

CMSI 2120 Exam I Review: Practice Problems

Oct 13, 2026

Practice for Exam I (Lectures 1-10), worked on paper. Try every part before checking the Answers at the end.

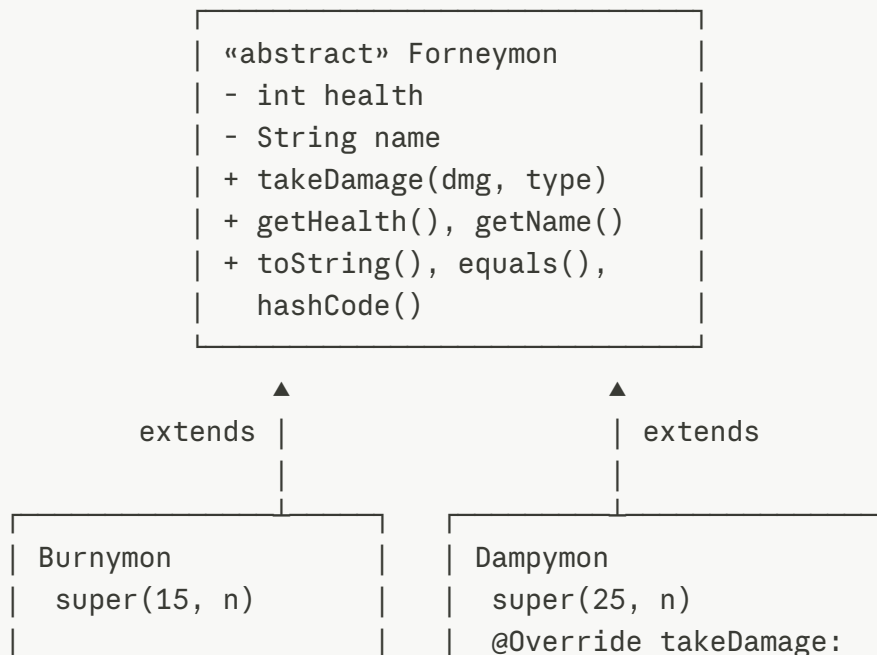
Part 1: Java warm-up

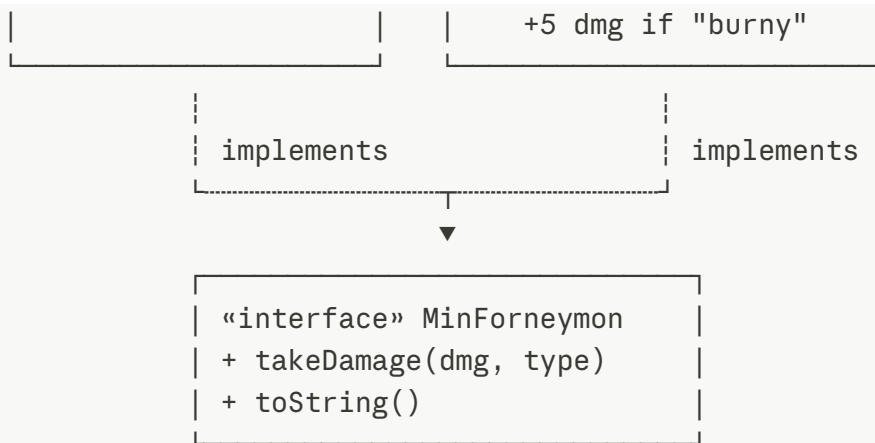
For each line, write what it prints, or say whether it fails to compile or throws an exception.

1. `int x = (int) 7.9; then print x`
2. `double d = 7 / 2; then print d`
3. `String s = "3"; then print s + s`
4. `"forney".charAt(3)`
5. `int[] a = new int[4]; a[4] = 1;`
6. `int n = 5; if (n) { System.out.println("yes"); }`

Part 2: Forneymon, inheritance and equality

Use this class hierarchy for every problem in this part. Solid arrows mean `extends`, dotted lines mean `implements`, `-` is private and `+` is public.





Forneymon's `toString()` returns the name twice with a space ("Ash Ash"). Its `equals` returns true when both objects are the same class and have the same name.

2A. What prints?

```

Forneymon d = new Dampymon("Drip");
Burnymon b = new Burnymon("Blaze");
d.takeDamage(3, "burny");
b.takeDamage(3, "dampy");
System.out.println(d.getHealth());
System.out.println(b.getHealth());
System.out.println(b);
  
```

2B. Does each line compile? Explain why or why not.

1. `Forneymon f = new Forneymon(10, "Glitch");`
2. `MinForneymon m = new Burnymon("Ash"); m.getHealth();`
3. `b.health = 100;` (written in a separate test class)
4. `class Steamymon extends Burnymon, Dampymon { }`
5. `Forneymon f = new Burnymon("Ash");`

2C. Equality. After these lines run, give the value of each expression.

```

Burnymon a = new Burnymon("Ash");
Burnymon b = new Burnymon("Ash");
Burnymon c = a;
Dampymon d = new Dampymon("Ash");
c.takeDamage(5, "dampy");
  
```

1. `a == b`

2. `a.equals(b)`
3. `a == c`
4. `a.equals(d)`
5. `a.getHealth()`

2D. A student writes `public boolean equals(Forneymon other)` instead of `public boolean equals(Object other)`. Why is this a problem, and what does `@Override` do about it?

Part 3: Stack and heap

```
public static void heal(Forneymon f, int amt) {
    amt = amt * 2;
    f.takeDamage(-amt, "heal");
    f = new Burnymon("Clone");
    // ← marked line
}

public static void main(String[] args) {
    int amt = 3;
    Burnymon b = new Burnymon("Sol");
    heal(b, amt);
    System.out.println(amt + " " + b.getHealth() + " " + b.getName());
}
```

3A. Draw the stack and heap after each line runs. Include every stack frame (`main`, `heal`, and `takeDamage` while it runs), every local variable, every object, and every reference arrow.

3B. What does the stack and heap look like at the marked line?

3C. What does `main` print?

3D. After `heal` returns, which object is eligible for garbage collection, and why?

Part 4: IntArrayList

`IntArrayList` starts with an `items` array of 8 slots and `size = 0`. `insertAt` shifts elements right, and `removeAt` shifts elements left and then does `size--`.

4A. Trace. Draw all 8 slots of `items` and the value of `size` after each call.

```

IntArraylist list = new IntArraylist();
list.append(5);
list.append(7);
list.append(9);
list.insertAt(3, 1);
list.removeAt(0);

```

Then answer:

1. What does `list.getAt(2)` return?
2. What does `list.getAt(3)` do? What value is stored in `items[3]`?
3. Starting from a new list, on which `append` does the array first grow, and to what length?

4B. Bug hunt. Explain in plain English what this method is supposed to do. Then give two inputs where it returns the wrong answer, and fix it.

```

public boolean contains(int target) {
    for (int i = 0; i < items.length; i++) {
        if (items[i] == target) { return true; }
    }
    return false;
}

```

4C. Change the bound. This is the `shiftLeft` helper used by `removeAt`:

```

private void shiftLeft(int index) {
    for (int i = index; i < this.size - 1; i++) {
        this.items[i] = this.items[i + 1];
    }
}

```

A student changes the loop condition to `i < this.size` so that the stale value past the end gets overwritten. When does this break, and what happens?

Part 5: IntLinkedList

`IntLinkedList` is singly linked. Each `Node` holds `data` and `next`, and the list holds `head` and `size`.

5A. Trace. Draw the chain of nodes, `head`, and `size` after each line, and circle the references each line changes.

```
IntLinkedList list = new IntLinkedList();
list.prepend(3);
list.prepend(1);
list.append(4);
list.insertAt(2, 1);
list.removeAt(2);
```

5B. Order matters. These are the last lines of `insertAt`, after `prev` has been moved to the node before the insertion spot:

```
Node toInsert = new Node(toAdd);
toInsert.next = prev.next;    // (1)
prev.next = toInsert;        // (2)
```

What goes wrong if lines (1) and (2) are swapped? Draw it.

5C. Bug hunt.

```
public void removeAt(int index) {
    if (index < 0 || index >= size) {
        throw new IllegalArgumentException("index out of legal range");
    }
    Node prev = head;
    for (int i = 0; i < index - 1; i++) {
        prev = prev.next;
    }
    prev.next = prev.next.next;
    size--;
}
```

- Trace `removeAt(0)`, `removeAt(1)`, and `removeAt(2)`, each on a fresh list `[1, 2, 4]`. For each, write what it should give and what it actually gives.
- What happens with `removeAt(0)` on the list `[7]`?
- Fix the method.

5D. Compare. For each operation, say which is faster, `IntArrayList` or `IntLinkedList` (with only a `head` reference), and why.

Operation	Faster	Why
<code>getAt(i)</code>		
<code>append(x)</code>		
<code>prepend(x)</code>		
<code>removeAt(0)</code>		

Part 6: Short answer

Answer each in 1–3 sentences.

1. What is the difference between an abstract data type and a data structure? Give an example of each.
2. Why make fields `private`?
3. What is the difference between a `static` method and a non-static method?
4. When is `size == items.length` in an `IntArrayList`, and what happens on the next `add`?
5. Why is `Node` a private inner class of `IntLinkedList`?
6. When is an object eligible for garbage collection?
7. If you override `equals`, what else should you override, and why?
8. Why does `IntList l = new IntLinkedList(); l.prepend(1);` fail to compile?

Answers

Part 1

1. `7`. Casting a double to an int truncates; it never rounds.
2. `3.0`. `7 / 2` is integer division (3), which is then widened to a double.
3. `33`. `+` on Strings concatenates. (`Integer.parseInt(s) * 2` would give 6.)
4. `'n'`. Strings are zero-indexed and use `.charAt()` and `.length()`, never `[i]`.
5. It compiles but throws `ArrayIndexOutOfBoundsException`; the valid indices are 0–3.
6. Compile error. Java conditions must be `boolean`; there is no truthiness.

Part 2

2A. 17, 12, Blaze Blaze. The variable `d` has type `Forneymon`, but the object is a `Dampymon`, so `Dampymon`'s override runs: $25 - (3 + 5) = 17$. `Burnymon` uses `Forneymon`'s `takeDamage`, which ignores the type: $15 - 3 = 12$. `println` calls the inherited `toString`.

2B.

Line	Compiles?	Why
1	No	<code>Forneymon</code> is abstract, so it can't be instantiated
2	No	The declared type decides what you can call, and <code>MinForneymon</code> has no <code>getHealth</code>
3	No	<code>health</code> is private
4	No	A Java class can extend only one class
5	Yes	A subclass object can be stored in a superclass variable

2C.

Expression	Value	Why
<code>a == b</code>	false	Two separate objects
<code>a.equals(b)</code>	true	Same class and same name
<code>a == c</code>	true	<code>c</code> is an alias: it points at the same object as <code>a</code>
<code>a.equals(d)</code>	false	Different classes (<code>getClass()</code> differs)
<code>a.getHealth()</code>	10	The damage was done through the alias <code>c</code>

2D. With a `Forneymon` parameter, the method overloads `equals` instead of overriding it, so any code that calls `equals(Object)` (including Java's collections) still uses `Object`'s identity check. Adding `@Override` makes the compiler reject it, because no superclass method has that signature.

Part 3

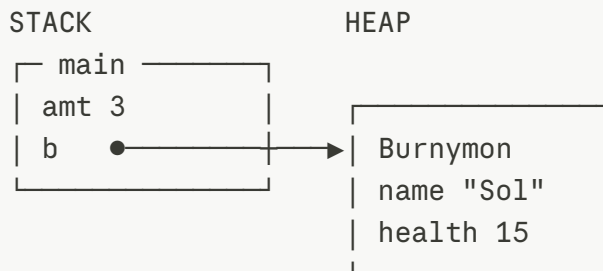
3A and 3B. One snapshot per action. The answer to 3B is Step 7.

Step 1: `int amt = 3;`



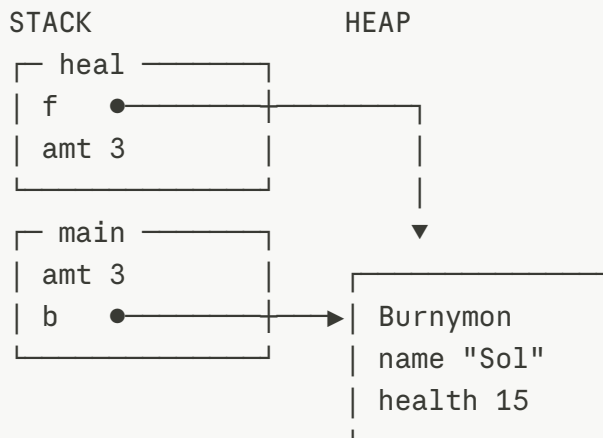
A primitive local lives directly in main's frame.

Step 2: `Burnymon b = new Burnymon("Sol");`



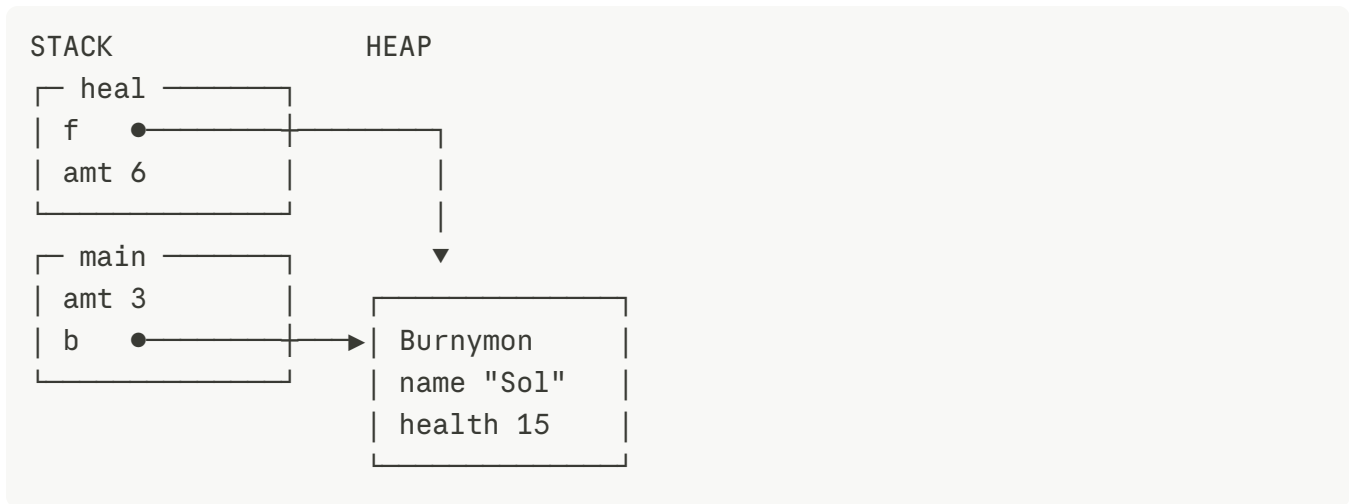
`new` makes an object on the heap; `b` holds an arrow to it.

Step 3: `heal(b, amt);` is called



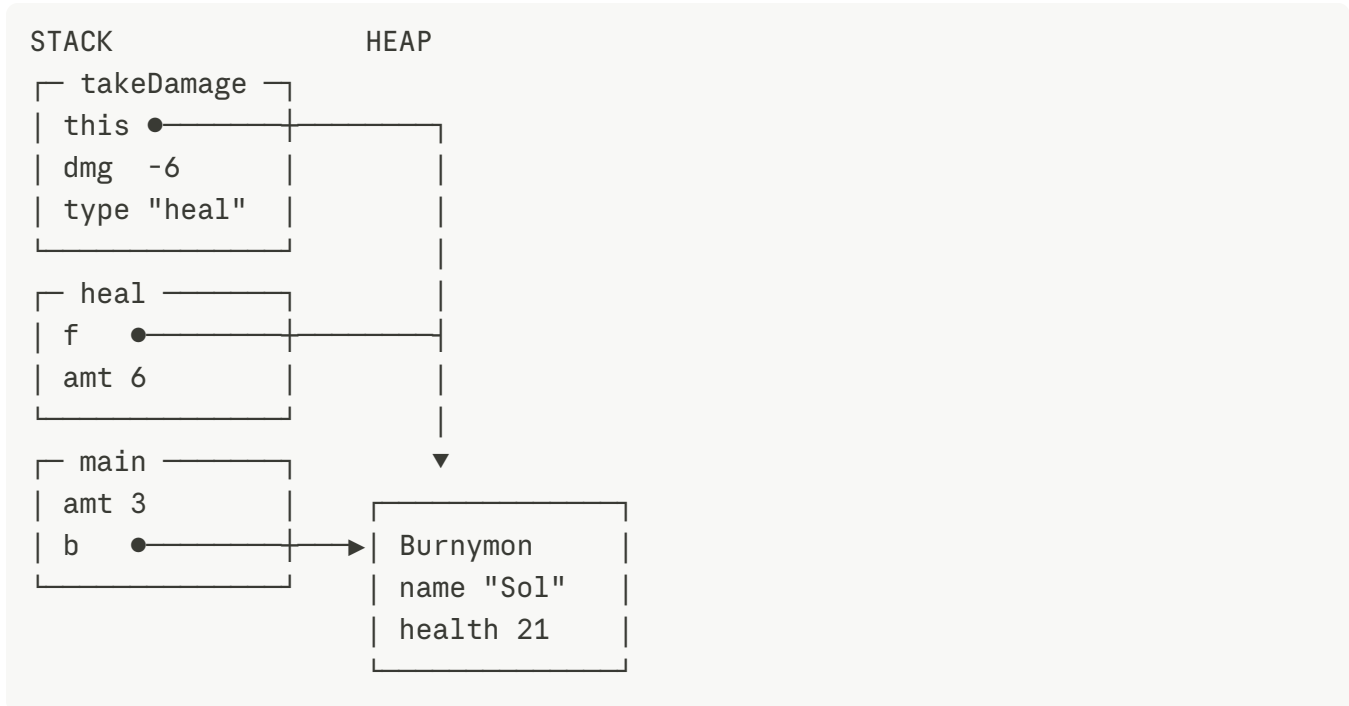
A new frame is pushed. Arguments are copied: `amt` gets a copy of 3, and `f` gets a copy of `b`'s arrow, so both point at the same Sol.

Step 4: `amt = amt * 2;`



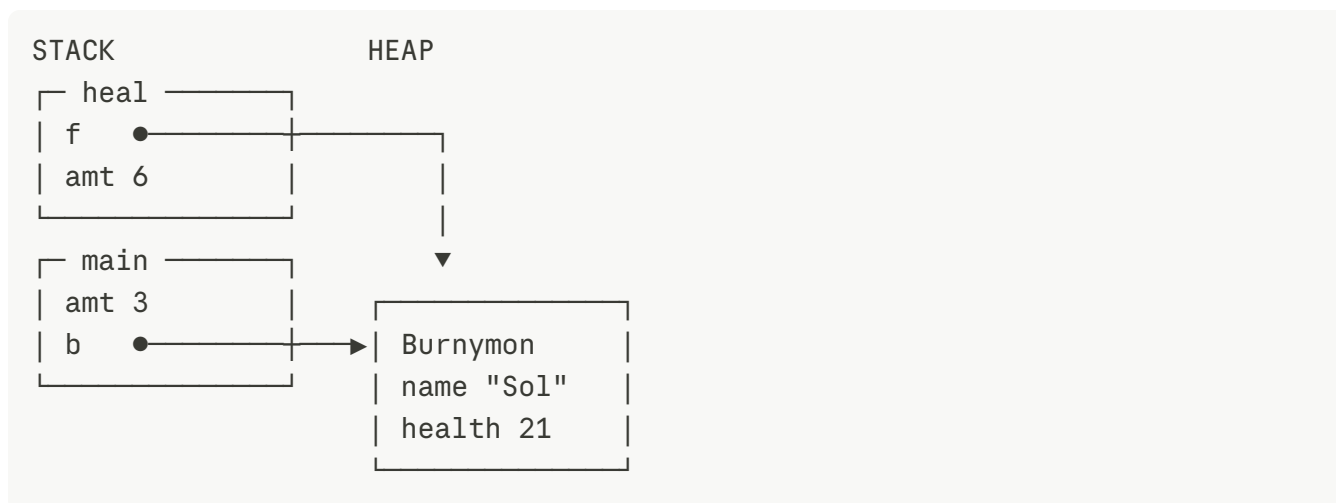
Only heal's `amt` changes. main's `amt` is a separate variable and stays 3.

Step 5: `f.takeDamage(-amt, "heal");` (inside the call)



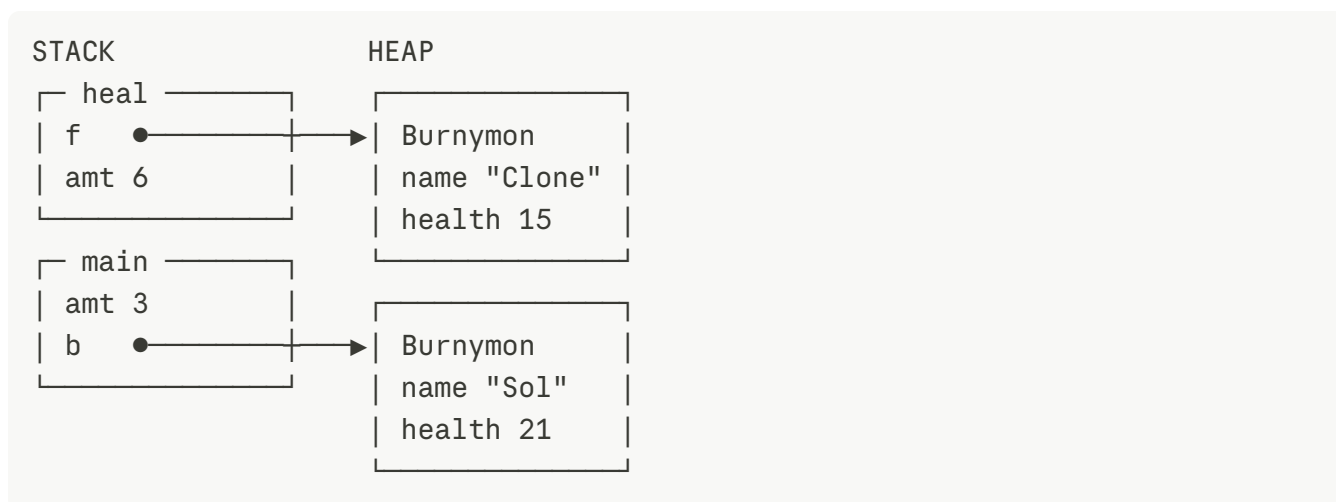
Calling a method pushes a frame. `this` is a copy of `f`'s arrow, so `health -= dmg` changes the shared object: $15 - (-6) = 21$. (`type` is really a reference to a String object; it's written as a value here to keep the drawing simple.)

Step 6: `takeDamage` returns



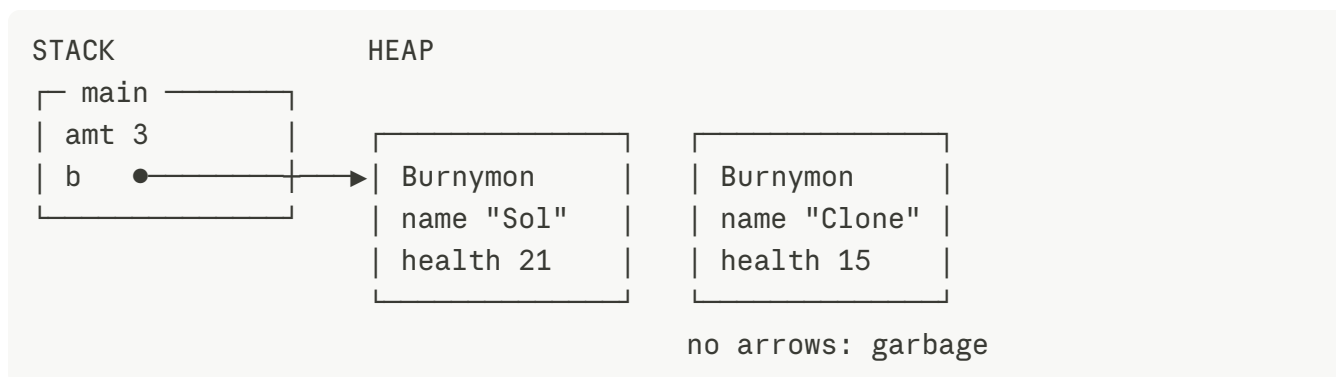
Its frame is popped, but Sol keeps health 21 because the change was made to the heap object.

Step 7: `f = new Burnymon("Clone");` (the marked line)



A second object is created and only heal's arrow moves to it. **b** still points at Sol.

Step 8: `heal` returns



heal's frame is popped. Nothing points to Clone anymore.

3C. `3` `21` `Sol` : main's unchanged `amt` , Sol's new health, and Sol's name.

3D. Clone. Its only reference was heal's local `f` , which disappeared when heal's frame was popped.

The three ideas: primitives are copied (Step 4); a copied reference still reaches the same object (Step 5); reassigning a parameter only moves the local arrow (Step 7).

Part 4

4A.

After	items (indices 0-7)	size
three appends	<code>[5, 7, 9, 0, 0, 0, 0, 0]</code>	<code>3</code>
<code>insertAt(3, 1)</code>	<code>[5, 3, 7, 9, 0, 0, 0, 0]</code>	<code>4</code>
<code>removeAt(0)</code>	<code>[3, 7, 9, 9, 0, 0, 0, 0]</code>	<code>3</code>

1. `9` .

2. It throws `IllegalArgumentException` , because `3` is not less than `size` . `items[3]` still holds a stale `9`, but anything at index `size` or beyond is not part of the list.

3. The 9th append. The 8 values are copied into a new array of length 16.

4B. It should return true when `target` is one of the list's values. The loop runs to `items.length` (the capacity) instead of `size` , so it also reads unused and stale slots.

Two wrong answers: a new, empty list returns true for `contains(0)` , because the array holds eight zeros; and `[3, 7, 9]` after `removeAt(2)` still returns true for `contains(9)` , because the array is still `[3, 7, 9, 9, ...]` . Fix: `i < size` .

4C. It breaks when the array is full (`size == items.length`). On the last pass the loop reads `items[size]` , which is past the end, so it throws `ArrayIndexOutOfBoundsException` . The stale value is harmless anyway, because nothing reads past `size` . To clear it safely, set `items[size - 1] = 0` after the loop instead.

Part 5

5A.

After	Chain	size	References changed
<code>prepend(3)</code>	<code>head → 3 → null</code>	1	<code>head</code>
<code>prepend(1)</code>	<code>head → 1 → 3 → null</code>	2	the new node's <code>next</code> , then <code>head</code>
<code>append(4)</code>	<code>head → 1 → 3 → 4 → null</code>	3	node 3's <code>next</code> (after walking to the end)
<code>insertAt(2, 1)</code>	<code>head → 1 → 2 → 3 → 4 → null</code>	4	the new node's <code>next</code> (<code>→ 3</code>), then node 1's <code>next</code> (<code>→ 2</code>)
<code>removeAt(2)</code>	<code>head → 1 → 2 → 4 → null</code>	3	node 2's <code>next</code> now skips to 4; node 3 becomes garbage

5B. Line (2) runs first, so `prev.next` already points to the new node. Then line (1) sets the new node's `next` to `prev.next`, which is the new node itself. The node points to itself, and the rest of the list after it is lost.

5C.

Call (on a fresh list)	Should give	Actually gives
<code>removeAt(0)</code> on <code>[1, 2, 4]</code>	<code>[2, 4]</code>	<code>[1, 4]</code> : wrong node removed
<code>removeAt(1)</code> on <code>[1, 2, 4]</code>	<code>[1, 4]</code>	<code>[1, 4]</code>
<code>removeAt(2)</code> on <code>[1, 2, 4]</code>	<code>[1, 2]</code>	<code>[1, 2]</code>
<code>removeAt(0)</code> on <code>[7]</code>	<code>[]</code>	<code>NullPointerException</code>

For index 0 the loop runs zero times, the same as for index 1, so `prev` stays at `head` and the method removes the second node. On `[7]`, `prev.next` is `null`, so `prev.next.next` throws. The first node has no node before it, so index 0 needs its own case:

```

if (index == 0) {
    head = head.next;
} else {
    Node prev = head;
    for (int i = 0; i < index - 1; i++) {
        prev = prev.next;
    }
    prev.next = prev.next.next;
}
size--;

```

5D.

Operation	Faster	Why
<code>getAt(i)</code>	<code>IntArrayList</code>	It jumps straight to <code>items[i]</code> ; the linked list walks <code>i</code> nodes from <code>head</code>
<code>append(x)</code>	<code>IntArrayList</code>	Usually one write (occasionally a copy to grow); the linked list walks to the end (unless it keeps a <code>tail</code>)
<code>prepend(x)</code>	<code>IntLinkedList</code>	One new node and <code>head</code> moves; the array list shifts every element right
<code>removeAt(0)</code>	<code>IntLinkedList</code>	<code>head = head.next</code> ; the array list shifts every element left

Part 6

1. An ADT says what operations exist (the List interface: `getAt` , `append` , ...). A data structure is a concrete implementation of it (`IntArrayList`, `IntLinkedList`).
2. So that only the class's own methods can change them, and outside code can't put an object in an invalid state.
3. A static method belongs to the class and has no `this` . A non-static method is called on an object and can use that object's fields.
4. Only when the array is full. The next add calls `checkAndGrow` , which allocates an array twice as long and copies everything over.
5. Users work with ints through the list's methods, so they never see the nodes and can't break the links.

6. When no references point to it anymore.
7. `hashCode` , so that objects that are equal also have the same hash code.
8. The declared type `IntList` decides which methods you can call, and the `IntList` interface has no `prepend` .