Repeated Arguments
Sources of Aliasing

• Aliased references for mutable types can cause trouble, so it is important to know how aliases might arise
• One way (which is easy to recognize and easy to record in a tracing table, using ➔) is the *simple assignment* of one reference variable to another
• There are other sources of aliasing as well...
Aliasing from Parameter Passing

• Because a formal parameter of a reference type is initialized by *copying* the corresponding argument’s reference value (which is tantamount to assignment of the argument to the formal parameter), *parameter passing* is another source of aliasing
Example

- Consider this method:

```java
/**
 * Adds 1 to the first number and 2 to the second.
 * ...
 * @updates x, y
 * @ensures
 * x = #x + 1 and y = #y + 2
 */

private static void foo(NaturalNumber x,
            NaturalNumber y) { ...}
```
Example

• Consider this method:

```java
/**
 * Adds 1 to the first and 2 to the second.
 * ...
 * @updates x, y
 * @ensures
 *  x = #x + 1   and   y = #y + 2
 */
private static void foo(NaturalNumber x,
            NaturalNumber y) {...}
```

How would you implement this contract specification?
Example: A Call

• Consider this call of the method:

```java
NaturalNumber a = new NaturalNumber2(10);
NaturalNumber b = new NaturalNumber2(319);
foo(a, b);
```

• How does this get executed, and what values result for a and b?
How Calls Work In Java

```java
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}
```

```java
NaturalNumber a = new NaturalNumber2(10);
NaturalNumber b = new NaturalNumber2(319);
foo(a, b);
```
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}

NaturalNumber a = new NaturalNumber2(10);
NaturalNumber b = new NaturalNumber2(319);
foo(a, b);
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}

NaturalNumber a = new NaturalNumber2(10);
NaturalNumber b = new NaturalNumber2(319);
foo(a, b);
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}

NaturalNumber a = new NaturalNumber2(10);
NaturalNumber b = new NaturalNumber2(319);
foo(a, b);
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}

NaturalNumber a = new NaturalNumber2(10);
NaturalNumber b = new NaturalNumber2(319);
foo(a, b);
Note: Harmless Aliasing

• Aliases are created, but since the method body for \( \text{foo} \) only has access to the variables \( x \) and \( y \) (i.e., the variables used as arguments in the client code, \( a \) and \( b \), are not in scope while the body of \( \text{foo} \) is executing), these aliases cause no trouble for reasoning
Example: A Different Call

• Now consider this call of the method:

```java
NaturalNumber a = new NaturalNumber2(10);
foo(a, a);
```

• How does this happen, and what value results for `a`?
How Calls Work In Java

```java
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}
```

```java
NaturalNumber a = new NaturalNumber2(10);
foo(a, a);
```
How Calls Work In Java

```java
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}
```

```java
NaturalNumber a = new NaturalNumber2(10);
foo(a, a);
```
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}

NaturalNumber a = new NaturalNumber2(10);
foo(a, a);
**How Calls Work In Java**

```java
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}
```

```java
NaturalNumber a = new NaturalNumber2(10);
foo(a, a);
```

26 April 2013  
OSU CSE
How Calls Work In Java

```java
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}
```

```java
NaturalNumber a = new NaturalNumber2(10);
foo(a, a);
```

Can we really be sure the resulting value is 13? Perhaps surprisingly, no!
How Calls Work In Java

```java
public static void foo(
    NaturalNumber x, NaturalNumber y) {
    ...
}
```

```java
NaturalNumber a = new NaturalNumber2(10);
foo(a, a);
```
Note: Harmful Aliasing

• Here, aliases are created between two variables that are \textit{in scope} while the method body for \texttt{foo} is executing (i.e., the variables \texttt{x} and \texttt{y}), and these aliases do cause trouble for reasoning

• Who is at fault for this anomalous outcome?
  – The implementer of \texttt{foo}?
  – The client of \texttt{foo}?
What Outcome Was Expected?

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = 10</code></td>
<td></td>
</tr>
<tr>
<td><code>foo(a, a);</code></td>
<td></td>
</tr>
</tbody>
</table>
What Outcome Was Expected?

Consult the contract for `foo`, substituting a for both parameters `x` and `y`; it ensures:

\[
a = 11 \quad \text{and} \quad a = 12
\]

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = 10</code></td>
<td></td>
</tr>
<tr>
<td><code>foo(a, a);</code></td>
<td></td>
</tr>
<tr>
<td><code>a = 10</code></td>
<td></td>
</tr>
</tbody>
</table>
What Outcome Was Expected?

Can we \textit{really} have this outcome?

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = 10 )</td>
</tr>
<tr>
<td>( a = 11 )</td>
</tr>
<tr>
<td>( a = 12 )</td>
</tr>
</tbody>
</table>

```c
foo(a, a);
```
Repeated Arguments

• In this case, it would be *impossible* for any implementation of `foo` to produce the outcome supposedly ensured according to its contract!

• The trouble arising from *repeated arguments* (i.e., a call like `foo(a, a)`) is not just in Java; it is a problem in any language with mutable types
The Receiver Is An Argument

• Note that the reference value of the receiver of a call (to an instance method) is copied to the formal parameter known as this
• Hence, there is a repeated argument if the receiver is also passed as another argument to such a call
• Example:
  \texttt{n.add(n);}
The Receiver Is An Argument

• Note that the reference value of the receiver of a call (to an instance method) is copied to the formal parameter known as this.

• Hence, there is a repeated argument if the receiver is also passed as another argument to such a call.

• Example:

```
n.add(n);
```

Does this call double n, as you might expect from using informal reasoning and “wishful naming” to predict the outcome?
The Receiver Is An Argument

• Note that the reference to the receiver of a call (to an instance method) is copied to the formal parameter known as this.

• Hence, there is a repeated argument if the receiver is also passed as another argument to such a call.

• Example:

```java
n.add(n);
```

Why, given the contract for `add`, can this call simply not be a good idea?
Best Practice for Repeated Arguments

• Never pass any variable of a mutable reference type as an argument \textit{twice} or more to a single method call
  – Remember that the receiver is an argument
• Checkstyle and FindBugs do not warn you about this!