Poll Everywhere

PollEv.com/javabear text **javabear** to **22333**

On the left is a *buggy* `med3()` definition. Which JUnit assertion will detect the bug?

```
1  /** Returns median of `a`, `b`, `c`. */
2  static int med3(int a, int b, int c) {
3    if (a >= b) {
4      return Math.max(b, c);
5    } else if (a >= c) { // a < b
6      return a;
7    } else { // a is smallest
8      return Math.min(b, c);
9    }
10 }
```

a can be median even if ≥ b

```
assertEquals(2,med3(1,2,3)); (A)
```

```
assertEquals(2,med3(2,3,1)); (B)
```

```
assertEquals(2,med3(2,1,3)); (C)
```

```
assertEquals(2,med3(3,1,2)); (D)
```

Lecture 4: Loop Invariants

CS 2110

September 3, 2026

Today's Learning Outcomes

18. Identify the loop invariant of an iterative method involving an array and visualize it using an array diagram.

19. Use an array diagram to develop an iterative method that maintains the visualized invariant.

20. Write precise specifications for methods involving arrays that use range notation.

Loop Anatomy

```
int sum = 0;
for (int i = 0; i < a.length; i++) {
    sum += a[i];
}
```

Loop body

Increment: loop variable(s)

Initialization:
declare + initialize
loop variable(s)

Loop guard:
boolean expression
true ⇒ run loop body
false ⇒ "fall through" loop

`while` Loops

Built from all the same components as `for` loops.
Every `for` loop can be transformed into a `while` loop.

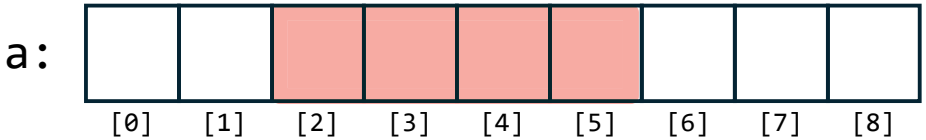
```
int sum = 0;
for (int i=0; i<a.length; i++) {
    sum += a[i];
}
```

```
int sum = 0;
int i=0;
while (i<a.length) {
    sum += a[i];
    i++;
}
```

We'll focus on `while` loops today, but everything also applies to `for` loops.

Range Notation

A **range** of an array is a contiguous subset of its entries.



$a[2..5] = a(1..6) = a[2..6)$

include endpoints

exclude endpoints

Special cases:

from start: $a[0..i] = a[0..i]$, $a(0..i] = a(0..i]$

from end: $a[i..a.length-1] = a[i..a.length-1]$

$a[i..j]$ empty when $i > j$

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

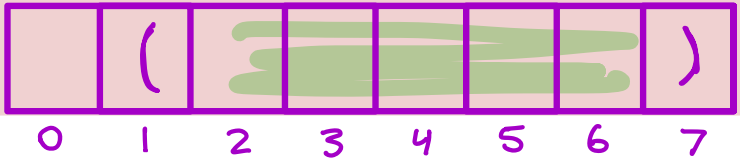
If `arr.length == 8`, how many elements belong to the range `arr(1..)` ?

4	**(A)**
5	**(B)**
6	**(C)**
7	**(D)**

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

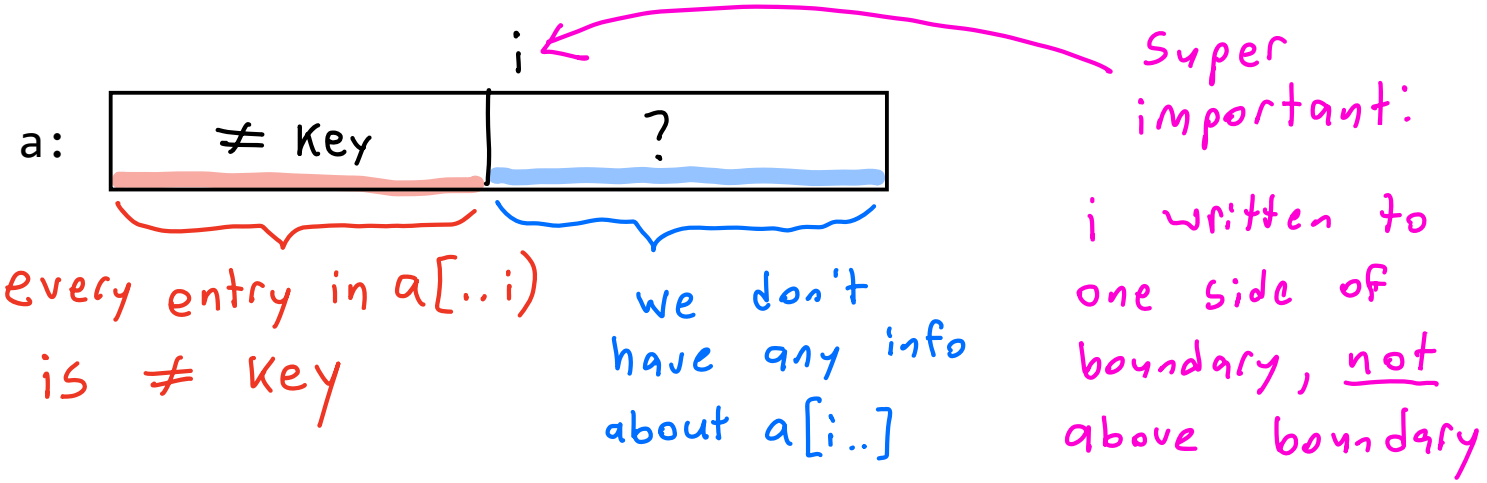
If `arr.length == 8`, how many elements belong to the range `arr(1..)` ?



4	(A)
5	(B)
6	(C)
7	(D)

Range Properties and Array Diagrams

We use **array diagrams** as a tool to visualize properties of array ranges.



Writing "Loopy" Code

```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) { ... }
```

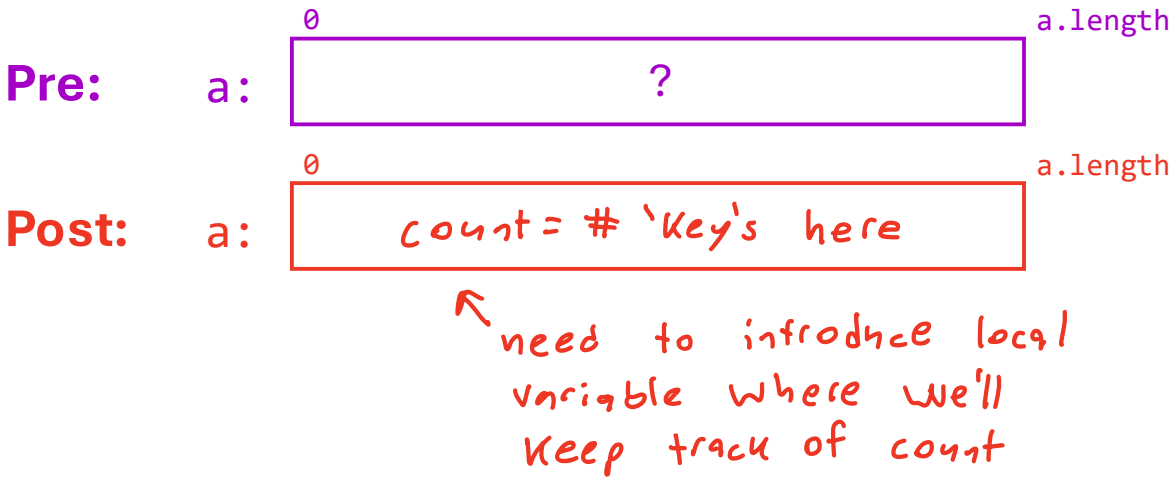
Pre: a:

when we enter the method/loop, we don't know anything about a's contents

*no pre-conditions

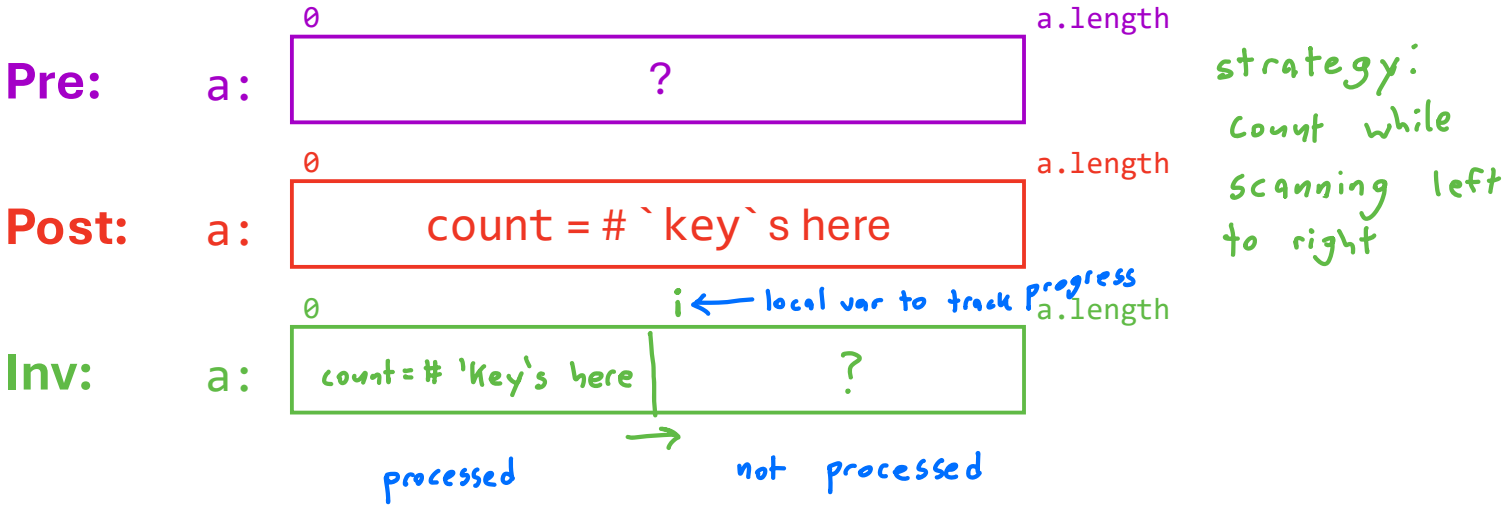
Writing "Loopy" Code

```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) { ... }
```



Writing "Loopy" Code

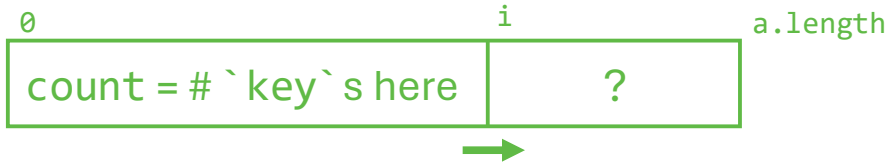
```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) { ... }
```



(Loop) Invariants

An **invariant** is an assertion about code that is true at multiple predetermined points of its execution.

A loop invariant describes relationship between loop + other local vars that is true every time loop guard is evaluated.



Loop Inv: count = # of occurrences of key in a[..i)

Once we have a loop invariant diagram, writing its loop becomes mechanical.

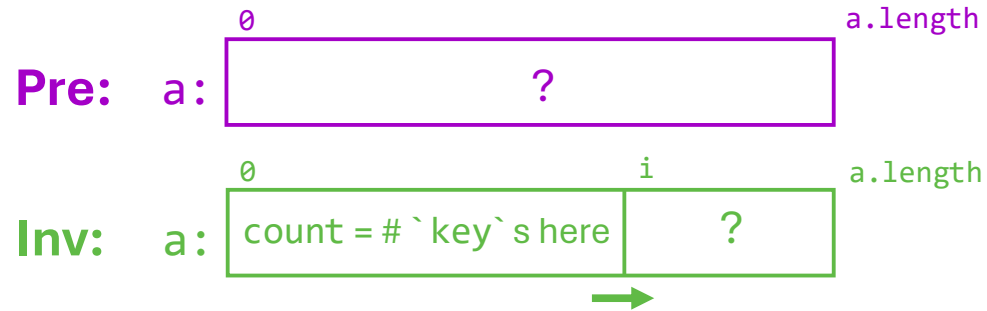
Developing a Loop: Initialization

```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) {
3      int i = 0;    // i = next index of 'a' to check
4
5      int count = 0;
6    /* Loop Inv: `count` = # of occurrences of `key` in `a[..i)` */
7    while (...) { ... }
8  }
```

(start by checking a[0])

a[..0) is empty

Initialize local variables to *truthify* the loop invariant when we enter the loop.



Poll Everywhere

PollEv.com/javabear text `javabear` to `22333`

Which of these is true *immediately after* we fall through the loop body?

- **loop invariant** is `true` **loop guard** is `true` (A)
- **loop invariant** is `true` **loop guard** is `false` (B)
- **loop invariant** is `false` **loop guard** is `true` (C)
- **loop invariant** is `false` **loop guard** is `false` (D)

Developing a Loop: Guard + Post-condition

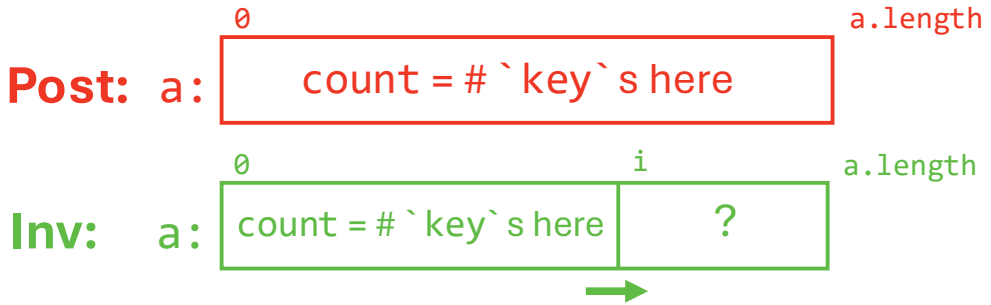
```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) {
3    int i = 0; // next index of `a` to check
4    int count = 0;
5    /* Loop Inv: `count` = # of occurrences of `key` in `a[..i)` */
6    while (i < a.length) { ... } // done when i == a.length
7    return count;
8  }
```

a[..a.length) is all of a

negate* continue looping while i < a.length

Guard the loop on a condition that *falsifies* once the loop is done with its work.

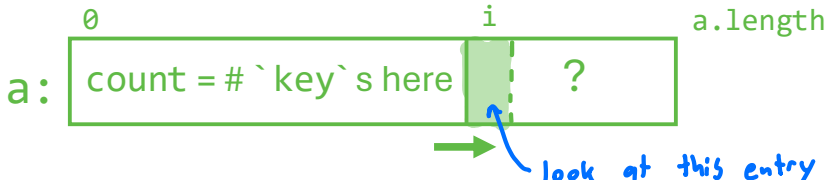
Use loop invariant to meet method post-condition.



Developing a Loop: Body

In each iteration:

- Progress toward the loop's goal
- Restore the loop invariant




```
1  /** Returns # of occurrences of `key` in array `a`. */
2  static int frequencyOf(int key, int[] a) {
3    int i = 0;  int count = 0;
4    /* Loop Inv: `count` = # of occurrences of `key` in `a[..i)` */
5    while (i < a.length) {
6      if (a[i] == key) {
7        count++;
8      }
       i++;
9    }
10   return count;
11 }
```

Example 2: `argmin()`

```
1  /** Returns an *index* of the minimum element in `a`.
2   *  Requires that `a.length > 0`. */
3  static int argmin(double[] a) { ... }
```

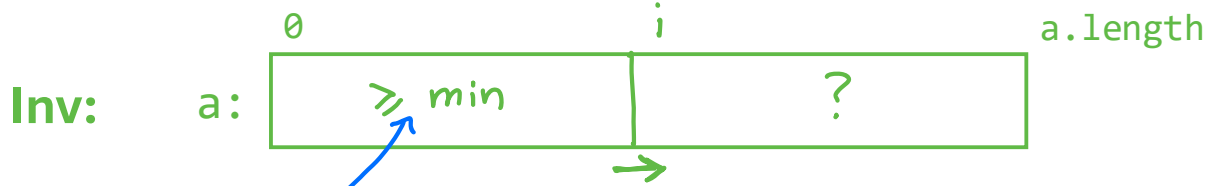
```
argmin(new double[]{1.0, 3.5, 4.2, 0.7, 6.3, 2.8}) = 3
                    0    1    2    3
```

What do we need to keep track of to compute the `argmin()`?

- which entries we've checked (i)
- the smallest element we've seen (min)
- where this smallest element is (loc)

argmin() Invariant Diagram

similar strategy sweeping left → right will work

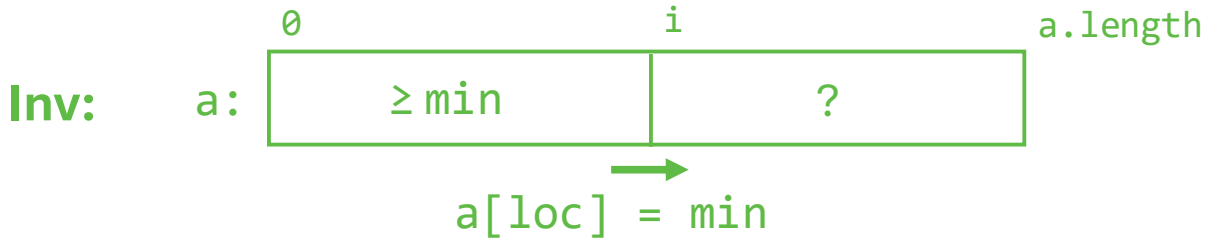


state fact that min is smallest value we've seen as a range property

this isn't a range property, so we'll write it below the diagram

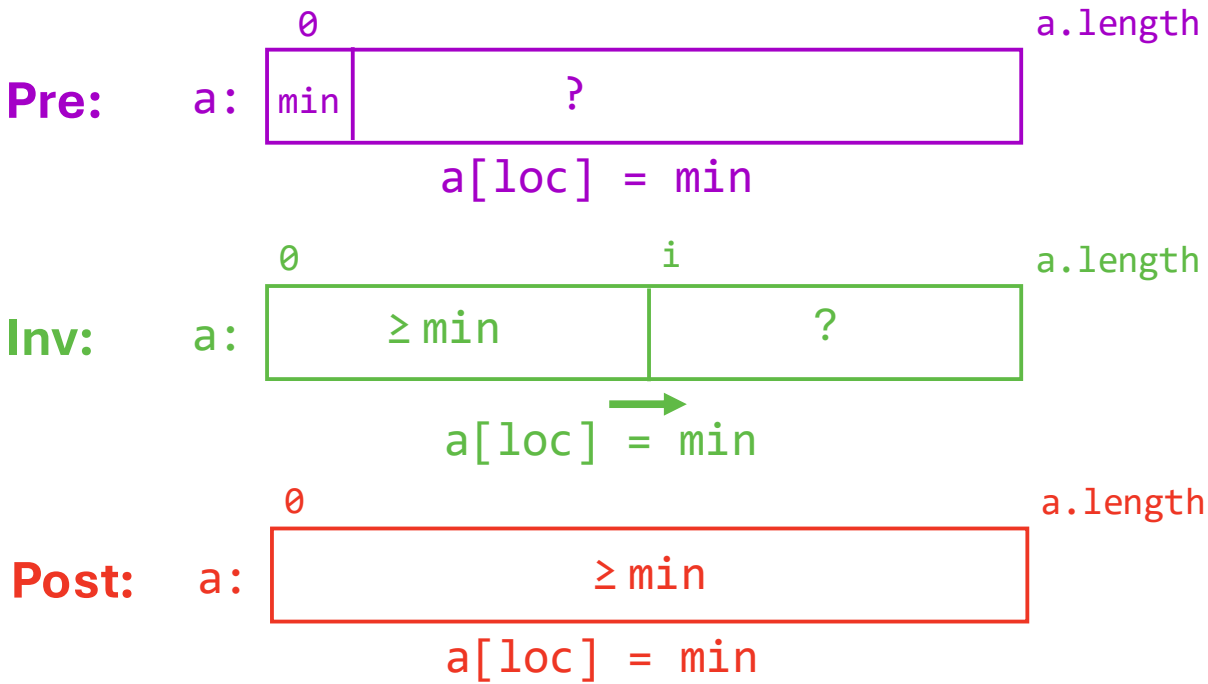
Exercise

Use the array diagram we developed to complete the definition of `argmin()`.




```
1  /** Returns an *index* of the minimum element in `a`.
2   *  Requires that `a.length > 0`. */
3  static int argmin(double[] a) { ... }
```

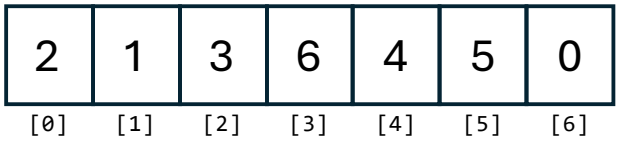
argmin() Pre and Post Diagrams

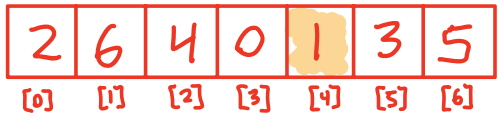


Coding Demo: `argmin()`

Example 3: paritySplit()

```
1  /** Rearranges `a` so that all even elements appear before
2   *  all odd elements. Returns the index of the first odd
3   *  element, or `a.length` if all elements are even. */
4  static int paritySplit(int[] a) { ... }
```





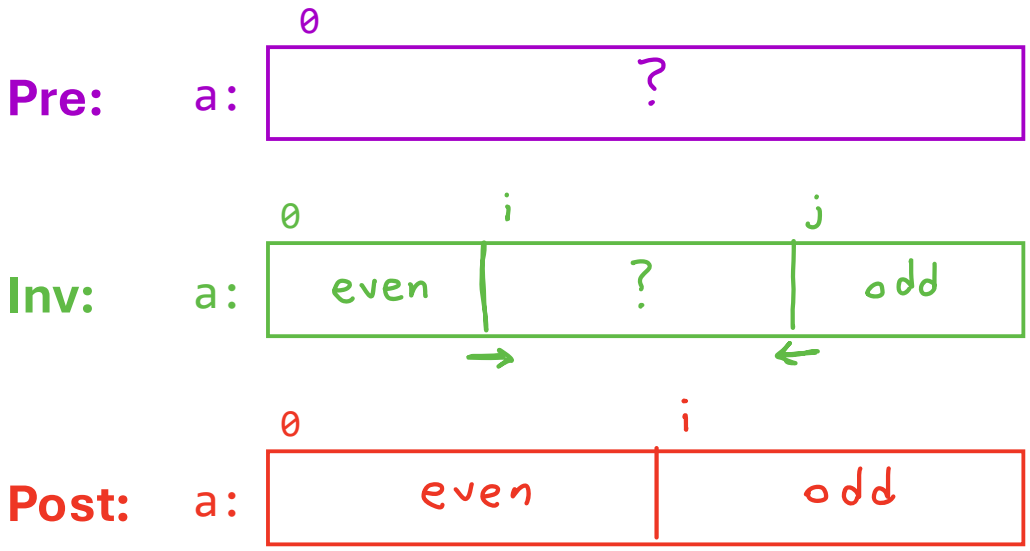
or

return 4

(under specified)

* look at lecture code to see how we test this!

paritySplit() Array Diagrams



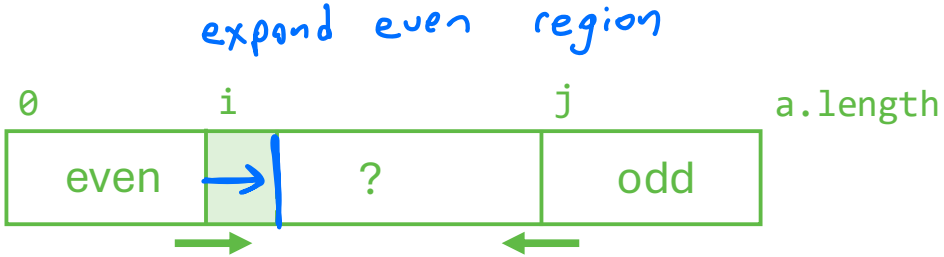
* We can put i, j on the other sides of the boundaries, giving another (correct) definition. See Lecture Exercise 4.7.

Coding Demo: `paritySplit()`

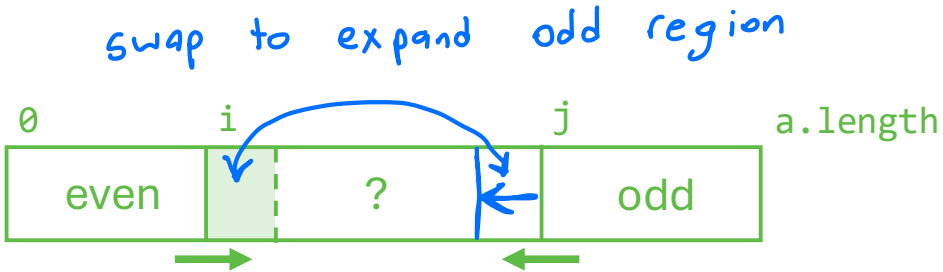
`paritySplit()` Loop Body

Inspect `a[i]` and consider two cases:

1. `a[i]` is even



2. `a[i]` is odd



Review: Steps for Developing Loops

1. Think about loop progress to sketch its invariant.
2. Identify local variables: one per boundary, perhaps others for properties.
3. Use the “Inv” diagram to write the loop invariant comment.
4. Slide the “Inv” boundaries to their "Pre" positions
   - Use overlaps to initialize the loop variables.
   - Use the loop invariant to initialize other local variables.
5. Slide the “Inv" boundaries to their "Post" positions
   - Use overlaps to to determine the loop guard.
   - Use the loop invariant to satisfy the method post-conditions
6. Develop the loop body so that it:
   - Makes progress toward the post-condition.
   - Re-establishes the loop invariant.