



## Lecture 3

# Primitive Types, Reference Types, and Linked Data Structures

CS61B, Spring 2024 @ UC Berkeley

Slides credit: Josh Hug

# Goals: Building a List

---

Lecture 3, CS61B, Spring 2024

## Goals: Building a List

Primitive Types

Reference Types

Parameter Passing

Instantiation of Arrays

IntList and Linked Data Structures

- Unlike Python, lists are not built directly into the Java language.

```
import java.util.List;
import java.util.LinkedList;
List<String> L = new LinkedList<>();
L.add("a");
L.add("b");
```

Today, we'll begin our 3 lecture journey towards building our own list implementation.

- We'll exploit recursion to allow our list to grow infinitely large.
- But first we need to solve... the mystery of the walrus.

# Primitive Types

---

Lecture 3, CS61B, Spring 2024

Goals: Building a List

## **Primitive Types**

Reference Types

Parameter Passing

Instantiation of Arrays

IntList and Linked Data Structures

## Quiz

```
Walrus a = new Walrus(1000, 8.3);  
Walrus b;  
b = a;  
b.weight = 5;  
System.out.println(a);  
System.out.println(b);
```

Will the change to b affect a?

- A. Yes
- B. No

```
weight: 5, tusk size: 8.30  
weight: 5, tusk size: 8.30
```

Answer: [Visualizer](#)

```
int x = 5;  
int y;  
y = x;  
x = 2;  
System.out.println("x is: " + x);  
System.out.println("y is: " + y);
```

Will the change to x affect y?

- A. Yes
- B. No

```
x is: 2  
y is: 5
```

Your computer stores information in “memory”.

- Information is stored in memory as a sequence of ones and zeros.
  - Example: 72 stored as 01001000
  - Example: 205.75 stored as ... 01000011 01001101 11000000 00000000
  - Example: The letter H stored as 01001000 (same as the number 72)
  - Example: True stored as 00000001

Each Java type has a different way to interpret the bits:

- 8 primitive types in Java: byte, short, **int**, long, float, **double**, boolean, char
- We won't discuss the precise representations in much detail in 61B.
  - Covered in much more detail in 61C.

Note: Precise representations may vary from machine to machine.

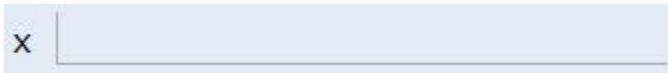
## Declaring a Variable (Simplified)

---

When you declare a variable of a certain type in Java:

- Your computer sets aside exactly enough bits to hold a thing of that type.
  - Example: Declaring an int sets aside a “box” of 32 bits.
  - Example: Declaring a double sets aside a box of 64 bits.
- Java creates an internal table that maps each variable name to a location.
- Java does NOT write anything into the reserved boxes.
  - For safety, Java will not let you access a variable that is uninitialized.

```
int x;  
double y;  
x = -1431195969;  
y = 567213.112;
```



## Declaring a Variable (Simplified)

When you declare a variable of a certain type in Java:

- Your computer sets aside exactly enough bits to hold a thing of that type.
  - Example: Declaring an int sets aside a “box” of 32 bits.
  - Example: Declaring a double sets aside a box of 64 bits.
- Java creates an internal table that maps each variable name to a location.
- Java does NOT write anything into the reserved boxes.
  - For safety, Java will not let you access a variable that is uninitialized.

```
int x;  
double y;  
x = -1431195969;  
y = 567213.112;
```

x

y



## Declaring a Variable (Simplified)

When you declare a variable of a certain type in Java:

- Your computer sets aside exactly enough bits to hold a thing of that type.
  - Example: Declaring an int sets aside a “box” of 32 bits.
  - Example: Declaring a double sets aside a box of 64 bits.
- Java creates an internal table that maps each variable name to a location.
- Java does NOT write anything into the reserved boxes.
  - For safety, Java will not let you access a variable that is uninitialized.

```
int x;  
double y;  
x = -1431195969;  
y = 567213.112;
```

x 10101010101100011010111010111111

y

## Declaring a Variable (Simplified)

When you declare a variable of a certain type in Java:

- Your computer sets aside exactly enough bits to hold a thing of that type.
  - Example: Declaring an int sets aside a “box” of 32 bits.
  - Example: Declaring a double sets aside a box of 64 bits.
- Java creates an internal table that maps each variable name to a location.
- Java does NOT write anything into the reserved boxes.
  - For safety, Java will not let you access a variable that is uninitialized.

```
int x;  
double y;  
x = -1431195969;  
y = 567213.112;
```

x 10101010101100011010111010111111

y 0100000100100001010011110101101000111001010110000001000001100010

## Simplified Box Notation

---

We'll use simplified box notation from here on out:

- Instead of writing memory box contents in binary, we'll write them in human readable symbols.

```
int x;  
double y;  
x = -1431195969;  
y = 567213.112;
```

x	-1431195969
---	-------------

y	567213.112
---	------------

## The Golden Rule of Equals (GRoE)

---

Given variables  $y$  and  $x$ :

- $y = x$  **copies** all the bits from  $x$  into  $y$ .

Example from earlier: [Link](#)

# Reference Types

---

Lecture 3, CS61B, Spring 2024

Goals: Building a List

Primitive Types

**Reference Types**

Parameter Passing

Instantiation of Arrays

IntList and Linked Data Structures

## Reference Types

---

There are 8 primitive types in Java:

- byte, short, **int**, long, float, **double**, boolean, char

Everything else, including arrays, is a **reference type**.

## Class Instantiations

When we instantiate an Object (e.g. Dog, Walrus, Planet):

- Java first allocates a box of bits for each instance variable of the class and fills them with a default value (e.g. 0, null).
- The constructor then usually fills every such box with some other value.

```
public static class Walrus {  
    public int weight;  
    public double tuskSize;  
  
    public Walrus(int w, double ts) {  
        weight = w;  
        tuskSize = ts;  
    }  
}
```

→ `new Walrus(1000, 8.3);`

[Demo Link](#)

Walrus instance

32 bits	{	weight	1000
64 bits	{	tuskSize	8.3

## Class Instantiations

## When we instantiate an Object (e.g. Dog, Walrus, Planet):

- Java first allocates a box of bits for each instance variable of the class and fills them with a default value (e.g. 0, null).
- The constructor then usually fills every such box with some other value.

[illegible]

```
→ new Walrus(1000, 8.3);
```



Green is `weight`, blue is `tuskSize`.

(In reality, total Walrus size is slightly larger than 96 bits.)



## Class Instantiations

Can think of `new` as returning the address of the newly created object.

- Addresses in Java are 64 bits.
- Example (rough picture): If object is created in memory location 2384723423, then new returns 2384723423.

2384723423<sup>th</sup> bit

[illegible]

2384723423

```
new Walrus(1000, 8.3);
```

### Walrus instance

32 bits {	weight	1000
64 bits {	tuskSize	8.3

## When we declare a variable of any reference type (Walrus, Dog, Planet):

- Java allocates exactly a box of size 64 bits, no matter what type of object.
- These bits can be either set to:
  - Null (all zeros).
  - The 64 bit “address” of a specific instance of that class (returned by **new**).

```
Walrus someWalrus;  
someWalrus = null;
```

64 bits

[illegible]

```
Walrus someWalrus;  
someWalrus = new Walrus(1000, 8.3);
```

64 bits

someWalrus	0100011000011100001001111100000100011101110111000001111000111111
------------	--

96 bits

### Walrus instance

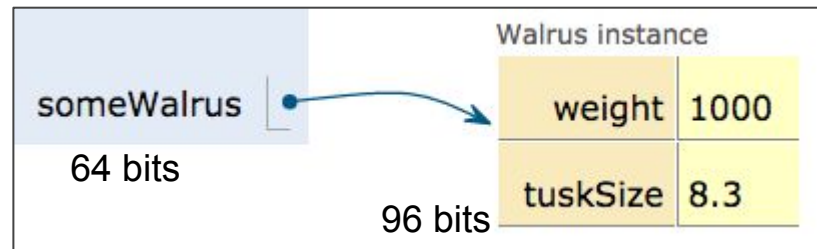
weight	1000
tuskSize	8.3

## Reference Type Variable Declarations

The 64 bit addresses are meaningless to us as humans, so we'll represent:

- All zero addresses with “null”.
- Non-zero addresses as arrows.

This is sometimes called “box and pointer” notation.

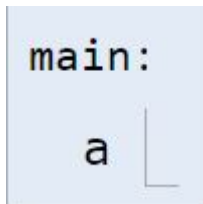


## Reference Types Obey the Golden Rule of Equals

Just as with primitive types, the equals sign copies the bits.

- In terms of our visual metaphor, we “copy” the arrow by making the arrow in the b box point at the same instance as a.

```
→ Walrus a;  
a = new Walrus(1000, 8.3);  
Walrus b;  
b = a;
```



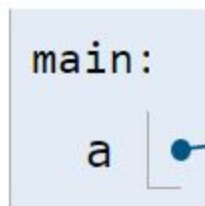
a is 64 bits

## Reference Types Obey the Golden Rule of Equals

Just as with primitive types, the equals sign copies the bits.

- In terms of our visual metaphor, we “copy” the arrow by making the arrow in the b box point at the same instance as a.

```
→ Walrus a;  
a = new Walrus(1000, 8.3);  
Walrus b;  
b = a;
```



a is 64 bits

The Walrus shown is 96 bits.

Walrus instance

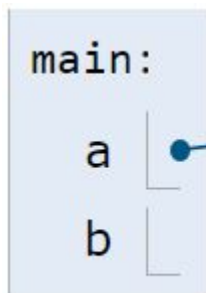
weight	1000
tuskSize	8.3

## Reference Types Obey the Golden Rule of Equals

Just as with primitive types, the equals sign copies the bits.

- In terms of our visual metaphor, we “copy” the arrow by making the arrow in the b box point at the same instance as a.

```
Walrus a;  
a = new Walrus(1000, 8.3);  
→ Walrus b;  
b = a;
```



a and b are 64 bits

The Walrus shown is 96 bits.  
Walrus instance

weight	1000
tuskSize	8.3

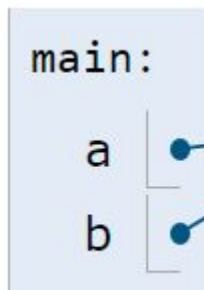
Note: b is currently  
undefined, not null!

## Reference Types Obey the Golden Rule of Equals

Just as with primitive types, the equals sign copies the bits.

- In terms of our visual metaphor, we “copy” the arrow by making the arrow in the b box point at the same instance as a.

```
Walrus a;  
a = new Walrus(1000, 8.3);  
Walrus b;  
→ b = a;
```



The Walrus shown is 96 bits.  
Walrus instance

weight	1000
tuskSize	8.3

a and b are 64 bits

# Parameter Passing

---

Lecture 3, CS61B, Spring 2024

Goals: Building a List

Primitive Types

Reference Types

**Parameter Passing**

Instantiation of Arrays

IntList and Linked Data Structures



## The Golden Rule of Equals (and Parameter Passing)

Given variables b and a:

- `b = a` **copies** all the bits from a into b.

Passing parameters obeys the same rule: Simply **copy the bits** to the new scope.

```
public static double average(double a, double b) {  
    return (a + b) / 2;  
}
```

```
public static void main(String[] args) {  
    → double x = 5.5;  
    double y = 10.5;  
    double avg = average(x, y);  
}
```

main	
x	5.5

## The Golden Rule of Equals (and Parameter Passing)

Given variables b and a:

- `b = a` **copies** all the bits from a into b.

Passing parameters obeys the same rule: Simply **copy the bits** to the new scope.

```
public static double average(double a, double b) {  
    return (a + b) / 2;  
}
```

```
public static void main(String[] args) {  
    → double x = 5.5;  
    double y = 10.5;  
    double avg = average(x, y);  
}
```

main

x	5.5
---	-----

## The Golden Rule of Equals (and Parameter Passing)

Given variables b and a:

- `b = a` **copies** all the bits from a into b.

Passing parameters obeys the same rule: Simply **copy the bits** to the new scope.

```
public static double average(double a, double b) {  
    return (a + b) / 2;  
}
```

```
public static void main(String[] args) {  
    double x = 5.5;  
    → double y = 10.5;  
    double avg = average(x, y);  
}
```

main	
x	5.5
y	10.5

## The Golden Rule of Equals (and Parameter Passing)

Given variables b and a:

- `b = a` **copies** all the bits from a into b.

Passing parameters obeys the same rule: Simply **copy the bits** to the new scope.

```
public static double average(double a, double b) {  
    return (a + b) / 2;  
}
```

```
public static void main(String[] args) {  
    double x = 5.5;  
    double y = 10.5;  
    → double avg = average(x, y);  
}
```

main	
x	5.5
y	10.5

## The Golden Rule of Equals (and Parameter Passing)

Given variables b and a:

- `b = a` **copies** all the bits from a into b.

This is also called pass by value.

Passing parameters obeys the same rule: Simply **copy the bits** to the new scope.

```
public static double average(double a, double b) {  
    return (a + b) / 2;  
}
```

```
public static void main(String[] args) {  
    double x = 5.5;  
    double y = 10.5;  
    double avg = average(x, y);  
}
```

average	
a	5.5
b	10.5

main	
x	5.5
y	10.5

## The Golden Rule: Summary

There are 9 types of variables in Java:

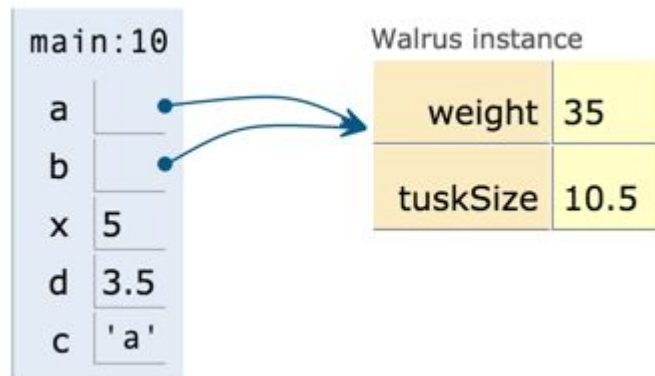
- 8 primitive types (byte, short, int, long, float, double, boolean, char).
- The 9th type is references to Objects (an arrow). References may be null.

In box-and-pointer notation, each variable is drawn as a labeled box and values are shown in the box.

- Addresses are represented by arrows to object instances.

The golden rule:

- `b = a` **copies the bits** from a into b.
- Passing parameters **copies the bits**.



Does the call to `doStuff(walrus, x)` have an affect on `walrus` and/or `main's x`?

- A. Neither will change.
- B. `walrus` will lose 100 lbs, but `main's x` will not change.
- C. `walrus` will not change, but `main's x` will decrease by 5.
- D. Both will decrease.

Answer: <http://goo.gl/ngsxkq>

```
public static void main(String[] args) {  
    Walrus walrus = new Walrus(3500, 10.5);  
    int x = 9;  
    doStuff(walrus, x);  
    System.out.println(walrus);  
    System.out.println(x);  
}  
  
public static void doStuff(Walrus W, int x) {  
    W.weight = W.weight - 100;  
    x = x - 5;  
}
```

# Instantiation of Arrays

---

Lecture 3, CS61B, Spring 2024

Goals: Building a List

Primitive Types

Reference Types

Parameter Passing

**Instantiation of Arrays**

IntList and Linked Data Structures



# Declaration and Instantiation of Arrays

Arrays are also Objects. As we've seen, objects are (usually) instantiated using the **new** keyword.

- `Planet p = new Planet(0, 0, 0, 0, 0, "blah.png");`
- `int[] x = new int[]{0, 1, 2, 95, 4};`

`int[] a;` Declaration

- Declaration creates a 64 bit box intended only for storing a reference to an int array. **No object is instantiated.**



`new int[]{0, 1, 2, 95, 4};`

Instantiation (HW0 covers this syntax)

- Instantiates a new Object, in this case an int array.
- Object is anonymous!

array

0	1	2	3	4
0	1	2	95	4

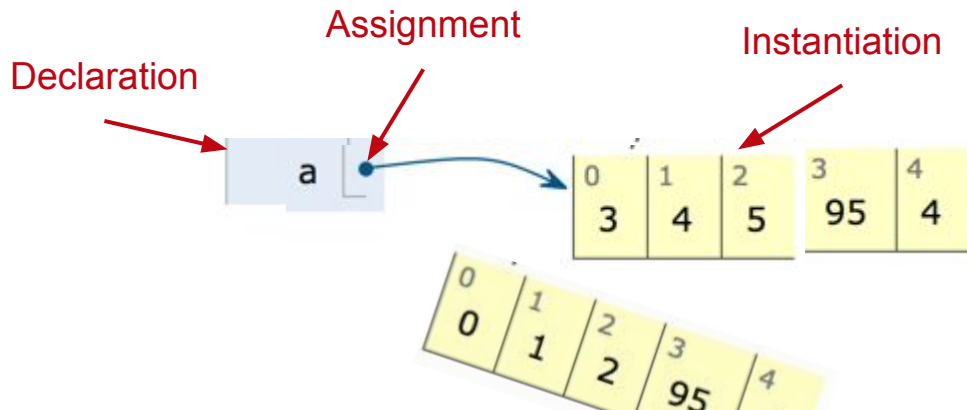
# Assignment of Arrays

`int[] a = new int[]{0, 1, 2, 95, 4};` ← Declaration, instantiation, and assignment.

- Creates a 64 bit box for storing an int array address. (declaration)
- Creates a new Object, in this case an int array. (instantiation)
- Puts the address of this new Object into the 64 bit box named a. (assignment)

Note: Instantiated objects can be lost!

- If we were to reassign a to something else, we'd never be able to get the original Object back!



# IntList and Linked Data Structures

---

Lecture 3, CS61B, Spring 2024

Goals: Building a List

Primitive Types

Reference Types

Parameter Passing

Instantiation of Arrays

**IntList and Linked Data Structures**

Let's define an IntList as an object containing two member variables:

- `int first;`
- `IntList rest;`

And define two versions of the same method:

- `size()`
- `iterativeSize()`

## Coding Demo: Adding to End of IntList

---

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;
```

```
}
```

## Coding Demo: Adding to End of IntList

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    public static void main(String[] args) {  
        IntList L = new IntList();  
        L.first = 5;  
        L.rest = null;  
  
    }  
}
```

## Coding Demo: Adding to End of IntList

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    public static void main(String[] args) {  
        IntList L = new IntList();  
        L.first = 5;  
        L.rest = null;  
  
        L.rest = new IntList();  
        L.rest.first = 10;  
  
    }  
}
```

## Coding Demo: Adding to End of IntList

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    public static void main(String[] args) {  
        IntList L = new IntList();  
        L.first = 5;  
        L.rest = null;  
  
        L.rest = new IntList();  
        L.rest.first = 10;  
  
        L.rest.rest = new IntList();  
        L.rest.rest.first = 15;  
  
    }  
}
```

[Java Visualizer](#)



## Coding Demo: Adding to Start of IntList

---

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;
```

```
}
```

## Coding Demo: Adding to Start of IntList

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    public IntList(int f, IntList r) {  
        first = f;  
        rest = r;  
    }  
}
```

}

## Coding Demo: Adding to Start of IntList

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    public IntList(int f, IntList r) {  
        first = f;  
        rest = r;  
    }  
  
    public static void main(String[] args) {  
        IntList L = new IntList(15, null);  
        L = new IntList(10, L);  
        L = new IntList(5, L);  
  
    }  
}
```

[Java Visualizer](#)

IntList.java

```
public class IntList {
    public int first;
    public IntList rest;

    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }

    public static void main(String[] args) {
        IntList L = new IntList(15, null);
        L = new IntList(10, L);
        L = new IntList(5, L);

        System.out.println(L.size()); // should print out 3
    }
}
```

## Coding Demo: IntList size

IntList.java

```
public class IntList {
    public int first;
    public IntList rest;

    /** Return the size of the list using... recursion! */
    public int size() {

    }

}
```

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    /** Return the size of the list using... recursion! */  
    public int size() {  
        if (rest == null) {  
  
        }  
  
    }  
  
}
```

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    /** Return the size of the list using... recursion! */  
    public int size() {  
        if (rest == null) {  
            return 1;  
        }  
    }  
}
```

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    /** Return the size of the list using... recursion! */  
    public int size() {  
        if (rest == null) {  
            return 1;  
        }  
        return 1 + this.rest.size();  
    }  
}
```



## Coding Demo: IntList iterativeSize

IntList.java

```
public class IntList {
    public int first;
    public IntList rest;

    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }

    public static void main(String[] args) {
        IntList L = new IntList(15, null);
        L = new IntList(10, L);
        L = new IntList(5, L);

        System.out.println(L.iterativeSize()); // should also print out 3
    }
}
```

## Coding Demo: IntList iterativeSize

IntList.java

```
public class IntList {
    public int first;
    public IntList rest;

    /** Return the size of the list using no recursion! */
    public int iterativeSize() {

    }
}
```

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    /** Return the size of the list using no recursion! */  
    public int iterativeSize() {  
        IntList p = this;  
  
    }  
  
}
```

## Coding Demo: IntList iterativeSize

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    /** Return the size of the list using no recursion! */  
    public int iterativeSize() {  
        IntList p = this;  
        int totalSize = 0;  
  
    }  
}
```

## Coding Demo: IntList iterativeSize

IntList.java

```
public class IntList {
    public int first;
    public IntList rest;

    /** Return the size of the list using no recursion! */
    public int iterativeSize() {
        IntList p = this;
        int totalSize = 0;
        while (p != null) {

        }

    }
}
```

## Coding Demo: IntList iterativeSize

IntList.java

```
public class IntList {
    public int first;
    public IntList rest;

    /** Return the size of the list using no recursion! */
    public int iterativeSize() {
        IntList p = this;
        int totalSize = 0;
        while (p != null) {
            totalSize += 1;
        }
    }
}
```

## Coding Demo: IntList iterativeSize

IntList.java

```
public class IntList {
    public int first;
    public IntList rest;

    /** Return the size of the list using no recursion! */
    public int iterativeSize() {
        IntList p = this;
        int totalSize = 0;
        while (p != null) {
            totalSize += 1;
            p = p.rest;
        }
    }
}
```

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    /** Return the size of the list using no recursion! */  
    public int iterativeSize() {  
        IntList p = this;  
        int totalSize = 0;  
        while (p != null) {  
            totalSize += 1;  
            p = p.rest;  
        }  
        return totalSize;  
    }  
}
```

[Java Visualizer](#)



## Challenge

Write a method `int get(int i)` that returns the *i*th item in the list.

- For simplicity, OK to assume the item exists.
- Front item is the 0th item.

Ways to work:

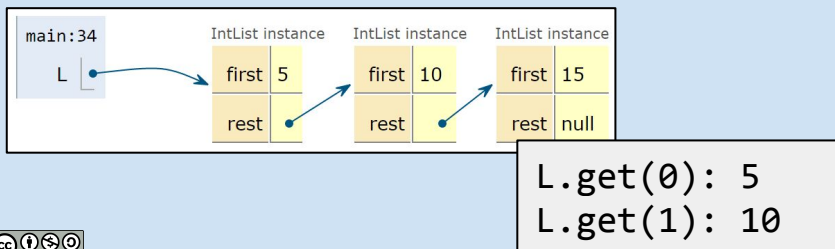
- Paper (best)
- Laptop (see lectureCode repo)
  - exercises/lists1/IntList.java
- In your head (worst)

See the video online for a solution:

[https://www.youtube.com/watch?v=qnmxD\\_21DNk](https://www.youtube.com/watch?v=qnmxD_21DNk)

```
public class IntList {
    public int first;
    public IntList rest;
    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }

    /** Return the size of this IntList. */
    public int size() {
        if (rest == null) {
            return 1;
        }
        return 1 + this.rest.size();
    }
    ...
}
```



IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    public IntList(int f, IntList r) {  
        first = f;  
        rest = r;  
    }  
  
    public static void main(String[] args) {  
        IntList L = new IntList(15, null);  
        L = new IntList(10, L);  
        L = new IntList(5, L);  
  
        System.out.println(L.get(1)); // should print out 10  
    }  
}
```

## Coding Demo: IntList get

IntList.java

```
public class IntList {
    public int first;
    public IntList rest;

    /** Return the ith item of this IntList. */
    public int get(int i) {

    }

}
```

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    /** Return the ith item of this IntList. */  
    public int get(int i) {  
        if (i == 0) {  
  
        }  
  
    }  
}
```

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    /** Return the ith item of this IntList. */  
    public int get(int i) {  
        if (i == 0) {  
            return first;  
        }  
    }  
}
```

IntList.java

```
public class IntList {  
    public int first;  
    public IntList rest;  
  
    /** Return the ith item of this IntList. */  
    public int get(int i) {  
        if (i == 0) {  
            return first;  
        }  
        return rest.get(i - 1);  
    }  
}
```

[Java Visualizer](#)

What is your comfort level with recursive data structure code?

- A. Very comfortable.
- B. Comfortable.
- C. Somewhat comfortable.
- D. I have never done this.

For further practice with IntLists, fill out the code for the methods listed below in the **lists1/exercises/ExtraIntListPractice.java** in **lectureCode** github directory.

- `public static IntList incrList(IntList L, int x)`
  - Returns an IntList identical to L, but with all values incremented by x.
  - Values in L cannot change!



- `public static IntList dincrList(IntList L, int x)`
  - Returns an IntList identical to L, but with all values incremented by x.
  - Not allowed to use 'new' (to save memory).<sup>3</sup>

