this may not always work:
  - □ Almost always: `new Integer( 666 ) != new Integer( 666 )`.
- □ Caching is implemented because it saves memory.
- □ In general caching works for “small” primitive values.

`boolean`:	`true` and `false`.
`byte`:	0–255.
`char`:	`\u0000`–`\u007f`.
`short`:	-128, -127, ..., 127.
`int`:	-128, -127, ..., 127.

Java

```
ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 0 );
ints.add( new Integer( -1 ) );
ints.add( 2 );
for (Integer i : ints) {
    ⟨use i⟩
}
```

ArrayLists are Iterable

Java

```java
ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 0 );
ints.add( -1 );
ints.add( 5 );

Iterator<Integer> iter = ints.iterator( );

// Remove all negative values.
while (iter.hasNext( )) {
    int next = iter.next( );
    if (next < 0) {
        iter.remove( );
    }
}
```

ArrayLists are Iterable

Java

```
ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 0 ); // autoboxing
ints.add( -1 ); // autoboxing
ints.add( 5 ); // autoboxing

Iterator<Integer> iter = ints.iterator( );

// Remove all negative values.
while (iter.hasNext( )) {
    int next = iter.next( );
    if (next < 0) {
        iter.remove( );
    }
}
```

ArrayLists are Iterable

Java

```
ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 0 ); // autoboxing
ints.add( -1 ); // autoboxing
ints.add( 5 ); // autoboxing

Iterator<Integer> iter = ints.iterator( );

// Remove all negative values.
while (iter.hasNext( )) {
    int next = iter.next( );
    if (next < 0) {
        iter.remove( );
    }
}
```

ArrayLists are Iterable

Java

```
ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 0 ); // autoboxing
ints.add( -1 ); // autoboxing
ints.add( 5 ); // autoboxing

Iterator<Integer> iter = ints.iterator( );

// Remove all negative values.
while (iter.hasNext( )) {
    int next = iter.next( );
    if (next < 0) {
        iter.remove( );
    }
}
```

Arrays
Array Algorithms
Array Lists
Introduction
Creation
Using
Wrapper Classes
Autoboxing & Unboxing
Caching
Enhanced for Loop
More Algorithms
For Monday
About this Document

ArrayLists are Iterable

Java

```
ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 0 ); // autoboxing
ints.add( -1 ); // autoboxing
ints.add( 5 ); // autoboxing

Iterator<Integer> iter = ints.iterator( );

// Remove all negative values.
while (iter.hasNext( )) {
    int next = iter.next( );
    if (next < 0) {
        iter.remove( );
    }
}
```

ArrayLists are Iterable

Java

```
ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 0 ); // autoboxing
ints.add( -1 ); // autoboxing
ints.add( 5 ); // autoboxing

Iterator<Integer> iter = ints.iterator( );

// Remove all negative values.
while (iter.hasNext( )) {
    int next = iter.next( ); // unboxing
    if (next < 0) {
        iter.remove( );
    }
}
```

ArrayLists are Iterable

Java

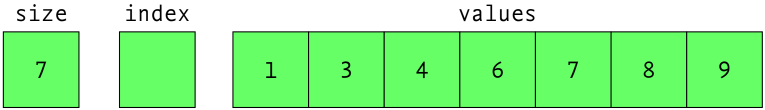
```java
ArrayList<Integer> ints = new ArrayList<Integer>( );
ints.add( 0 ); // autoboxing
ints.add( -1 ); // autoboxing
ints.add( 5 ); // autoboxing

Iterator<Integer> iter = ints.iterator( );

// Remove all negative values.
while (iter.hasNext( )) {
    int next = iter.next( ); // unboxing
    if (next < 0) {
        iter.remove( );
    }
}
```

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
    - The operations require much time.
- E.g., assume you delete the first member from a sorted array.
    - You can't have holes in the array.
    - All members have to be moved down one position.
    - If the array contains many members this will take long.

Java

```
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```

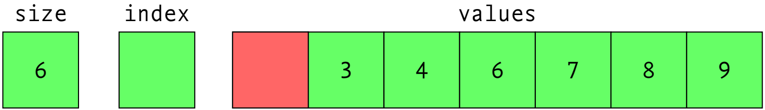


Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- E.g., assume you delete the first member from a sorted array.
  - You can't have holes in the array.
  - All members have to be moved down one position.
  - If the array contains many members this will take long.

Java

```java
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```



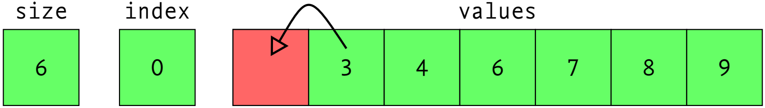
size	index	values						
6		🟥	3	4	6	7	8	9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- E.g., assume you delete the first member from a sorted array.
  - You can't have holes in the array.
  - All members have to be moved down one position.
  - If the array contains many members this will take long.

Java

```
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```

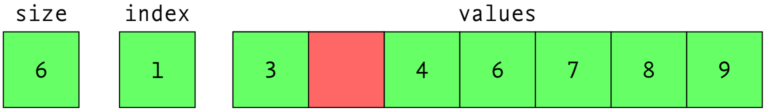


Modifying Sorted Arrays

- □ Applications such as binary search require sorted arrays.
- □ Unfortunately, this makes modifying such arrays expensive:
    - □ The operations require much time.
- □ E.g., assume you delete the first member from a sorted array.
    - □ You can't have holes in the array.
    - □ All members have to be moved down one position.
    - □ If the array contains many members this will take long.

Java

```
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```



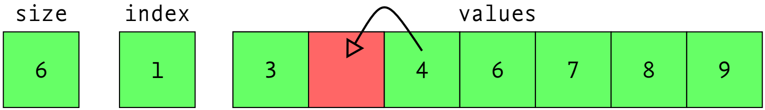
size	index	values						
6	1	3	🟥	4	6	7	8	9

Modifying Sorted Arrays

- □ Applications such as binary search require sorted arrays.
- □ Unfortunately, this makes modifying such arrays expensive:
  - □ The operations require much time.
- □ E.g., assume you delete the first member from a sorted array.
  - □ You can’t have holes in the array.
  - □ All members have to be moved down one position.
  - □ If the array contains many members this will take long.

Java

```
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```

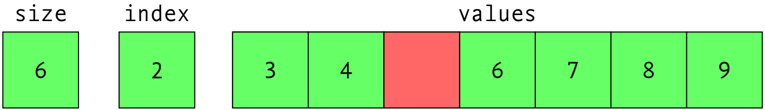


Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- E.g., assume you delete the first member from a sorted array.
  - You can't have holes in the array.
  - All members have to be moved down one position.
  - If the array contains many members this will take long.

Java

```java
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```

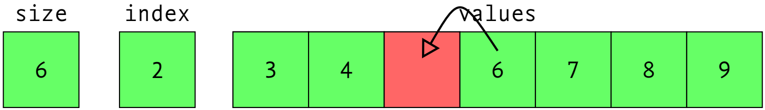


Modifying Sorted Arrays

- ☐ Applications such as binary search require sorted arrays.
- ☐ Unfortunately, this makes modifying such arrays expensive:
    - ☐ The operations require much time.
- ☐ E.g., assume you delete the first member from a sorted array.
    - ☐ You can't have holes in the array.
    - ☐ All members have to be moved down one position.
    - ☐ If the array contains many members this will take long.

Java

```
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```



size
6

index
2

values

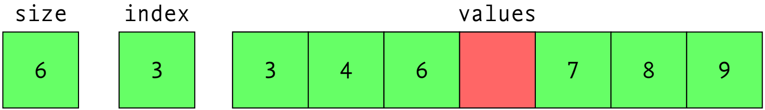
3	4		6	7	8	9

Modifying Sorted Arrays

- □ Applications such as binary search require sorted arrays.
- □ Unfortunately, this makes modifying such arrays expensive:
  - □ The operations require much time.
- □ E.g., assume you delete the first member from a sorted array.
  - □ You can't have holes in the array.
  - □ All members have to be moved down one position.
  - □ If the array contains many members this will take long.

Java

```
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```



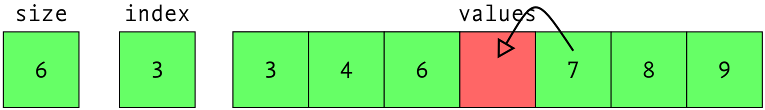
size	index	values						
6	3	3	4	6	🟥	7	8	9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- E.g., assume you delete the first member from a sorted array.
  - You can't have holes in the array.
  - All members have to be moved down one position.
  - If the array contains many members this will take long.

Java

```
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```



Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
    - The operations require much time.
- E.g., assume you delete the first member from a sorted array.
    - You can’t have holes in the array.
    - All members have to be moved down one position.
    - If the array contains many members this will take long.

Java

```
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```

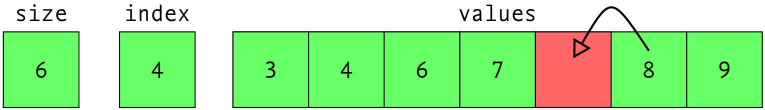
size	index	values						
6	4	3	4	6	7	🟥	8	9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- E.g., assume you delete the first member from a sorted array.
  - You can't have holes in the array.
  - All members have to be moved down one position.
  - If the array contains many members this will take long.

Java

```
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```

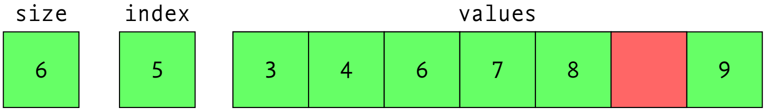


Modifying Sorted Arrays

- □ Applications such as binary search require sorted arrays.
- □ Unfortunately, this makes modifying such arrays expensive:
  - □ The operations require much time.
- □ E.g., assume you delete the first member from a sorted array.
  - □ You can't have holes in the array.
  - □ All members have to be moved down one position.
  - □ If the array contains many members this will take long.

Java

```java
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```



size

6

index

5

values

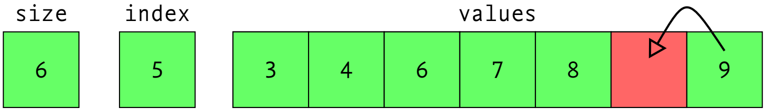
3	4	6	7	8	🟥	9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- E.g., assume you delete the first member from a sorted array.
  - You can’t have holes in the array.
  - All members have to be moved down one position.
  - If the array contains many members this will take long.

Java

```
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```



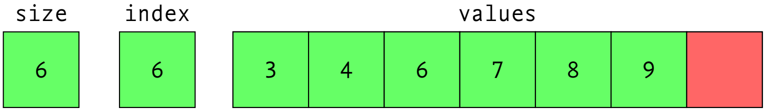
size	index	values						
6	5	3	4	6	7	8		9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- E.g., assume you delete the first member from a sorted array.
  - You can't have holes in the array.
  - All members have to be moved down one position.
  - If the array contains many members this will take long.

Java

```java
size--;
for (int index = 0; index != size; index++) {
    values[ index ] = values[ index + 1 ];
}
```



size

6

index

6

values

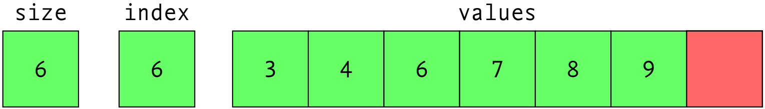
3	4	6	7	8	9	🟥

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You’re not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - ...

Java

```java
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```



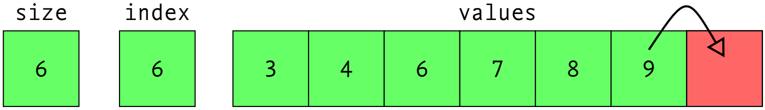
size	index	values						
6	6	3	4	6	7	8	9	🟥

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You’re not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - ...

Java

```java
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```



size

6

index

6

values

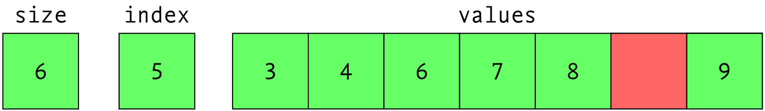
3 4 6 7 8 9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You’re not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - ...

Java

```java
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```



size
6

index
5

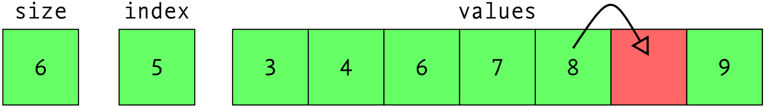
values
3 4 6 7 8 🟥 9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You're not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - ...

Java

```
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```



size: 6

index: 5

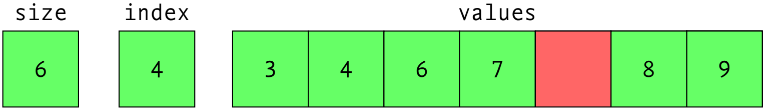
values: 3 | 4 | 6 | 7 | 8 | 🟥 | 9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You’re not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - ...

Java

```
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```

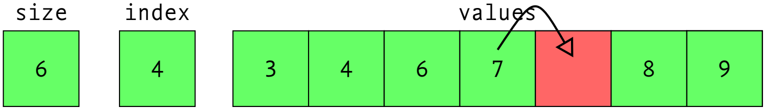


Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You’re not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - ...

Java

```
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```



Modifying Sorted Arrays

- □ Applications such as binary search require sorted arrays.
- □ Unfortunately, this makes modifying such arrays expensive:
    - □ The operations require much time.
- □ Likewise, assume you want to insert an element at the start.
    - □ You’re not allowed to overwrite the first member.
    - □ All members have to be moved up one position.
    - □ …

Java

```
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```

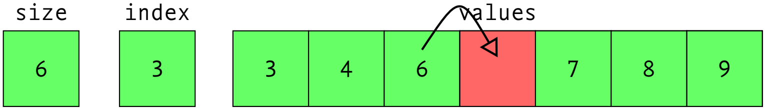
size	index	values						
6	3	3	4	6	🟥	7	8	9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You’re not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - …

Java

```
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```

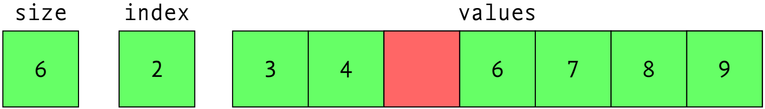


Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You're not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - ...

Java

```
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```



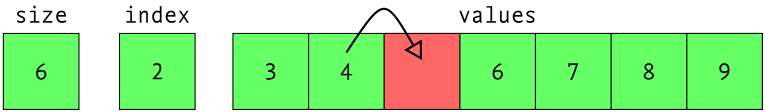
size	index	values						
6	2	3	4	🟥	6	7	8	9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You’re not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - ...

Java

```
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```



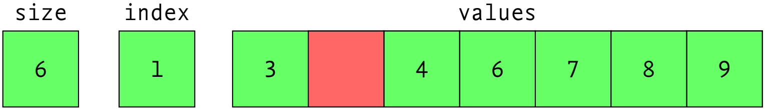
size	index	values						
6	2	3	4		6	7	8	9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You're not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - ...

Java

```java
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```



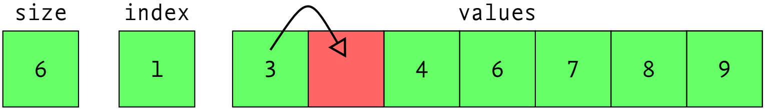
size	index	values						
6	1	3	🟥	4	6	7	8	9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
    - The operations require much time.
- Likewise, assume you want to insert an element at the start.
    - You’re not allowed to overwrite the first member.
    - All members have to be moved up one position.
    - ...

Java

```
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```



size: 6

index: 1

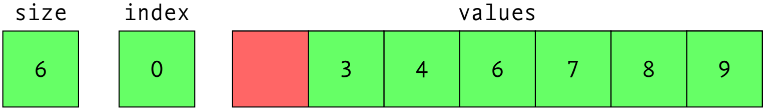
values: 3 🟥 4 6 7 8 9

Modifying Sorted Arrays

- □ Applications such as binary search require sorted arrays.
- □ Unfortunately, this makes modifying such arrays expensive:
  - □ The operations require much time.
- □ Likewise, assume you want to insert an element at the start.
  - □ You're not allowed to overwrite the first member.
  - □ All members have to be moved up one position.
  - □ ...

Java

```java
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```



size	index	values						
6	0	🟥	3	4	6	7	8	9

Modifying Sorted Arrays

- Applications such as binary search require sorted arrays.
- Unfortunately, this makes modifying such arrays expensive:
  - The operations require much time.
- Likewise, assume you want to insert an element at the start.
  - You're not allowed to overwrite the first member.
  - All members have to be moved up one position.
  - ...

Java

```java
for (int index = size; index != 0; index--) {
    values[ index ] = values[ index - 1 ];
}
size++;
```

size	index	values						
7	0	-1	3	4	6	7	8	9

Prime Numbers

- A positive integer is called a *prime* if it only has two proper positive integer divisors.
- For example 2, 3, 5, 7, 11, 13, 17, ...
- There are infinitely many primes.
  - For example, assume the contrary.
  - Let p be the product of all primes.
  - Then $p + 1 \bmod i = 1$ for all integers i such that $2 \leq i \leq p$.
  - A contradiction.

Sieve of Erathostenes

- There's a famous algorithm for computing prime numbers.
- The algorithm is called *the Sieve of Erathostenes.*
- The algorithm is very simple.
- It starts with an empty list of known primes.
  - Next it enumerates all integers greater than 1.
    - (Up to some maximum number.)
  - Let i be the next current integer.
  - If no known prime divides i, then it adds i to its known primes.
- The algorithm is ideal for `ArrayLists`.

The Algorithm

Java

```
final ArrayList<Integer> sieve = new ArrayList<>( );

for (int candidate = 2; candidate <= 100; candidate++) {
    int index = 0;

    while ((index != sieve.size( ))
               && (candidate % sieve.get( index ) != 0)) {
        index++;
    }

    if (index == sieve.size( )) {
       sieve.add( candidate );
    }
}

System.out.println( primes );
// prints: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
// 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

- This lecture is based on [Horstmann 2013, Chapter 6.1 – 6.3].

For Monday

For Monday

Enjoy Guinness Sensibly

For Tuesday

- I’ll post Assignment 3 on Tuesday.

For Wednesday

- Study [Horstmann 2013, Chapter 6].
- Prove that the linear and binary search algorithms terminate.
- Prove that binary search is correct.
- Answer [Horstmann 2013, R6.23].
- Carry out [Horstmann 2013, P6.4 and P6.7].

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.