Building Java Programs

Chapter 16

References and linked nodes

reading: 16.1

Value semantics

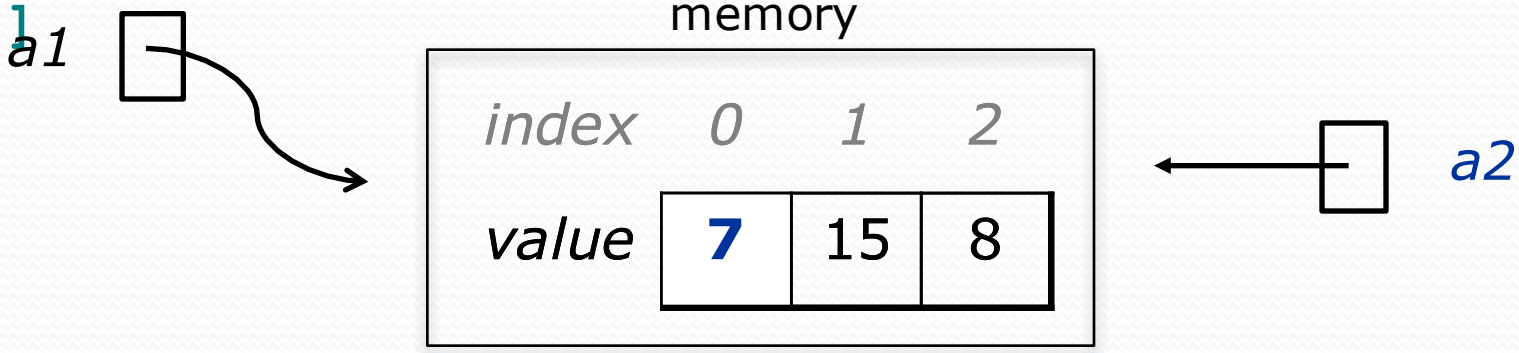
- **value semantics**: Behavior where values are copied when assigned, passed as parameters, or returned.
  - All primitive types in Java use value semantics.
  - When one variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```
int x = 5;
int y = x;      // x = 5, y = 5
y = 17;         // x = 5, y = 17
x = 8;          // x = 8, y = 17
```

Reference semantics (objects)

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
  - Modifying the value of one variable *will* affect others.

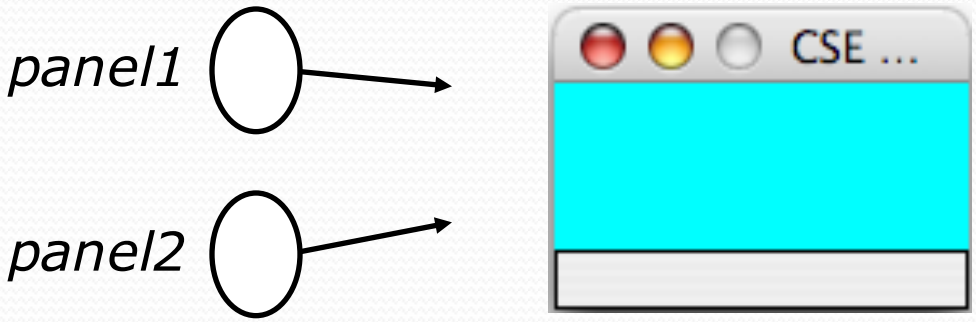
```
int[] a1 = {4, 15, 8};
int[] a2 = a1;              // refer to same array as
a1
a2[0] = 7;
System.out.println(Arrays.toString(a1)); // [7, 15,
8]
```



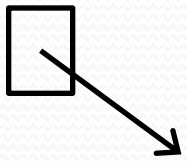
References and objects

- Arrays and objects use reference semantics. Why?
  - *efficiency.* Copying large objects slows down a program.
  - *sharing.* It's useful to share an object's data among methods.

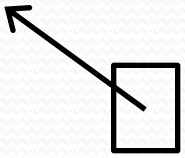
```
DrawingPanel panel1 = new DrawingPanel(80, 50);
DrawingPanel panel2 = panel1;   // same window
panel2.setBackground(Color.CYAN);
```



cats1

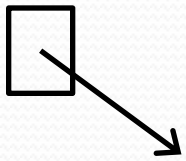


```
cat[] cats1 = {🐱,🐱,🐱,🐱};
cat[] cats2 = cats1;
```

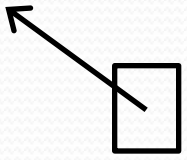


cats2

dogs1



```
dog[] dogs1 = {🐶,🐶,🐶};
dog[] dogs2 = dogs1;
```



dogs2

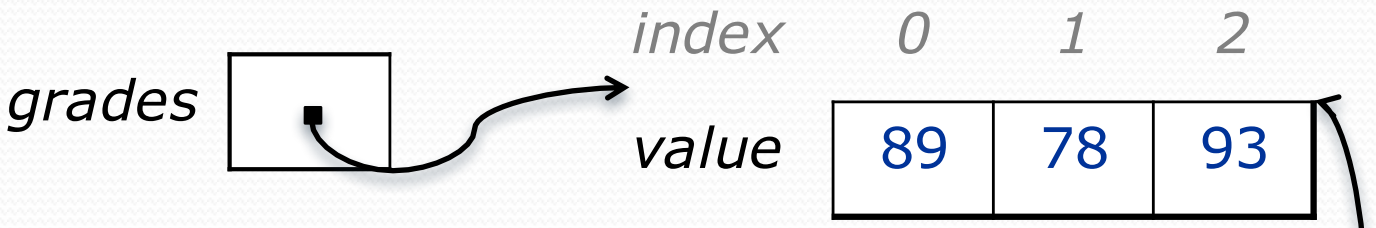
Value/Reference Semantics

- Variables of primitive types store values directly:

- Values are copied from one variable to another:

```
cats = age;
```

- Variables of object types store references to memory:



- References are copied from one variable to another:

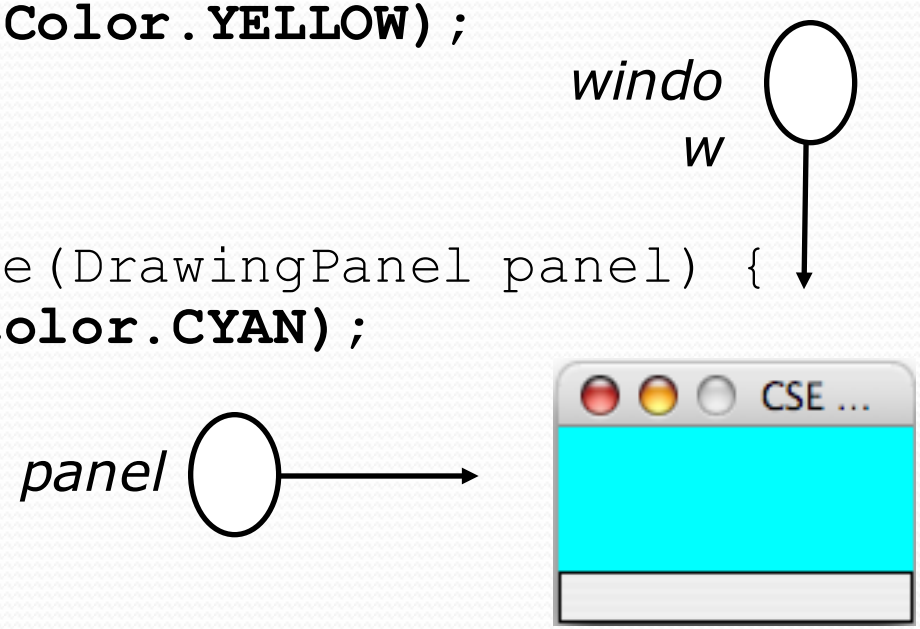
```
scores = grades;
```

Objects as parameters

- When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.
  - If the parameter is modified, it *will* affect the original object.

```
public static void main(String[] args) {
    DrawingPanel window = new DrawingPanel(80, 50);
    window.setBackground(Color.YELLOW);
    example(window);
}

public static void example(DrawingPanel panel) {
    panel.setBackground(Color.CYAN);
    ...
}
```

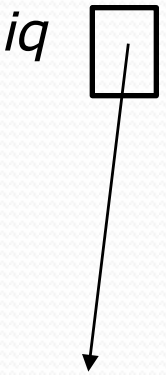


Arrays pass by reference

- Arrays are passed as parameters by *reference.*
  - Changes made in the method are also seen by the caller.

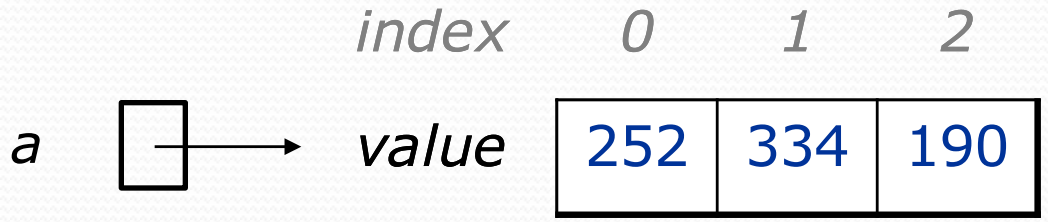
```
public static void main(String[] args) {
    int[] iq = {126, 167, 95};
    increase(iq);
    System.out.println(Arrays.toString(iq));
}

public static void increase(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}
```



  - Output:

```
[252, 334, 190]
```

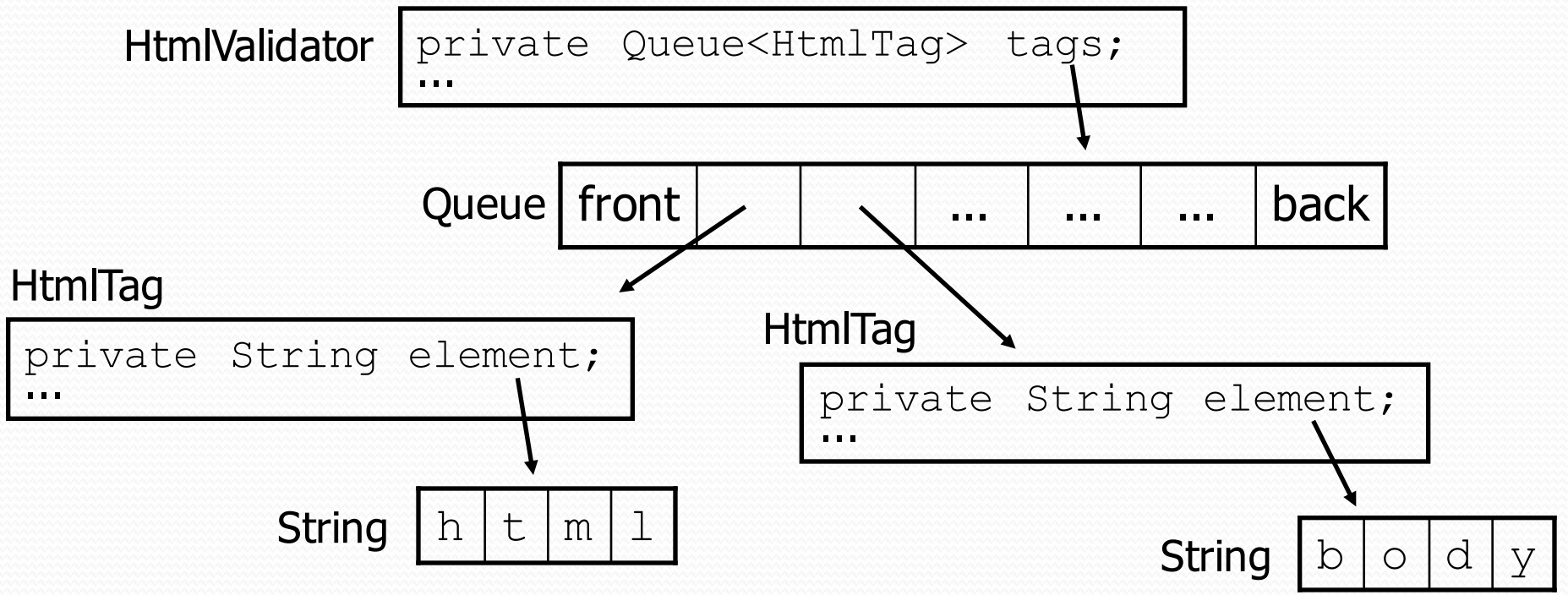


References as fields

- Objects can store references to other objects as fields.

  Example: Homework 2 (HTML Validator)

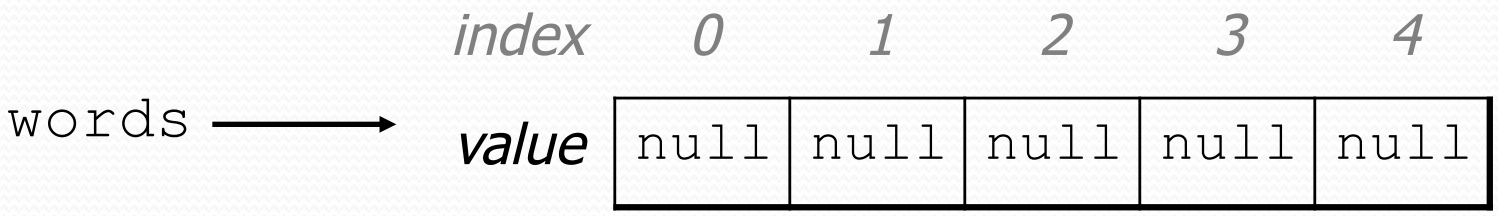
  - `HtmlValidator` stores a reference to a `Queue`
  - the `Queue` stores many references to `HtmlTag` objects
  - each `HtmlTag` object stores a reference to its element `String`



Null references

- **`null` :** A value that does not refer to any object.
  - The elements of an array of objects are initialized to `null`.

```
String[] words = new String[5];
```



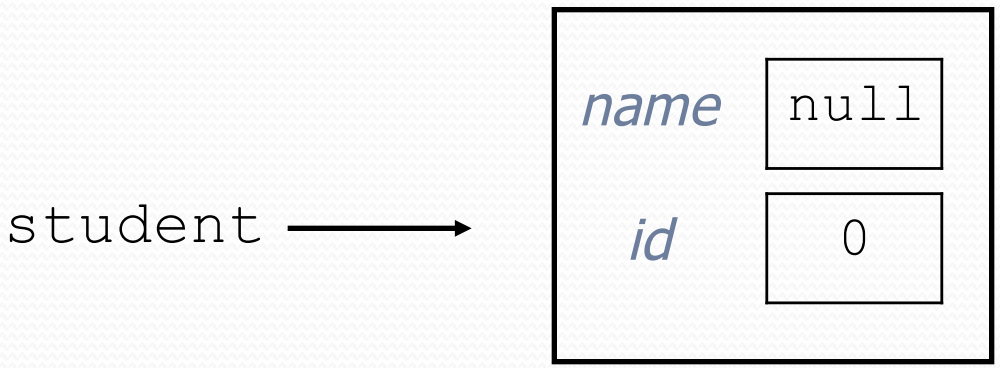
  - not the same as the empty string `""` or the string `"null"`
  - Why does Java have `null` ? What is it used for?

Null references

- Unset reference fields of an object are initialized to `null`.

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

Student student = new Student();
```



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(student.name);      // null
```

- ask whether a variable or array element is `null`

```
if (student.name == null) { ...        // true
```

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

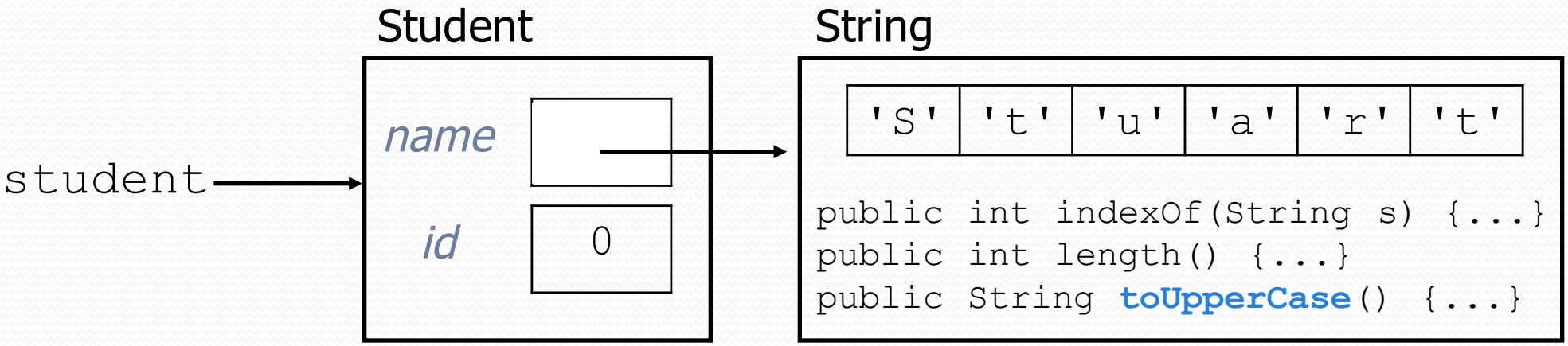
- return `null` from a method (often to indicate failure)

```
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.

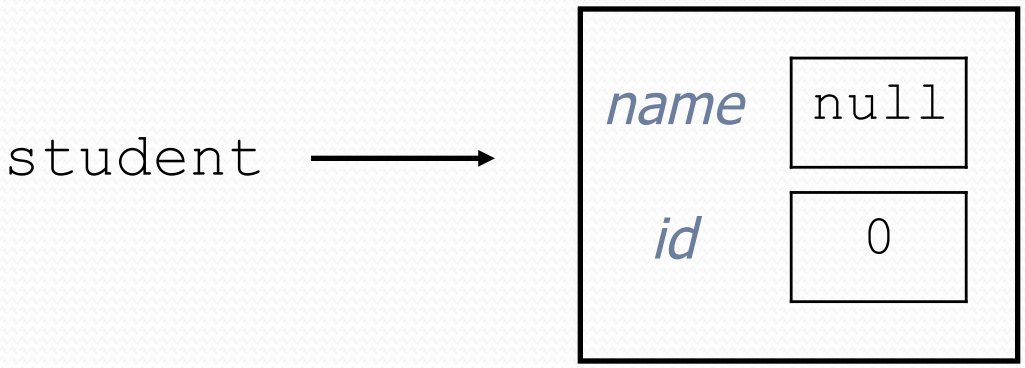
```
Student student = new Student();
student.name = "Stuart";
String s = student.name.toUpperCase();
```



Null pointer exception

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

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

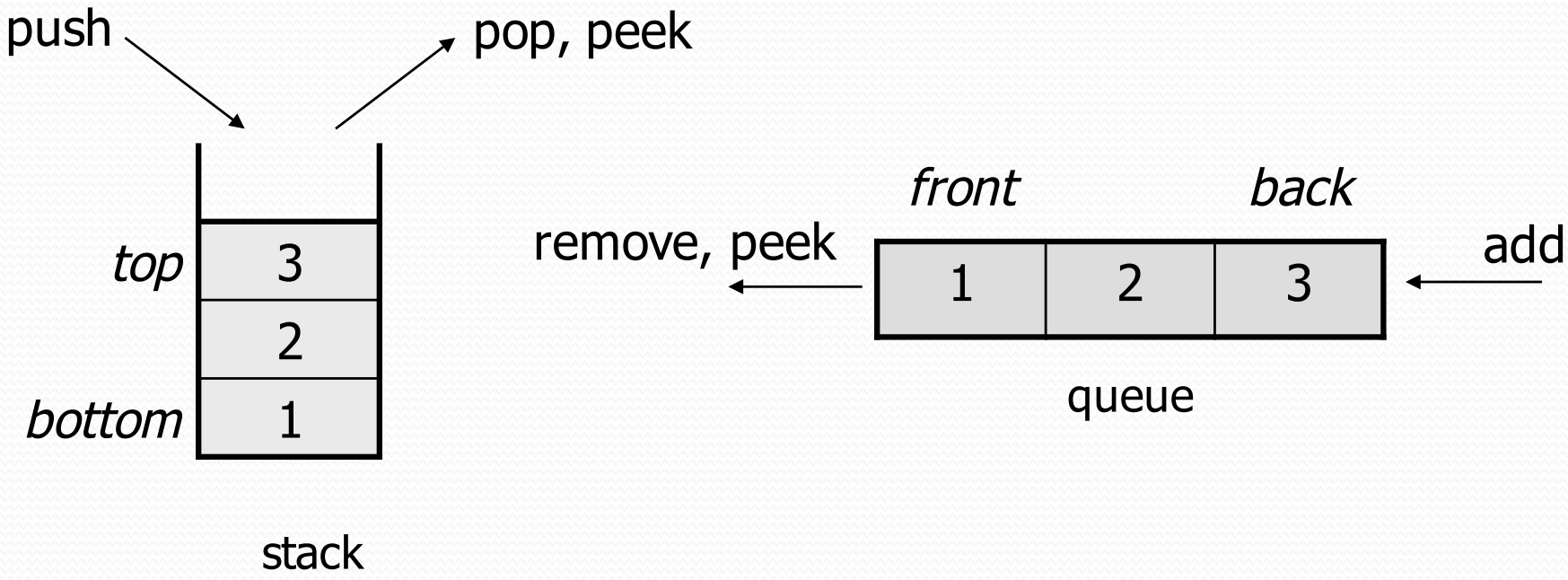


Output:

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

Recall: stacks and queues

- **stack**: retrieves elements in reverse order as added
- **queue**: retrieves elements in same order as added



Collection efficiency

- Complexity class of various operations on collections:

Method	ArrayList	Stack	Queue
`add` (or `push`)	O(1)	O(1)	O(1)
`add(`**`index, value`**`)`	O(N)	-	-
`indexOf`	O(N)	-	-
`get`	O(1)	-	-
`remove`	O(N)	O(1)	O(1)
`set`	O(1)	-	-
`size`	O(1)	O(1)	O(1)

- Could we build lists differently to optimize other operations?

Array vs. linked structure

- All collections in this course use one of the following:
  - an **array** of all elements
    - examples: `ArrayList`, `Stack`, `HashSet`, `HashMap`

  - **linked objects** storing a value and references to other(s)
    - examples: `LinkedList`, `TreeSet`, `TreeMap`

    - First, we will learn how to create a *linked list*.
    - To understand linked lists, we must understand *references*.

Memory for a List

- Array (contiguous in memory)

- Spread in memory


{array n., a group of hedgehogs}

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;
    private Strange other;
}
```

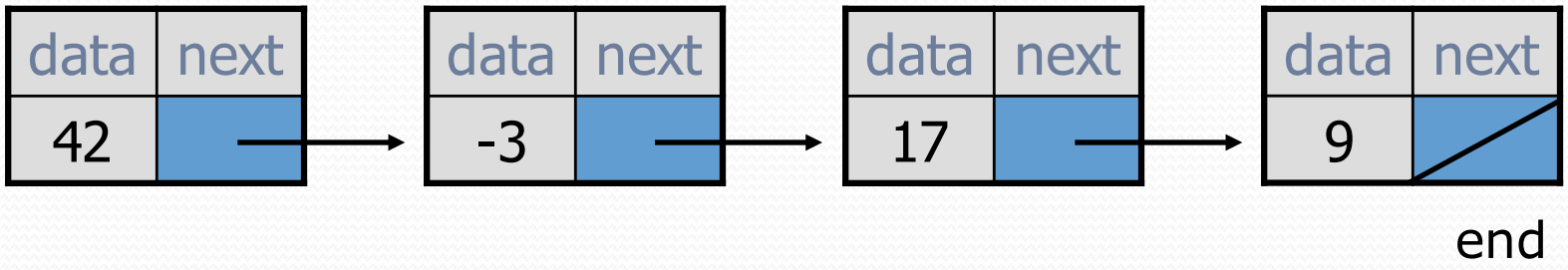
  - Will this compile?
    - If so, what is the behavior of the `other` field? What can it do?
    - If not, why not? What is the error and the reasoning behind it?

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`s can be "linked" into chains to store a list of values:



Arrays vs. linked lists

- Array advantages
  - Random access: can quickly retrieve any value
- Array disadvantages
  - Adding/removing in middle is O(n)
  - Expanding requires creating a new array and copying elements
- Linked list advantages
  - Adding/removing in middle is O(1)
  - Expanding is O(1) (just add a node)
- Linked list disadvantages
  - Sequential access: can't directly retrieve any value