Class 4: Specifications

6.102 — Software Construction
Spring 2024

Choosing types

In `warmup.ts`:

Choose the types in place of `TODO`,
so that the specifications make sense

Exercise:  yellkey.com/**stand**

Nanoquiz:  yellkey.com/**guess**

`clicker.mit.edu/6.102`

- Please leave empty seats on the ends of rows, not in the middle, so everyone can easily find a seat.
- Please take your phone off Wi-Fi, it helps people who have old computers, even if you have a shiny new computer.

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/**guess**

Choosing types

In `warmup.ts`:

Choose the types in place of `TODO`,
so that the specifications make sense

Exercise:  yellkey.com/**stand**

Nanoquiz:  yellkey.com/**guess**

`clicker.mit.edu/6.102`

- Please leave empty seats on the ends of rows, not in the middle, so everyone can easily find a seat.
- Please take your phone off Wi-Fi, it helps people who have old computers, even if you have a shiny new computer.

Step zero

TurtleSoup

```
/**
 * Draw a square.
 *
 * @param turtle the turtle context
 * @param sideLength length of each side, must be >= 0
 */
function drawSquare(turtle: Turtle, sideLength: number): void
```

TurtleSoup

```
/**
 * Draw a square.
 *
 * @param turtle the turtle context
 * @param sideLength length of each side, must be >= 0
 */
function drawSquare(turtle: Turtle, sideLength: number): void
```

A. Precondition
B. Postcondition
C. Neither
D. Both

TurtleSoup

```
/**
 * Draw a square.
 *
 * @param turtle the turtle context
 * @param sideLength length of each side, must be >= 0
 */
function drawSquare(turtle: Turtle, sideLength: number): void
```

A. Precondition
B. Postcondition
C. Neither
D. Both

TurtleSoup

```
/**
 * Draw a square.
 *
 * @param turtle the turtle context
 * @param sideLength length of each side, must be >= 0
 */
function drawSquare(turtle: Turtle, sideLength: number): void
```

A. Precondition
B. Postcondition
C. Neither
D. Both

TurtleSoup

```
/**
 * Find a sequence of turns and moves that visits points in order.
 *
 * @param points  array of N points, adjacent points distinct, ...
 * @returns an array [turn_0,move_0,...,turn_N+1]
 *    such that if turtle starts at (0,0) heading up,
 *    and does turn(turn_i) and forward(move_i) actions in order,
 *    then it will be at points[i] after move_i for all valid i,
 *    and finish heading up.
 */
function findPath(points: Array<Point>): Array<number>
```

A. Precondition
B. Postcondition
C. Neither
D. Both

TurtleSoup

```
/**
 * Find a sequence of turns and moves that visits points in order.
 *
 * @param points  array of N points, adjacent points distinct, ...
 * @returns an array [turn_0,move_0,...,turn_N+1]
 *    such that if turtle starts at (0,0) heading up,
 *    and does turn(turn_i) and forward(move_i) actions in order,
 *    then it will be at points[i] after move_i for all valid i,
 *    and finish heading up.
 */
function findPath(points: Array<Point>): Array<number>
```

A. Precondition
B. Postcondition
C. Neither
D. Both

TurtleSoup

```
/**
 * Find a sequence of turns and moves that visits points in order.
 *
 * @param points  array of N points, adjacent points distinct, ...
 * @returns an array [turn_0,move_0,...,turn_N+1]
 *    such that if turtle starts at (0,0) heading up,
 *    and does turn(turn_i) and forward(move_i) actions in order,
 *    then it will be at points[i] after move_i for all valid i,
 *    and finish heading up.
 */
function findPath(points: Array<Point>): Array<number>
```

A. Precondition
B. Postcondition
C. Neither
D. Both

Warmup

Preconditions

Exercise:  yellkey.com/**stand**

In `precond.ts`:

Fill in each `requires: ???` with an appropriate precondition
so that the spec is implementable (possible to satisfy the postcondition)

Don't change the signature or postcondition.

Preconditions

Exercise:  yellkey.com/**stand**

In `precond.ts`:

Fill in each `requires: ???` with an appropriate precondition
so that the spec is implementable (possible to satisfy the postcondition)

Don't change the signature or postcondition.

`evaluateParabola` — constrain `a.length`? constrain `x`?

`winner` — allow empty `s`?

`replace` — testing hat asks if `replace('a', {a:'b', b:'a'})` is allowed

Precondition for `replace`

When does it make sense to use a precondition?

What's the alternative?

Postconditions

In `postcond.ts`:

Fill in each `effects: ???` with an appropriate postcondition

Don't change the signature or precondition.

Postconditions

In `postcond.ts`:

Fill in each `effects: ???` with an appropriate postcondition

Don't change the signature or precondition.

`evaluateParabola` — fail fast?

`factor` — does "$p \times q = n$" by itself promise enough to the client?

`deleteAllOccurrences` — returns `void`! what do we do?

`split` — how to write the postcondition concisely?

Postcondition for `split`

```
function split(s: string, sep: string): Array<string>
// requires: sep.length = 1
// effects: returns `list` such that ???
```

Postcondition for `split`

```
function split(s: string, sep: string): Array<string>
// requires: sep.length = 1
// effects: returns `list` such that ???
```

What would you put in place of ??? (can pick more than one, to concatenate them)

A. `list` is not empty
B. `list` has no empty strings
C. it finds the first `sep` in `s`, makes that the first element of `list`, then repeats
D. `list` consists of substrings of `s`, none of which contain `sep`
E. `s` is the concatenation of `list` with one `sep` between each string in `list`

A trial spec — let's check it

```
function split(s: string, sep: string): Array<string>
// requires: sep.length = 1
// effects: returns a k-element `list` such that
//          text = list[0] + sep + list[1] + ... + sep + list[k-1]
```

A trial spec — let's check it

```
function split(s: string, sep: string): Array<string>
// requires: sep.length = 1
// effects: returns a k-element `list` such that
//          text = list[0] + sep + list[1] + ... + sep + list[k-1]
```

Which of these input/output pairs is allowed by the spec above? (can pick more than one)

A. `split( "ab,cd,ef", "," ) → [ "ab", "cd", "ef" ]`
B. `split( "ab,cd,ef", "," ) → [ "ab", "cd,ef" ]`
C. `split( "ab,cd,ef", "," ) → [ "a", "b", "c", "d", "e", "f" ]`
D. `split( "ab,cd,ef", "," ) → [ "ab", "", "cd", "", "ef" ]`
E. none of the above

Iterating!

```
function split(s: string, sep: string): Array<string>
// requires: sep.length = 1
// effects: returns a k-element `list` such that
//          no elements of `list` contain `sep`, and
//          text = list[0] + sep + list[1] + ... + sep + list[k-1]
```

Iterating!

```
function split(s: string, sep: string): Array<string>
// requires: sep.length = 1
// effects: returns a k-element `list` such that
//          no elements of `list` contain `sep`, and
//          text = list[0] + sep + list[1] + ... + sep + list[k-1]
```

Finally, let's rewrite this as TypeDoc:

```
/**
 * Splits a string into parts separated by a separator character
 * @param s    string to split
 * @param sep  separator to split on; requires sep.length = 1
 * @returns    a k-element `list` such that
 *               no elements of `list` contain `sep`, and
 *               text = list[0] + sep + list[1] + ... + sep + list[k-1]
 */
function split(s: string, sep: string): Array<string>
```