`this` in Java — step-by-step memory (Stack vs Heap) with diagrams

A focused, visual walkthrough that shows **exactly** where `this` and other things live as your Java program runs. Includes step-by-step ASCII diagrams you can use as a reference when thinking about stack frames, references, and objects on the heap.

Example code (the running example used below)

```
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    void move(int dx, int dy) {
        this.x += dx;
        this.y += dy;
    }

    Point returnThis() {
        return this;
    }
}

public class Main {
    public static void main(String[] args) {
        Point p = new Point(1, 2);
        p.move(3, 4);
        Point q = p.returnThis();
        p = null;
        // q still references the object
    }
}
```

Quick memory primer (short)

- **Heap**: where objects and arrays live. Instance fields live inside objects on the heap.

- **Stack** (per thread): contains method frames (a frame = local variable array + operand stack + return address). Local variables (including references like `p` and the `this` reference) live in the stack frame for that method.
- **Method Area / Metaspace**: class metadata (bytecode, static fields, method objects). Implementation detail: static fields are not per-instance.

Key point: `this` is a *reference value* stored in the method frame (stack). The **object** that `this` refers to is on the **heap**.

Legend for all diagrams below

- `STACK` frames are shown top → bottom (top is the most recently pushed frame)
- `HEAP` shows objects with a fake address like `Point@100` for clarity
- `->` indicates a reference value (pointer)

Step 0: JVM starts `main`

Action: JVM creates the `main` frame and passes `args`.

```
STACK (top → bottom)
--------------------------------
main frame:
  locals: args -> String[] (ref)
--------------------------------

HEAP
--------------------------------
(empty besides JVM internals and any interned Strings)
--------------------------------
```

Nothing related to `Point` yet.

Step 1: `Point p = new Point(1, 2);` — allocate object, call constructor

Action sequence (simplified): 1. `new Point(1,2)` allocates a `Point` object on the **heap** (call it `Point@100`). Fields initially default (`x=0, y=0`). 2. JVM pushes the constructor frame `Point.<init>(int,int)` onto the **stack**. That frame has an implicit `this` local, and parameters for the ctor. 3. Constructor executes `this.x = x; this.y = y;` — these write into the heap object. 4. Constructor returns; the reference to `Point@100` is stored in `main`'s local `p`.

```
STACK (top → bottom)
--------------------------------
Point.<init>(int,int) frame (while running):
  this -> Point@100    // reference value in constructor's local array
  param x = 1
  param y = 2
--------------------------------
main frame (after constructor returns):
  p -> Point@100
  args -> String[]
--------------------------------

HEAP
--------------------------------
Point@100 { x: 1, y: 2 }
--------------------------------
```

Notes: - `this` inside the constructor is a reference **variable** in the constructor's stack frame. It points to the heap object `Point@100`. - The object itself (with fields `x, y`) lives in the heap.

Step 2: `p.move(3, 4);` — call an instance method

Action: call `move`, which receives an implicit `this` reference to the same `Point@100`.

```
STACK (top → bottom)
--------------------------------
move(int,int) frame (during p.move):
  this -> Point@100   // reference to the heap object
  dx = 3
  dy = 4
--------------------------------
main frame:
  p -> Point@100
  args -> String[]
--------------------------------

HEAP
--------------------------------
Point@100 { x: 4, y: 6 }
--------------------------------
```

What happened: `move` read the fields from the heap and updated them; updates are visible to every reference that points to `Point@100`.

Step 3: `Point q = p.returnThis();` — return `this` reference

Action: `returnThis()` returns the *same* reference value (the value of `this`) back to `main`, which stores it into `q`.

```
STACK (top → bottom)
--------------------------------
returnThis() frame:
  this -> Point@100
--------------------------------
main frame (after return):
  p -> Point@100
  q -> Point@100
  args -> String[]
--------------------------------

HEAP
--------------------------------
Point@100 { x: 4, y: 6 }
--------------------------------
```

Note: Both `p` and `q` are local reference variables in `main`'s frame that contain the same reference value (they alias the same heap object).

Step 4: `p = null;` — remove one reference

Action: main's local `p` is set to `null`. `q` still points to the object.

```
STACK (top → bottom)
--------------------------------
main frame:
  p -> null
  q -> Point@100
  args -> String[]
--------------------------------

HEAP
--------------------------------
Point@100 { x: 4, y: 6 }
--------------------------------
```

Garbage-collection note: The object is **not** eligible for GC because `q` still holds a reference. Only when there are zero reachable references (from stack, static fields, other GC roots) will the object be eligible.

What is *in the Stack* (detail)

- **Method frames** (one per active method call). Each frame contains:
- **Local variable array**: holds primitive values (e.g., `int dx`) and reference values (e.g., `p`, `q`, and the implicit `this`).
- **Operand stack**: used for intermediate calculations (JVM bytecode detail).
- **Return address / frame metadata**.
- **Important:** local *reference values* are stored in the stack frame; but the object instances they reference live on the heap.

Examples from the run: `this -> Point@100` (inside `move` frame) and `q -> Point@100` (inside `main` frame) are both reference values inside the stack.

What is *in the Heap* (detail)

- **Objects and their instance fields** (e.g., `Point@100` with `x` and `y`).
- **Arrays** and their contents.
- **Often** (implementation detail) the string pool and class-related objects live in special areas (method area/metaspace or in heap depending on JVM).
- **Important:** the heap stores the actual object state. Any method that references the object (via any reference) sees the same state.

Common confusions and clarifications

- **Is `this` on the heap?** No — `this` is a reference value inside the method frame on the **stack**. It points to a heap object.
- **Can you change `this` to point somewhere else?** No — `this` is an implicit final-like reference inside an instance method; Java forbids assigning to `this`.
- **Where are static fields?** In the method area / metaspace (conceptually separate from instance storage). They're not part of any instance on the heap.
- **Inner classes:** A non-static inner instance holds a reference to its outer instance as a field (that *outer* reference is stored in the inner object on the **heap**). The `this` reference inside the inner's instance methods still lives on that method's **stack** frame.

Compact final diagram (snapshot after Step 3, both p and q set)

```
STACK (top → bottom)
--------------------------------
(main) frame:
  p -> Point@100
  q -> Point@100
  args -> String[]
--------------------------------

HEAP
--------------------------------
Point@100 { x:4, y:6 }
--------------------------------
```

If you want this visualized differently

If you'd like I can: - convert these ASCII diagrams into PNG images, or - add a variant that shows the JVM operand stack and local variable indexes, or - show the inner-class `Outer.this` example.

Tell me which of those (PNG, operand-stack view, inner-class) you'd like next and I will produce it right away.

Short takeaway: `this` is a reference stored **on the stack** (in the current method frame); the object `this` points to is stored on the **heap**.