References and objects

In Java, objects and arrays use reference semantics. Why?

- **efficiency.** Copying large objects slows down a program.
- **sharing.** It's useful to share an object's data among methods.

```java
DrawingPanel panel1 = new DrawingPanel(80, 50);
DrawingPanel panel2 = panel1;  // same window
panel2.setBackground(Color.CYAN);
```

![Diagram showing references panel1 and panel2]
**References as fields**

- Objects can store references to other objects as fields.
  - `HtmlValidator` stores a reference to a `Queue`
  - the `Queue` stores many references to `HtmlTag` objects
  - each `HtmlTag` object stores a reference to its `String` element

```java
private Queue<HtmlTag> queue;...
```

```java
private String element;
...
```

**Null references**

- `null`: A value that does not refer to any object.
  - Unset reference fields of an object are initialized to `null`

```java
public class Student {
    String name;
    int id;
}
```

```java
Student timmy = new Student();
```

```java

timmy    

<table>
<thead>
<tr>
<th>name</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Things you can do w/ null

• store null in a variable or an array element
  String s = null;
  words[2] = null;

• print a null reference
  System.out.println(timmy.name);  // null

• ask whether a variable or array element is null
  if (timmy.name == null) { ...  // true

• pass null as a parameter to a method
  - some methods don't like null parameters and throw exceptions

• return null from a method (often to indicate failure)
  return null;

Dereferencing

• dereference: To access data or methods of an object.
  - Done with the dot notation, such as s.length()
  - When you use a . after an object variable, Java goes to the memory for that object and looks up the field/method requested.

  Student timmy = new Student();
  timmy.name = "Timmah";
  String s = timmy.name.toUpperCase();
Null pointer exception

- It is illegal to dereference null (it causes an exception).
  - null does not refer to any object, so it has no methods or data.

    Student timmy = new Student();
    String s = timmy.name.toUpperCase();  // ERROR

    timmy →
    \[\begin{array}{|c|}
    \hline
    \text{name} & \text{null} \\
    \text{id} & 0 \\
    \hline
    \end{array}\]

    Output:
    Exception in thread "main"
    java.lang.NullPointerException
    at Example.main(Example.java:8)

References to same type

- What would happen if we had a class that declared one of its own type as a field?

  public class Strange {
    private String name;
    \textbf{private Strange other;}  
  }

  Will this compile?
  - If so, what is the behavior of the other field? What can it do?
  - If not, why not? What is the error and the reasoning behind it?
Linked data structures

• All of the collections we will use and implement in this course use one of the following two underlying data structures:
  - an **array** of all elements
    • ArrayList, Stack, HashSet, HashMap
  - a set of **linked objects**, each storing one element, and one or more reference(s) to other element(s)
    • LinkedList, TreeSet, TreeMap

```
front  42  -3  17  9 null
```

A list node class

```
public class ListNode {
    int data;
    ListNode next;
}
```

• Each list node object stores:
  - one piece of integer data
  - a reference to another list node

• ListNode objects can be "linked" into chains to store a list of values:

```
data next  data next  data next  data next
42      -3      17      9 null
```
List node client example

```java
public class ConstructList1 {
    public static void main(String[] args) {
        ListNode list = new ListNode();
        list.data = 42;
        list.next = new ListNode();
        list.next.data = -3;
        list.next.next = new ListNode();
        list.next.next.data = 17;
        list.next.next.next = null;
        System.out.println(list.data + " " + list.next.data
                           + " " + list.next.next.data);
        // 42 -3 17
    }
}
```

List node w/ constructor

```java
public class ListNode {
    int data;
    ListNode next;

    public ListNode(int data) {
        this.data = data;
        this.next = null;
    }

    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}
```
References vs. objects

\[ \text{variable} = \text{value}; \]

- A variable (left side of \(=\)) is an arrow (the base of an arrow)
- A value (right side of \(=\)) is an object (a box; what an arrow points at)

• For the list at right:
  - \(a.\text{next} = \text{value};\)
    - means to adjust where \(1\) points
  - \(\text{variable} = a.\text{next};\)
    - means to make \(\text{variable}\) point at \(2\)

Reassigning references

• When you say:
  - \(a.\text{next} = b.\text{next};\)

• You are saying:
  - "Make the variable \(a.\text{next}\) refer to the same value as \(b.\text{next}\)."
  - Or, "Make \(a.\text{next}\) point to the same place that \(b.\text{next}\) points."
Linked Node Question

• Suppose we have a long chain of list nodes:

- We don't know exactly how long the chain is.

• How would we print the data values in all the nodes?
**Algorithm pseudocode**

- Start at the **front** of the list.
- While (there are more nodes to print):
  - Print the current node's data.
  - Go to the **next** node.

**How do we walk through the nodes of the list?**

```
list = list.next;    // is this a good idea?
```

**Traversing a list?**

- One (bad) way to print every value in the list:

  ```java
  while (list != null) {
      System.out.println(list.data);
      list = list.next;   // move to next node
  }
  ```

  - What's wrong with this approach?
A current reference

• Don't change list. Make another variable, and change that.
  - A ListNode variable is NOT a ListNode object

```java
ListNode current = list;
```

- A ListNode variable is NOT a ListNode object

• What happens to the picture above when we write:

```java
current = current.next;
```

Traversing a list correctly

• The correct way to print every value in the list:

```java
ListNode current = list;
while (current != null) {
    System.out.println(current.data);
    current = current.next; // move to next node
}
```

- Changing current does not damage the list.
**Linked list vs. array**

- **Algorithm to print list values:**
  
  ```java
  ListNode front = ...;
  ListNode current = front;
  while (current != null) {
      System.out.println(current.data);
      current = current.next;
  }
  ```

- **Similar to array code:**
  
  ```java
  int[] a = ...;
  int i = 0;
  while (i < a.length) {
      System.out.println(a[i]);
      i++;
  }
  ```

**A LinkedIntList class**

- **Let's write a collection class named** `LinkedIntList`.
  
  - **Has the same methods as** `ArrayIntList`:
    
    - `add, add, get, indexOf, remove, size, toString`

  - **The list is internally implemented as a chain of linked nodes**
    
    - The `LinkedIntList` keeps a reference to its `front` as a field
    - **null is the end of the list; a null front signifies an empty list**
public class LinkedIntList {
    private ListNode front;
    private int size;

    public LinkedIntList() {
        front = null;
        size = 0;
    }

    methods go here
}

Implementing add

// Adds the given value to the end of the list.
public void add(int value) {
    ...
}

- How do we add a new node to the end of a list?
- Does it matter what the list's contents are before the add?
Adding to an empty list

• Before adding 20:

  front = | |  size = 0

After:

  front = | |  size = 1
  data    next
  20       element 0

- We must create a new node and attach it as the front of the list.

The add method, 1st try

// Adds the given value to the end of the list.
public void add(int value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list

        ...

    }
}


Adding to non-empty list

• Before adding value 20 to end of list:

  front =  
  size = 2

  data  data
  42  -3

  element 0  element 1

• After:

  front =  
  size = 3

  data  data  data
  42  -3  20

  element 0  element 1  element 2

Don't fall off the edge!

• To add/remove from a list, you must modify the next reference of the node before the place you want to change.

  front =  
  size = 2

  data  data
  42  -3

  element 0  element 1

  - Where should current be pointing, to add 20 at the end?
  - What loop test will stop us at this place in the list?
The `add` method

```java
// Adds the given value to the end of the list.
public void add(int value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ListNode current = front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new ListNode(value);
    }
    size++;
}
```

Implementing `add` (2)

```java
// Inserts the given value at the given index.
public void add(int index, int value) {
    ...
}
```
Conceptual questions

• What is the difference between a `LinkedList` and a `ListNode`?

• What is the difference between an empty list and a null list?
  - How do you create each one?

• Why are the fields of `ListNode` public? Is this bad style?

• What effect does this code have on a `LinkedList`?

  ```java
  ListNode current = front;
  current = null;
  ```

Implementing `remove`

// Removes value at given index from list.
public void remove(int index) {
    ...
}

- How do we remove any node from a list?
- Does it matter what the list's contents are before the remove?
Removing from a list

• Before removing element at index 1:

Before:
- front = 42
- size = 3
- next:
  - data: 42
  - next:
    - data: -3
    - next:
      - data: 20

After:
- front = 42
- size = 2
- next:
  - data: 20

Removing from the front

• Before removing element at index 0:

Before:
- front = 42
- size = 3
- next:
  - data: 42
  - next:
    - data: -3
    - next:
      - data: 20

After:
- front = -3
- size = 2
- next:
  - data: 20
Removing the only element

• Before: After:

- We must change the front field to store `null` instead of a node.
- Do we need a special case to handle this?

```java
// Removes value at given index from list.
// Precondition: 0 <= index < size()
public void remove(int index) {
    if (index == 0) {
        // special case: removing first element
        front = front.next;
    } else {
        // removing from elsewhere in the list
        ListNode current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current.next;
        }
        current.next = current.next.next;
    }
    size--;  
}
```
Exercise

• Write pseudocode for `addSorted` that accepts an integer value as a parameter and adds that value to a sorted list in sorted order.
  - Before `addSorted(17)`:
    - `front = size = 3`
    - `data next data next data next`
      - element 0
      - element 1
      - element 2
    - `data next`
      - `-4`
      - `8`
      - `22`
  - After `addSorted(17)`:
    - `front = size = 4`
    - `data next data next data next data next`
      - element 0
      - element 1
      - element 2
      - element 3
    - `data next`
      - `-4`
      - `8`
      - `17`
      - `22`

The common case

• Adding to the middle of a list:
  `addSorted(17)`
  - Which references must be changed?
  - What sort of loop do we need?
  - When should the loop stop?
First attempt

• An incorrect loop:
   ```java
   ListNode current = front;
   while (current.data < value) {
       current = current.next;
   }
   ```

• What is wrong with this code?
  - The loop stops too late to affect the list in the right way.

Key idea: peeking ahead

• Corrected version of the loop:
  ```java
  ListNode current = front;
  while (current.next.data < value) {
      current = current.next;
  }
  ```

  - This time the loop stops in the right place.
Another case to handle

• Adding to the end of a list:
  addSorted(42)

Exception in thread "main": java.lang.NullPointerException

  - Why does our code crash?
  - What can we change to fix this case?

Third case to handle

• Adding to the front of a list:
  addSorted(-10)

  - What will our code do in this case?
  - What can we change to fix it?
Handling the front

• Another correction to our code:

```java
if (value <= front.data) {
    // insert at front of list
    front = new ListNode(value, front);
} else {
    // insert in middle of list
    ListNode current = front;
    while (current.next != null &&
        current.next.data < value) {
        current = current.next;
    }
}
```

– Does our code now handle every possible case?

Fourth case to handle

• Adding to (the front of) an empty list:

```
addSorted(42)
```

front = [ ]

– What will our code do in this case?
– What can we change to fix it?
Final version of code

```java
// Adds given value to list in sorted order.
// Precondition: Existing elements are sorted
public void addSorted(int value) {
    if (front == null || value <= front.data) {
        // insert at front of list
        front = new ListNode(value, front);
    } else {
        // insert in middle of list
        ListNode current = front;
        while (current.next != null &&
            current.next.data < value) {
            current = current.next;
        }
    }
}
```

Linked vs. array lists

- We have implemented the following two collection classes:
  - ArrayIntList
    ```
    index 0 1 2 3
    value 42 -3 17 9
    ```
  - LinkedIntList
    ```
    data 42 next
    data -3
    data 17 next
    data 9 next
    ```
- They have similar behavior.
  We should be able to treat them the same way in client code.
An IntList interface

// Represents a list of integers.
public interface IntList {
    public void add(int value);
    public void add(int index, int value);
    public int get(int index);
    public int indexOf(int value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, int value);
    public int size();
}

public class ArrayIntList implements IntList { ... public class LinkedIntList implements IntList { ...