

INTERFACES

TERMINOLOGY

Data Type

- A *data type* is kind of value and the operations that can be applied to those values.
- Examples:
 - integers; addition, subtraction, multiplication, etc.
 - Strings; concatenation, length, substring, etc.
- Java has primitive and object data types.
 - Eight primitive types: int, short, byte, long, double, float, boolean, char
 - Object types: defined by classes (e.g., String, Scanner, Movie, etc.)

Data Structure

- An aggregate, or collection, of data values and the operations supported by that collection
- Examples
 - Array
 - List
 - Bag
 - Queue
 - Stack
 - Tree

Abstract Data Type

- An ADT is a logical specification of a data type in which the implementation details are hidden from the user.
 - The ADT specifies what the data type can do. But the details of *how it is done* are hidden.
- In Java, we create ADTs by creating interfaces
 - Interfaces define types that can be used to declare objects
 - Classes implement those interfaces and specify the implementation details

INTERFACES IN JAVA

Interfaces

- A Java *interface* is a collection of abstract methods and constants
 - An abstract method can be declared with the modifier `abstract`
- An interface is used to establish a set of methods that a class will implement
 - It's like a contract

Interfaces

- An interface is declared with the reserved word `interface`
- A class indicates that it is implementing an interface with the reserved word `implements` in the class header
- As of Java 8, interfaces can now also contain *default methods*, which are implemented.
 - We won't be focusing on these.

Interface Syntax

interface is a reserved word



```
public interface Doable {  
    void doThis();  
    int doThat(int num);  
}
```

Often public and abstract are left off since these are the defaults.

None of the methods in an interface are given a definition (body)



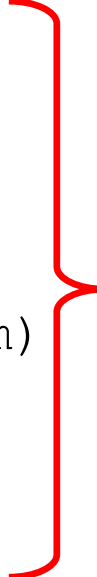
A semicolon immediately follows each method header

Interface Syntax

```
public class CanDo implements Doable{  
    @Override  
    public void doThis ()    {  
        // whatever  
    }  
  
    @Override  
    public void doThat (int num)  
        // whatever  
    }  
  
    // other methods in the class  
}
```



**implements is a
reserved word**



**Each method listed
in Doable is
given a definition**

Interface Constants

- Interfaces can also provide public, final, static constants.

Properties of Interfaces

- An interface cannot be instantiated
- Methods of an interface have public visibility
- If a parent class implements an interface, then by definition, all child classes do as well.
 - That functionality is inherited!

Properties of Classes that Implement an Interface

- Provide implementations for *every* method in the interface
 - Can choose whether to override default methods.
- Can have additional methods as well
- Have access to the constants in that interface
- Can implement multiple interfaces but must implement all methods in each interface

```
class DoesALot implements interface1, interface2 {  
    // all methods of both interfaces  
}
```

Using Interfaces

- Interfaces describe common *functionality* across classes rather than common *features* (which is more suited for inheritance)
 - Inheritance “is a”
 - Interface “does ...” “can ...” “is ...able”
- Interfaces are Java’s way of ensuring that a class contains an implementation for a specific method.
 - That an object has a specific functionality.

Interfaces and Polymorphism

- An interface can be used as a declared type.
 - But not as an actual type, since you cannot instantiate it!
- The variable can be instantiated with any class that implements the interface
 - The method that is invoked is based on the actual type.

Interfaces and Polymorphism

```
public interface Speaker {  
    public abstract void speak();  
}  
  
public class Dog extends Animal implements Speaker {  
    public void speak() {  
        System.out.println("Woof");  
    }  
}  
  
public class Parrot extends Bird implements Speaker {  
    public void speak() {  
        System.out.println("Polly wants a cracker...");  
    }  
}
```

```
Speaker s1 = new Dog(); // allowed  
Speaker s2 = new Parrot(); // allowed  
Speaker s3 = new Speaker(); // NOT allowed! compiler error!
```


Interfaces and Polymorphism

```
public interface Speaker {  
    public abstract void speak();  
}  
  
public class Dog extends Animal implements Speaker {  
    public void speak() { System.out.println("Woof"); }  
}  
  
public class Parrot extends Bird implements Speaker {  
    public void speak() { System.out.println("Polly wants a cracker...") }  
}
```

```
Speaker[] speakers = new Speaker[2]; // allowed  
speakers[0] = new Parrot(); // declared type Speaker, actual type Parrot  
speakers[1] = new Dog(); // declared type Speaker, actual type Dog  
for(Speaker sp : speakers) {  
    sp.speak();  
    // allowed because the method speak exists in the declared class  
    // Speaker; at runtime, the version will be implemented based on the  
    // actual type (Dog, Parrot, etc.)  
}  
  
ArrayList<Speaker> speakerList = new ArrayList<Speaker>(); // allowed
```

Abstract Classes vs. Interfaces

	Abstract Classes	Interfaces
Can be used as a declared type?	Yes	Yes
Can be instantiated (used as an actual type)?	No	No
Can contain constructors?	Yes	No
Can contain abstract methods?	Yes	Yes
Can contain non-abstract, implemented methods?	Yes	As of Java 8, yes (default methods)
Can contain public, static, final constants?	Yes	Yes
Can contain variables that are not public, static, final?	Yes	No
Other classes can...	Inherit from only one class	Implement multiple interfaces

Programming to the Interface

- We often use code like this:
 - `ListInterface<Integer> numberList= new AList<>();`
 - declared type is an interface, instantiated type is a class that implements that interface
 - `BagInterface<String> wordBag= new ArrayBag<>();`
 - `List<Student> studentList = new ArrayList<Student>();`
- In this code, the declared type is the interface (or could be a parent class). The actual type is the concrete (child or implementing) class.
- This is a general principle known as *programming to the interface*.
 - For this principle, we mean interface in a general sense, not "a Java interface."

Programming to the Interface

- Programming to the interface promotes code that is more flexible and maintainable.
- Declare objects to be the most general (or high up the inheritance tree) type as possible to suit your needs.
- Example: If you're going to treat your object just as a list throughout the code, then it's best to declare it as a List object, even though the actual type is ArrayList.

Programming to the Interface

- Another way to think about it is to declare objects based just on *what they will do* (often specified in a Java interface) rather than *how they will do it* (as implemented by a concrete class).

Programming to the Interface

- Why? Better logical representation.
 - For the list example, we just need a list- we don't really care what kind of list it is, we are just going to invoke methods that could apply to *any* list implementation.
- Why? Technical benefit.
 - We might discover some new, more efficient implementation of a list. If that happens, we could swap out `ArrayList<>()` with `NewImprovedList<>()` in one line of code, and the rest of the code would continue to compile and run correctly.

Javadoc

- A style of comments
- The javadoc tool can parse these comments and pull out information about classes and methods
- This information can be used to generate API pages
- javadoc style comments are critical for interfaces because they specify the operations associated with an ADT
- Oracle javadoc guide:
<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

CLIENT VS. IMPLEMENTATION PERSPECTIVE

Client Perspective

- The perspective you take when *using the interface* to accomplish a task
- Also called the *logical* or *conceptual* perspective
- Examples:
 - Using a list to keep track of customers
 - Using a list to keep track of purchases
 - Using a set to keep track of students

Client Perspective

- The client only sees the interface. The client does not see the implementation
- The client can use all methods defined in the interface, but they don't know how they are implemented behind the scenes.
- The client relies on the public methods in the interface (described by the javadoc comments) to know how the interface works

Implementation Perspective

- The perspective you take when *developing* a class that implements an interface
- Also called the *physical* or *actual* perspective
- Examples:
 - Implementing a List with linked nodes
 - Implementing a Set with an array

Implementation Perspective

- The developer must implement all methods defined in the interface and should adhere to the description contained in the javadoc comments
- The developer gets to decide *how* to implement the methods
- The developer makes decisions about what data structures and data types to use behind the scenes

Playing Both Roles

- In our class, as in real life, you will sometimes act as the client and sometimes act as the developer.

LISTS

The List ADT

- A list is an ordered, indexed organization of items.
- Lists allow you to add items, remove items, and retrieve items.

The List ADT

- In this class, we'll be using two interfaces
 - List (Java standard library)
 - ListInterface (textbook)
- List is what you'll use in the real world
- ListInterface will allow us to go behind the scenes!

The Java List Interface

- List.java:

<https://docs.oracle.com/javase/10/docs/api/java/util/List.html>

ListInterface.java

- ListInterface objects start at position 1!
- Ways to alter the list
 - Two ways to add items
 - end of the list
 - specific position in the list
 - One way to remove items
 - based on position
 - One way to clear the list
 - One way to replace an item
- Ways to access the list
 - Retrieve an item at a specific position
 - Check if an item is on the list
 - Get the length of the list
 - Check if the list is empty
 - Get an array with the list's contents

Example

- Write code at the client level to remove the lowest score from a list of quiz scores.
 - Use a ListInterface object.
 - Use a List object.

BAGS

The Bag ADT

- A bag is an unordered organization of items.
- Bags allow you to add items, remove an item, test if an item is in the bag, and determine the size of the bag.
- Bags allow duplicate values.

BagInterface.java

- Ways to alter the bag
 - One way to add an item to the bag
 - Two ways to remove items
 - remove an unspecified item
 - remove a specific item
 - One way to empty the bag
- Ways to access the bag
 - Check how many items are in the bag
 - Check whether the bag is empty
 - Check whether the bag contains a certain item and how many times
 - Get an array with the bag's contents

Example

- Write code at the client level to remove the smallest score from a bag of quiz scores.

CLIENT PERSPECTIVE, REVISITED

The Client Perspective

- To write code as the client, you must study the methods available and make sure you understand the method inputs (parameters) and method outputs (return values).
- You must understand how the methods work in order to use them!
- Remember that you have no idea how the methods are implemented!

The Client Perspective

- You'll need to act as both the client and the developer in our course.
- In this module, we focus only on the client perspective.
- In homework and exams, make sure you are careful about what role I am asking you to take!
 - Acting as the client means can only use the existing methods provided by the interface
 - Acting as the developer means you will be writing code that implements an interface

Writing Code as the Client

- What kind of data structure do I need?
- Should I restrict what kind of objects go into the data structure?
- What should happen if the data structure is empty? singleton?
- What interface specifications do I need to adhere to?
- Check that I am:
 - Only using methods provided in the interface
 - Implementing methods on an object

Examples

- Write code statements from the client level to print only the Strings in a bag that contain a certain character. The bag should not be altered.
- Write a complete method to count of all of the negative integers in a list.
 - Use ListInterface and then List.

PASSING OBJECTS AS PARAMETERS

Passing Objects as Parameters

- When you pass an object as parameter, the **value** of the object is sent.
- For objects, the value is the memory address.
- This means you create an alias between the formal and actual parameter.
- Changes made to the object inside the method will affect the object outside the method.
 - Because it's the same object!

Passing Parameters

- When a method is invoked, behind the scenes you have the code
formalParameter = actualParameter
- When the parameter is an object, this means you create an alias

Review the ListAsParameterExample

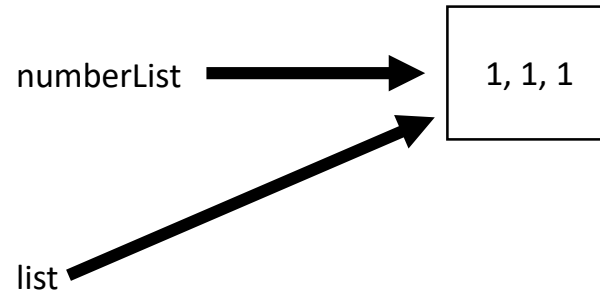
- In this example, we pass a list into a method as a parameter.
- We want to make a copy of that list in the method and do something with that copy.
- **But we don't want the list to be altered when the method ends.**
- We'll look at the correct way to do this and three commonly seen incorrect ways.

Main



Incorrect Non Copy Method Trace

```
incorrectNonCopyMethod(numberList);
```

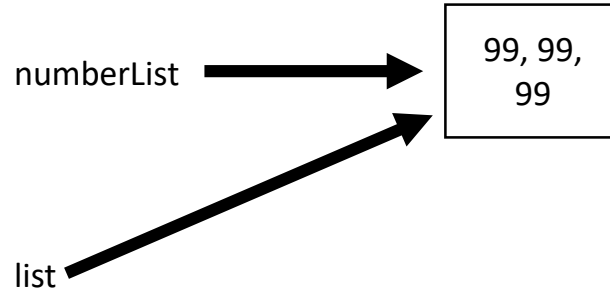


Incorrect Non Copy Method Trace

```
// loop to modify
```

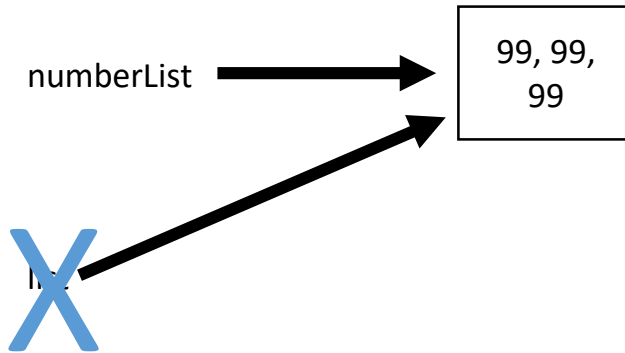
This is the error!!!

**Our formal parameter
and actual parameter
are aliases. So
modifying inside the
method will modify
outside the method
because it's the same
object!**



Incorrect Non Copy Method Trace

```
// end of method- local variables and formal  
parameters are garbage collected
```



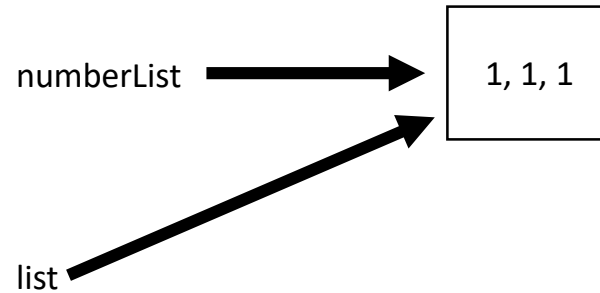
Incorrect Non Copy Method Trace

```
// back in main
```



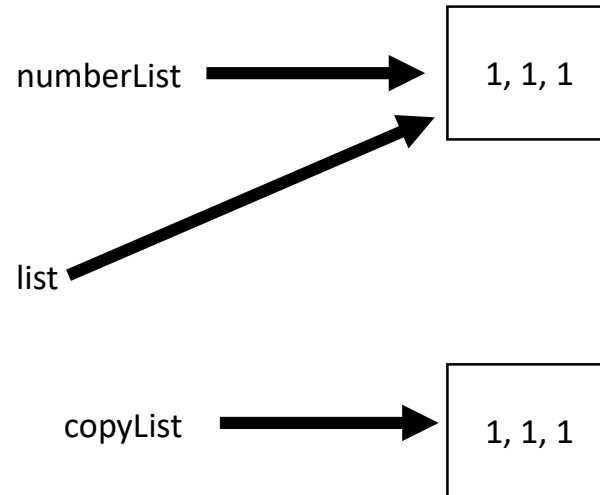
Correct Method Trace

```
correctCopyMethod(numberList) ;
```



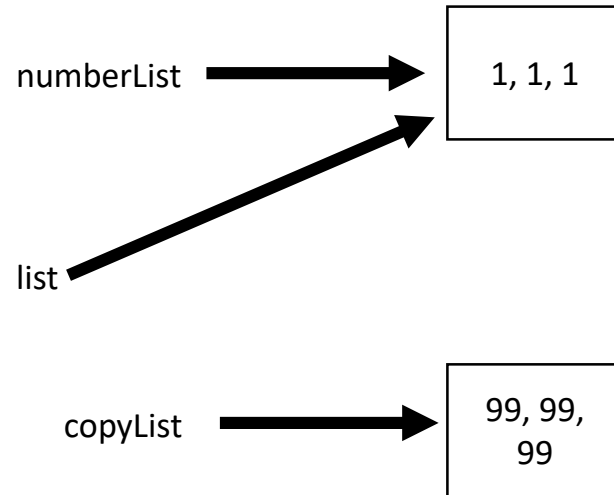
Correct Method Trace

```
List<Integer> copyList = new ArrayList<Integer>();  
// loop to fill
```



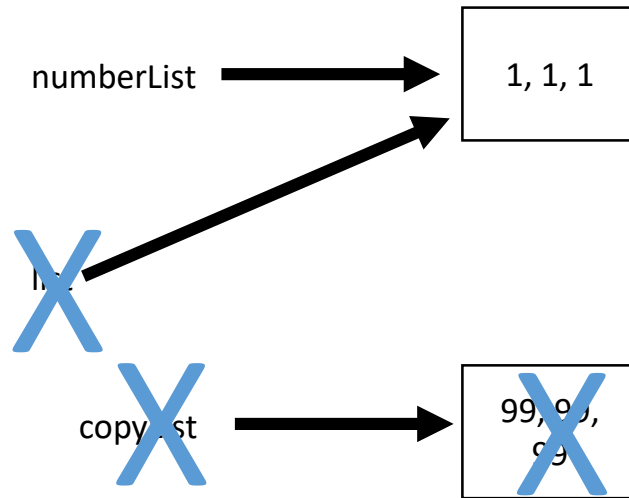
Correct Method Trace

```
// loop to modify
```



Correct Method Trace

// end of method- local variables and formal parameters are garbage collected



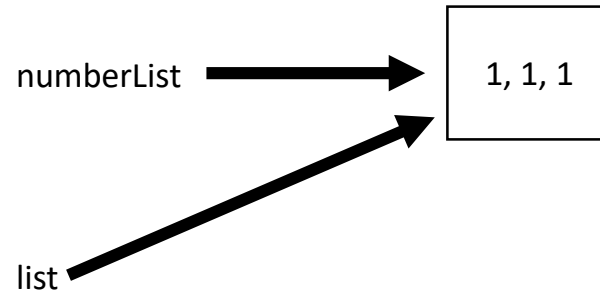
Correct Method Trace

```
// back in main
```



Incorrect Method A Trace

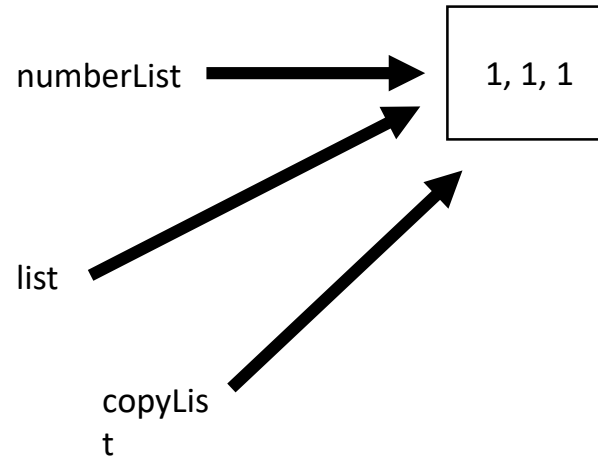
```
incorrectCopyMethodA (numberList) ;
```



Incorrect Method A Trace

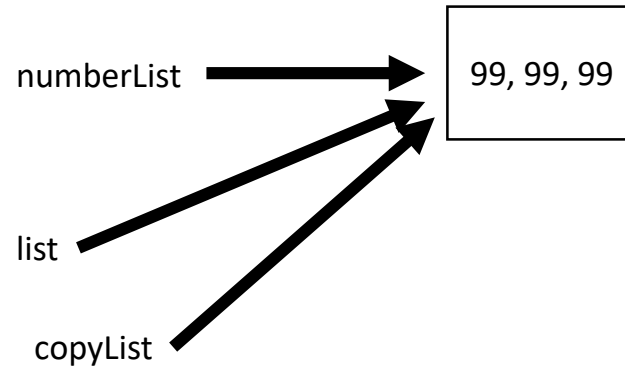
```
List<Integer> copyList = list;
```

This is the error!!!
We haven't created a copy!
We've created an alias!



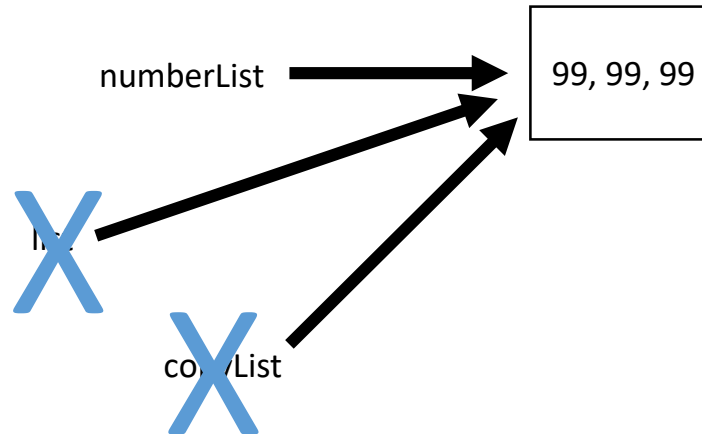
Incorrect Method A Trace

```
// loop to modify
```



Incorrect Method A Trace

```
// end of method- local variables and formal  
parameters are garbage collected
```



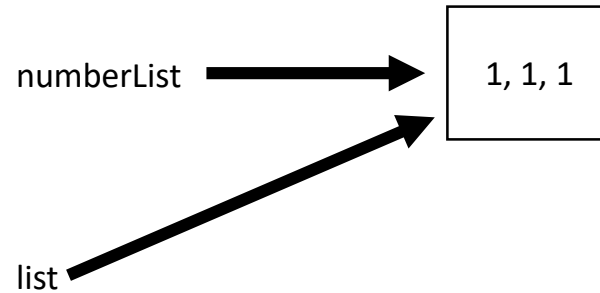
Incorrect Method A Trace

```
// back in main
```



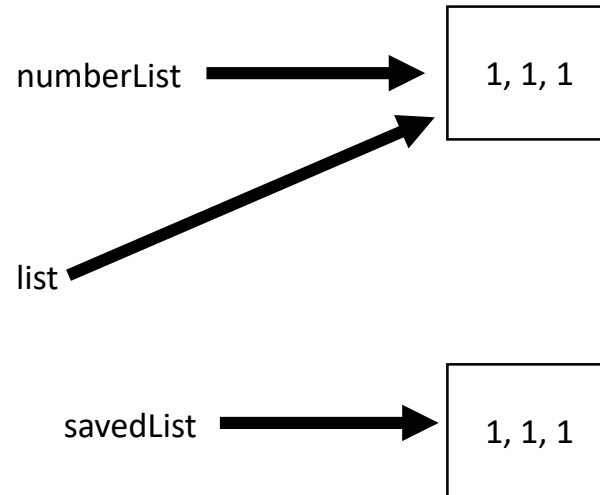
Incorrect Method B Trace

```
incorrectCopyMethodB (numberList) ;
```



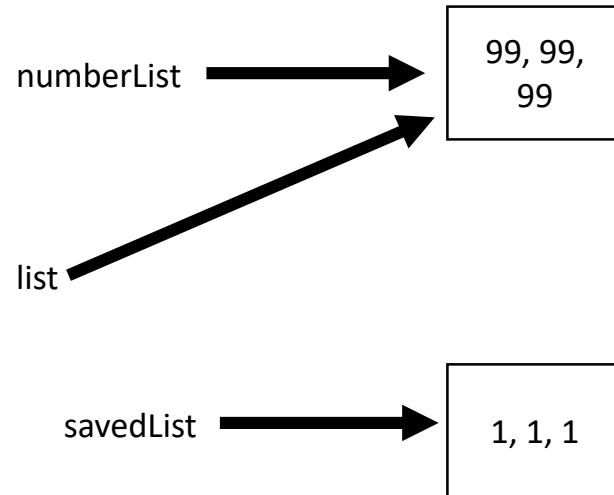
Incorrect Method B Trace

```
List<Integer> savedList = new ArrayList<Integer>();  
// loop to fill
```



Incorrect Method B Trace

```
// loop to modify
```



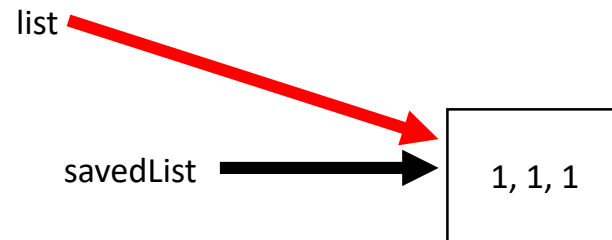
Incorrect Method B Trace

```
list = savedList;
```



This is the error!!!

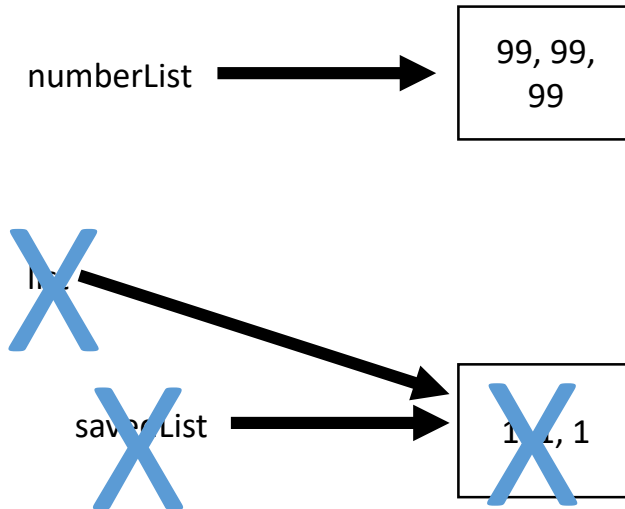
**We assigned a new
value to list,
but not to numberList!**



The alias link is broken.

Incorrect Method B Trace

// end of method- local variables and formal parameters are garbage collected



Incorrect Method B Trace

```
// back in main
```



Key Points

- When objects are passed as parameters, you create an alias.
 - Changes made to the object by invoking methods (with the dot operator) inside the method affect the object outside the method. Because there is only one object!
- Direct assignment with objects creates aliases (**not** new objects).
 - The only way to create a new object is with `new` (or a factory method, clone method, etc.).
- Using direct assignment for a formal parameter breaks the alias link.
 - This is almost always a mistake.

Key Points

- To pass an object into a method and modify it only internally within the method:

create a new copy object (with the new operator)

make the copy object have the same data as the original (might have to be done manually, like with a loop or setters)

modify the new copy object only

THE COMPARABLE INTERFACE

Comparing Objects

- Sorting items is a common thing to do. There are many different ways to sort objects.
- All methods of sorting, however, at some point involve comparing two objects to each other- is object A less than, greater than, or equal to object B?

Comparing Objects

- Implementing the `Comparable` interface allows us to provide a method for how to make this comparison of two objects.
- This is called the *natural ordering* of objects.

The Comparable Interface

- States that two objects can be *compared* or *ordered* with each other and specifies how to make that comparison.
 - The compareTo method defines how the ordering is done.

The Comparable Interface

- Many Java classes implement `compareTo`.
 - `String`, which is why we can call the `compareTo` method on two `Strings`
- Any class we write can implement `Comparable`
 - We get to decide how our objects are ordered, what makes one object "greater than" or "less than" another.

Comparing Objects

- We cannot compare objects with `<` and `>`
 - These operators can only be used for primitives.
 - They define the *natural ordering* for primitives.
- For objects, we must use the `compareTo` method.
 - And we can only do this if an object's class implements the `Comparable` interface!

The Comparable Interface

- The Comparable interface has one abstract method used to compare two objects

```
public int compareTo(Object obj)
```

- You can (and should!) use generics to improve the method:

```
public MyClass implements Comparable<MyClass> {
```

```
    public int compareTo(MyClass obj) { ... }
```

The compareTo Method

- The value returned from compareTo is :
 - **negative** if obj1 is less than obj2
 - **0** if they are equal
 - **positive** if obj1 is greater than obj2

```
if (obj1.compareTo(obj2) < 0) {  
    // obj1 less than obj2  
} else if(obj1.compareTo(obj2) > 0) {  
    // obj1 greater than obj2  
} else {  
    // they are equal  
}
```

The compareTo Method

- It's up to you how to determine what makes one object greater to, less than, or equal to another
 - Example: For an `Employee` class, you could order employees by name (alphabetically), by employee ID number, or by start date
- The implementation of the `compareTo` method can be as straightforward or as complex as needed

The compareTo Method

- Often, you will invoke `compareTo` on objects that are instance data variables from inside your `compareTo` method!
 - Example: using `name.compareTo(otherName)` inside of a `compareTo` method to compare `Employee` objects.
 - This example invokes the `compareTo` method of the `String` class in order to implement the `compareTo` method of the `Employee` class.
- This is similar to the `equals` method.

The compareTo Method

- You might also use the static methods in the wrapper classes to compare primitives:
 - `Integer.compare(num1, num2);`
- Returns the same kind of value:
 - negative if num1 is smaller
 - 0 if they are equal
 - positive if num1 is bigger

The Comparable Interface

- Implementing the Comparable interface allows us to use nice methods from the Java standard class library, such as sorting methods.
 - Collections.sort(myArrayList)
 - Arrays.sort(myArray)
- These methods only works if the class implements Comparable

Comparable and Sorting

- Note that implementing `compareTo` doesn't actually sort anything!
- It only defines *how* to compare two objects to each other.
- This is needed in order to sort. But to actually do the sort, we need another method.

Example

- Implement the Comparable interface in the Student class.
- Sort a collection of Student objects.

Example

- Review the example of how to use Comparable with generics by looking at the Pair class.
 - Order Pairs by the first element, then the second.