Class 10: Equality

6.102 — Software Construction
Spring 2024

Warmup

Start your exercise collaboration

Exercise: yellkey.com/**probably**

Nanoquiz: yellkey.com/**easy**

Look at `Point` and `Stroke` …

Draw a snapshot diagram for:

```
const seg = new Stroke(5, 10, 20, 15, Color.BLACK);
```

… and then there will be a couple clicker questions.

Warmup

Start your exercise collaboration

Exercise: yellkey.com/**probably**

Nanoquiz: yellkey.com/**easy**

Look at `Point` and `Stroke` …

Draw a snapshot diagram for:

```
const seg = new Stroke(5, 10, 20, 15, Color.BLACK);
```

… and then there will be a couple clicker questions.

How many **arrows** are in your snapshot diagram?

Nanoquiz

- This quiz is just for you and your own brain:
    - closed-book, closed-notes
    - nothing else on your screen
- Lower your laptop screen when you’re done

yellkey.com/**easy**

Which of these implementations of `equalValue()` are **correct**?

```
/** Immutable set of characters */
export class CharSet {
  private readonly s: string;
  ...
  public equalValue(that: CharSet): boolean {

    (A) return this.s === that.s;

    (B) return this.s.equalValue(that.s);

    (C) return this.toString() === that.toString();

-or-(D) // none of them

  }
  public toString(): string {
    ... // correct implementation of toString() spec
  }
}
```

Which of these implementations of `equalValue()` are **correct**?

```
/** Immutable set of characters */
export class CharSet {
  private readonly s: string;
   // AF(s) = { c | c is in s }
   // RI(s) = true
  ...
  public equalValue(that: CharSet): boolean {

      (A) return this.s === that.s;

      (B) return this.s.equalValue(that.s);

      (C) return this.toString() === that.toString();

-or-(D) // none of them

  }
  public toString(): string {
    ... // correct implementation of toString() spec
  }
}
```

yellkey.com/**probably**

Equality

Implement `equalValue()` for `Point`

→ pass the tests for `Point.equalValue` in `equalsTest.ts` yellkey.com/**probably**

Equality

Implement `equalValue()` for `Point`

→ pass the tests for `Point.equalValue` in `equalsTest.ts`

…and for `Stroke`

Equality

yellkey.com/**probably**

Implement `equalValue()` for `Point`

→ pass the tests for `Point.equalValue` in `equalsTest.ts`

…and for `Stroke`

…and for `LineSegment`

Equality

yellkey.com/**probably**

Implement `equalValue()` for `Point`

→ pass the tests for `Point.equalValue` in `equalsTest.ts`

...and for `Stroke`

...and for `LineSegment`

Where can you change your `equalValue()` implementations to use observers rather than rep fields?

Which of these implementations of `Stroke.equalValue()` are **both correct and good**?

```
(A) return this.start.equalValue(that.start)
        && this. end .equalValue(that.end)
        && this.color === that.color;

(B) return this.start.x === that.start.x
        && this.start.y === that.start.y
        && this. end .x === that. end .x
        && this. end .y === that. end .y
        && this.color   === that.color;

(C) if (this.start.equalValue(that.start)) {
      if (this.end.equalValue(that.end)) {
        if (this.color === that.color) {
          return true; } } }
    return false;
```

Which of these implementations of `LineSegment.equalValue()` are **both correct and good**?

```
(A) return this.p1.equalValue(that.p1)
        && this.p2.equalValue(that.p2);

(B) return (   this.p1.equalValue(that.p1)
            && this.p2.equalValue(that.p2) )
        || (   this.p1.equalValue(that.p2)
            && this.p2.equalValue(that.p1) );

(C) for (const p of this.endpoints()) {
        if ( ! that.endpoints().includes(p)) {
            return false; } }
    return true;

(D) return this.length() === that.length();

(E) return this.toString() === that.toString();
```

Which of these implementations of `LineSegment.equalValue()` are **both correct and good**?

```
(A) return this.p1.equalValue(that.p1)
        && this.p2.equalValue(that.p2);

(B) return (   this.p1.equalValue(that.p1)
            && this.p2.equalValue(that.p2) )
        || (   this.p1.equalValue(that.p2)
            && this.p2.equalValue(that.p1) );

(C) for (const p of this.endpoints()) {
        if ( ! that.endpoints().includes(p)) { // watch out
            return false; } }
    return true;

(D) return this.length() === that.length();

(E) return this.toString() === that.toString();
```

Hashability in TS/JS vs. Python

A *hashable* type can be safely stored in a set and used as a map/dict key

Are these types hashable?

Hashability in TS/JS vs. Python

A *hashable* type can be safely stored in a set and used as a map/dict key

Are these types hashable?

```
// number in TS/JS
const x = 7
const s = new Set<number>()

s.add(x)
s.has(x)
s.has(7)
```

```
# int in Python
x = 7
s = set()

s.add(x)
x in s
7 in s
```

Hashability in TS/JS vs. Python

A *hashable* type can be safely stored in a set and used as a map/dict key

Are these types hashable?

```
// Array in TS/JS
const x: Array<number> = [1,2]
const s = new Set<Array<number>>()

s.add(x)
s.has(x)
s.has([1,2])

x.push(3)
s.has(x)
s.has([1,2,3])
```

```
# list in Python
x = [1,2]
s = set()

s.add(x)
x in s
[1,2] in s

x.append(3)
x in s
[1,2,3] in s
```

Hashability in TS/JS vs. Python

A *hashable* type can be safely stored in a set and used as a map/dict key

Are these types hashable?

```
// Point (from today and ps0)
const x: Point = new Point(3,4)
const s = new Set<Point>()

s.add(x)
s.has(x)
s.has(new Point(3,4))
```

```
# tuple in Python
x = (3,4)
s = set()

s.add(x)
x in s
(3,4) in s
```

Hashability in TS/JS vs. Python

A *hashable* type can be safely stored in a set and used as a map/dict key

Are these types hashable?

```
// Flashcard (from ps1)
const x: Flashcard = Flashcard.make("yes","oui")
const s = new Set<Flashcard>()

s.add(x)
s.has(x)
s.has(Flashcard.make("yes","oui"))
```

```
# tuple in Python
x = ("yes","oui")
s = set()

s.add(x)
x in s
("yes","oui") in s
```

Hashability

```
/** Mutable line art. */
export class LineArt {
    ...
    public add(stroke: Stroke): void { ... }
    public remove(stroke: Stroke): void { ... }
    public equalValue(that: LineArt): boolean { ... }
    ...
}
```

Hashability

```
/** Mutable line art. */
export class LineArt {
    ...
    public add(stroke: Stroke): void { ... }
    public remove(stroke: Stroke): void { ... }
    public equalValue(that: LineArt): boolean { ... }
    ...
}
```

If we are a client of mutable `LineArt`, which are likely to work as expected?

```
(A)
const userPictures: Map<string, LineArt>   // each user has one picture
(B)
const pictureAuthors: Map<LineArt, string> // each picture has one author
(C)
const strokeCounts: Map<LineArt, number>   // when we edit, increment count
```

Hashability

```
/** Mutable line art. */
export class LineArt {
    ...
    public add(stroke: Stroke): void { ... }
    public remove(stroke: Stroke): void { ... }
    public equalValue(that: LineArt): boolean { ... }
    ...
}
```

If we are a client of mutable `LineArt`, which are likely to work as expected?

```
(A)
const userPictures: Map<string, LineArt>   // each user has one picture
(B)
const pictureAuthors: Map<LineArt, string> // each picture has one author
(C)
const strokeCounts: Map<LineArt, number>   // when we edit, increment count
```

Mutable keys are compared with `===`. OK — just remember they can be mutated!

Hashability

```
/** Mutable line art. */
export class LineArt {
    ...
    public add(stroke: Stroke): void { ... }
    public remove(stroke: Stroke): void { ... }
    public equalValue(that: LineArt): boolean { ... }
    ...
}
```

If we implement `LineArt` using our immutable types, which are likely to work as expected?

```
(A)
private readonly strokes: Set<Stroke>               // unique strokes
(B)
private readonly visibility: Map<Stroke, boolean>   // toggle visibility
(C)
private readonly layers: Map<number, Array<Stroke>> // multiple layers
```

Hashability

```
/** Mutable line art. */
export class LineArt {
    ...
    public add(stroke: Stroke): void { ... }
    public remove(stroke: Stroke): void { ... }
    public equalValue(that: LineArt): boolean { ... }
    ...
}
```

If we implement `LineArt` using our immutable types, which are likely to work as expected?

```
(A)
private readonly strokes: Set<Stroke>                // unique strokes
(B)
private readonly visibility: Map<Stroke, boolean>   // toggle visibility
(C)
private readonly layers: Map<number, Array<Stroke>> // multiple layers
```

`Set`/`Map` use `===` to compare, but that’s wrong for immutable types like `Stroke`