Loops

EECS1021:
Object Oriented Programming:
from Sensors to Actuators
Winter 2019

CHEN-WEI WANG

YORK UNIVERSITÉ UNIVERSITY

Learning Outcomes

Understand about *Loops* :

- Motivation: *Repetition* of *similar actions*
- Two common loops: `for` and `while`
- Primitive vs. Compound Statements
- *Nesting* loops within `if` statements
- *Nesting* `if` statements within loops
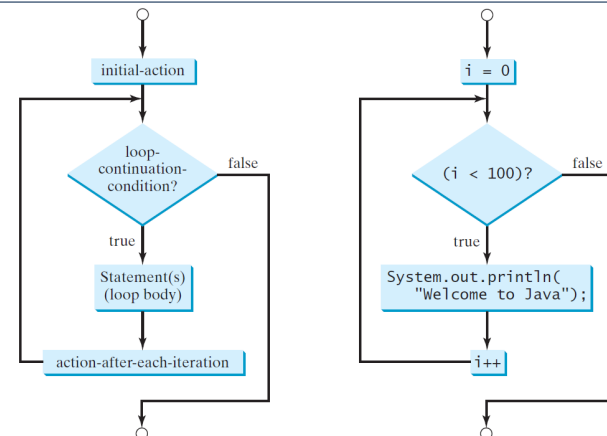- Common Errors and Pitfalls

---

Motivation of Loops

- We may want to **repeat** the *similar action(s)* for a (bounded) number of times.

  e.g., Print the "Hello World" message for 100 times

  e.g., To find out the maximum value in a list of numbers
- We may want to **repeat** the *similar action(s)* under certain circumstances.

  e.g., Keep letting users enter new input values for calculating the BMI until they enter "quit"
- *Loops* allow us to repeat similar actions either
  - **for** a specified number of times; or
  - **while** a specified condition holds *true*.

The `for` Loop (1)

```
for (int i = 0; i < 100; i ++) {
    System.out.println("Welcome to Java!");
}
```

The `for` Loop (2)

```
for (int i = 0; i < 100; i ++) {
   System.out.println("Welcome to Java!");
}
```

i	i < 100	Enter/Stay Loop?	Iteration	Actions
0	0 < 100	True	1	print, i ++
1	1 < 100	True	2	print, i ++
2	2 < 100	True	3	print, i ++
...				
99	99 < 100	True	100	print, i ++
100	100 < 100	False	–	–

- The number of *iterations* (i.e., 100) corresponds to the number of times the loop body is executed.
- # of times that we check the *stay condition (SC)* (i.e., 101) is # of iterations (i.e., 100) plus 1. [*True* × 100; *False* × 1]

The `for` Loop (3)

```
for ( int i = 0 ; i < 100; i ++ ) {
   System.out.println("Welcome to Java!");
}
```

- The *"initial-action"* is executed *only once*, so it may be moved right before the `for` loop.
- The *"action-after-each-iteration"* is executed repetitively to *make progress*, so it may be moved to the end of the `for loop` body.

```
int i = 0;
for (; i < 100; ) {
   System.out.println("Welcome to Java!");
   i ++;
}
```

The `for` Loop: Exercise (1)

Compare the behaviour of this program

```
for (int count = 0; count < 100; count ++) {
   System.out.println("Welcome to Java!");
}
```

and this program

```
for (int count = 1; count < 201; count += 2) {
   System.out.println("Welcome to Java!");
}
```

- Are the outputs same or different?
- It is similar to asking if the two intervals

$$[0, 1, 2, \ldots, 100) \text{ and } [1, 3, 5, \ldots, 201)$$

  contain the same number of integers.
- *Same*, both loop bodies run exactly 100 times and do not depend on the value of *count*.

The `for` Loop: Exercise (2)

Compare the behaviour of this program

```
int count = 0;
for (; count < 100; ) {
   System.out.println("Welcome to Java " + count + "!");
   count ++; /* count = count + 1; */
}
```

and this program

```
int count = 1;
for (; count <= 100; ) {
   System.out.println("Welcome to Java " + count + "!");
   count ++; /* count = count + 1; */
}
```

Are the outputs same or different? *Different*, both loop body run exactly 100 times and depend on the value of *count*.

Compare the behaviour of the following three programs:

```java
for (int i = 1; i <= 5 ; i ++) {
  System.out.print(i); }
```

Output: `12345`

```java
int i = 1;
for ( ; i <= 5 ; ) {
  System.out.print(i);
  i ++; }
```
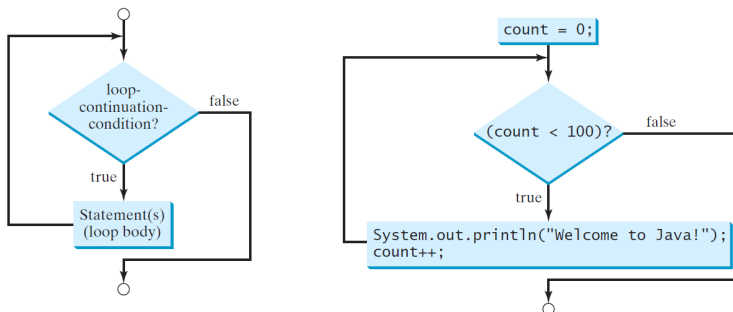
Output: `12345`

```java
int i = 1;
for ( ; i <= 5 ; ) {
  i ++;
  System.out.print(i); }
```

Output: `23456`

```java
int j = 3;
while (j < 103) {
  System.out.println("Welcome to Java!");
  j ++; /* j = j + 1; */ }
```

j	$j < 103$	Enter/Stay Loop?	Iteration	Actions
3	3 < 103	*True*	1	print, `j ++`
4	4 < 103	*True*	2	print, `j ++`
5	5 < 103	*True*	3	print, `j ++`
...				
102	102 < 103	*True*	100	print, `j ++`
103	103 < 103	*False*	–	–

- The number of *iterations* (i.e., 100) corresponds to the number of times the loop body is executed.
- # of times that we check the *stay condition (SC)* (i.e., 101) is # of iterations (i.e., 100) plus 1. [*True* × 100; *False* × 1]

```java
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count ++; /* count = count + 1; */
}
```

Compare the behaviour of this program

```java
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count ++; /* count = count + 1; */
}
```

and this program

```java
int count = 1;
while (count <= 100) {
  System.out.println("Welcome to Java!");
  count ++; /* count = count + 1; */
}
```

Are the outputs same or different? *Same*, both loop bodies run exactly 100 times and do not depend on the value of *count*.

The `while` Loop: Exercise (2)

Compare the behaviour of this program

```java
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java " + count + "!");
  count ++; /* count = count + 1; */
}
```

and this program

```java
int count = 1;
while (count <= 100) {
  System.out.println("Welcome to Java " + count + "!");
  count ++; /* count = count + 1; */
}
```

Are the outputs same or different? *Different*, both loop body run exactly 100 times and depend on the value of *count*.

Primitive Statement vs. Compound Statement

- A **statement** is a block of Java code that modifies value(s) of some variable(s).
- An assignment (=) statement is a *primitive statement*: it only modifies its left-hand-side (LHS) variable.
- An `for` or `while` loop statement is a *compound statement*: the loop body may modify more than one variables via other statements (e.g., assignments, `if` statements, and `for` or `while` statements).
  - e.g., a loop statement may contain as its body `if` statements
  - e.g., a loop statement may contain as its body loop statements
  - e.g., an `if` statement may contain as its body `loop` statements

---

Compound Loop: Exercise (1.1)

How do you **extend** the following program

```java
System.out.println("Enter a radius value:");
double radius = input.nextDouble();
double area = radius * radius * 3.14;
System.out.println("Area is " + area);
```

with the ability to *repeatedly* prompt the user for a radius value, until they explicitly enter a negative radius value to terminate the program (in which case an error message is also printed)?

```java
System.out.println("Enter a radius value:");
double radius = input.nextDouble();
while (radius >= 0)  {
  double area = radius * radius * 3.14;
  System.out.println("Area is " + area);
  System.out.println("Enter a radius value:");
  radius = input.nextDouble();  }
System.out.println("Error: negative radius value.");
```

Compound Loop: Exercise (1.2)

Another alternative: Use a boolean variable `isPositive`

```java
1  System.out.println("Enter a radius value:");
2  double radius = input.nextDouble();
3  boolean isPositive = radius >= 0;
4  while (isPositive)  {
5    double area = radius * radius * 3.14;
6    System.out.println("Area is " + area);
7    System.out.println("Enter a radius value:");
8    radius = input.nextDouble();
9    isPositive = radius >= 0;  }
10 System.out.println("Error: negative radius value.");
```

- In **L2**: What if user enters 2? What if user enters −2?
- Say in **L2** user entered 2, then in **L8**:
  What if user enters 3? What if user enters −3?
- What if `isPositive = radius >= 0` in **L9** is missing?

Compound Loop: Exercise (1.3)

Another alternative: Use a boolean variable `isNegative`

```
1  System.out.println("Enter a radius value:");
2  double radius = input.nextDouble();
3  boolean isNegative = radius < 0;
4  while (!isNegative) {
5    double area = radius * radius * 3.14;
6    System.out.println("Area is " + area);
7    System.out.println("Enter a radius value:");
8    radius = input.nextDouble();
9    isNegative = radius < 0; }
10 System.out.println("Error: negative radius value.");
```

- In **L2**: What if user enters 2? What if user enters −2?
- Say in **L2** user entered 2, then in **L8**:
  What if user enters 3? What if user enters −3?
- What if `isNegative = radius < 0` in **L9** is missing?

Converting between `for` and `while` Loops (2)

- To convert a `for` loop to a `while` loop, move the initialization part immediately before the `while` loop and place the update part at the end of the `while` loop body.

```
for(int i = 0 ; B ; i ++ ) {
  /* Actions */
}
```

is equivalent to:

```
int i = 0;
while(B) {
  /* Actions */
  i ++;
}
```

where *B* is any valid Boolean expression.

- However, when there is a loop counter (i.e., *i*, *count*, *etc.*) that you intend to explicitly maintain, stick to a `for` loop.

Converting between `for` and `while` Loops (1)

- To convert a `while` loop to a `for` loop, leave the initialization and update parts of the `for` loop empty.

```
while(B) {
  /* Actions */
}
```

is equivalent to:

```
for( ; B ; ) {
  /* Actions */
}
```

where *B* is any valid Boolean expression.

- However, when there is not a loop counter (i.e., *i*, *count*, *etc.*) that you intend to explicitly maintain, stick to a `while` loop.

Stay Condition (SC) vs. Exit Condition (1)

- A `for(...; SC ; ...)` loop or a `while(SC)` loop
  - *stays* to repeat its body **as long as** SC evaluates to *true*.
  - *exits* **as soon as** its SC evaluates to *false*.
- Say we have two Boolean variables:

```
boolean p, q;
```

- When does the loop exit (i.e., stop repeating Action 1)?

```
while(p && q) { /* Action 1 */ }
```

  !(p && q)
  this is equivalent to !p || !q
- When does the loop exit (i.e., stop repeating Action 2)?

```
while(p || q) { /* Action 2 */ }
```

  !(p || q)
  this is equivalent to !p && !q

Stay Condition (SC) vs. Exit Condition (2)

Consider the following loop:

```
int x = input.nextInt();
while(10 <= x || x <= 20) {
  /* body of while loop */
}
```

- It compiles, but has a logical error. Why?
- Think about the *exit condition* :
  - `!(10 <= x || x <= 20)` [∵ *negation* of stay condition]
  - `!(10 <= x) && !(x <= 20)` [∵ law of disjunction]
  - `10 > x && x > 20` [∵ law of negation]
- `10 > x && x > 20` is equivalent to *false*, since there is no number smaller than 10 and larger than 20 at the same time.
- An exit condition being *false* means that there is no way to exit from the loop! [infinite loops are *BAD*!]

Problems, Data Structures, and Algorithms

- A *well-specified* *computational problem* precisely describes the desired *input/output relationship*.
  - **Input:** A sequence of *n* numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - **Output:** The maximum number *max* in the input array, such that $max \geq a_i$, where $1 \leq i \leq n$
  - An *instance* of the problem: $\langle 3, 1, 2, 5, 4 \rangle$
- A *data structure* is a systematic way to store and organize data in order to facilitate *access* and *modifications*.
- An *algorithm* is:
  - A solution to a well-specified *computational problem*
  - A *sequence of computational steps* that takes value(s) as *input* and produces value(s) as *output*
- Steps in an *algorithm* manipulate well-chosen *data structure(s)*.

Arrays: A Simple Data Structure

- An array is a *linear* sequence of elements.

940	880	830	790	750	660	650	590	510	440
0	1	2	3	4	5	6	7	8	9

- *Types* of elements in an array are the *same*.
  - an array of integers [int[]]
  - an array of doubles [double[]]
  - an array of characters [char[]]
  - an array of strings [String[]]
  - an array of booleans [boolean[]]
- Each element in an array is associated with an integer index.
- *Range of valid indices* of an array is constrained by its *size*.
  - The 1st element of an array has the index *0*.
  - The 2nd has index 1.
  - The ith element has index $i - 1$.
  - The last element of an array has the index value that is equal to the *size of the array minus one*.

Arrays: Initialization and Indexing

- Initialize a new array object with a *fixed* size:

```
String[] names = new String[10];
```

- Alternatively, initialize a new array explicitly with its contents:

```
String[] names = {"Alan", "Mark", "Tom"};
```

- Access elements in an array through indexing:

```
String first = names[0];
String last = names[names.length - 1];
```

An illegal index triggers an *ArrayInexOutOfBoundsException*.

Arrays: Iterations

- Iterate through an array using a *for-loop*:

```
for (int i = 0; i < names.length; i ++) {
    System.out.println (names[i]);
}
```

- Iterate through an array using a *while-loop*:

```
int i = 0;
while (i < names.length) {
    System.out.println (names[i]);
    i ++;
}
```

The `for` Loop: Exercise (4)

Problem: Given an array `numbers` of integers, how do you print its contents backwards?

e.g., Given array $\{1, 2, 3, 4\}$, print `4 3 2 1`.

Solution 1: Change bounds and updates of loop counter.

```
for(int i = numbers.length - 1; i >= 0; i --) {
    System.out.println(numbers[i]);
}
```

Solution 2: Change indexing.

```
for(int i = 0; i < names.length; i ++) {
    System.out.println(numbers[ names.length - i - 1 ]);
}
```

The `for` Loop: Exercise (3)

Problem: Given an array `numbers` of integers, how do you print its average?

e.g., Given array $\{1, 2, 6, 8\}$, print `4.25`.

```
int sum = 0;
for(int i = 0; i < numbers.length; i ++) {
    sum += numbers[i];
}
double average = (double) sum / numbers.length;
System.out.println("Average is " + average);
```

Q: What's the printout when the array is empty

(e.g., `int[] numbers = {};`)?

A: Division by zero (i.e., `numbers.length` is 0). Fix?

The `for` Loop: Exercise (5)

Problem: Given an array `names` of strings, how do you print its contents separated by commas and ended with a period?

e.g., Given array $\{$"*Alan*","*Mark*","*Tom*"$\}$,
print "Names: Alan, Mark, Tom."

```
System.out.print("Names:")
for(int i = 0; i < names.length; i ++) {
    System.out.print(names[i]);
    if (i < names.length - 1) {
        System.out.print(", ");
    }
}
System.out.println(".");
```

Array Iterations: Translating `for` to `while` (1)

- Use either when you intend to iterate through the <u>entire</u> array.

```
int[] a = new int[100];
for(int i = 0; i < a.length; i ++) {
  /* Actions to repeat. */
}
```

In a `for` loop, the *initialization* and *update* of the *loop counter i* are specified as part of the loop header.

```
int[] a = new int[100];
int i = 0;
while(i < a.length) {
  /* Actions to repeat. */
  i ++;
}
```

In a `while` loop, the *loop counter i*
- Is *initialized* outside and before the loop header
- Is *updated* at the end of the loop body

Array Iterations: Translating `for` to `while` (2)

- In both the `for` and `while` loops:
  - The stay/continuation conditions are *identical*.
  - The loop counter *i* is initialized only *once* before first entrance.
  - In each iteration, the loop counter *i* is executed *at the end* of the loop body.

Compound Loop: Exercise (2)

Given an integer array:

```
int[] a = {2, 1, 3, 4, -4, 10}
```

How do you print out positive numbers only?
Hint: Use a `for` loop to *iterate* over the array. In the loop body, *conditionally* print out positive numbers only.

```
1  for(int i = 0; i < a.length; i ++) {
2    if (a[i] > 0) {
3      System.out.println(a[i]);
4    }
5  }
```

Exercise: Write the equivalent using a `while` loop.

Compound Loop: Exercise (3)

Given a *non-empty* integer array, e.g., `int[] a = {2, 1, 3, 4, -4, 10}`, find out its maximum element.
Hint: *Iterate* over the array. In the loop body, maintain the *maximum found so far* and update it when necessary.

```
1  int max = a[0];
2  for(int i = 0; i < a.length; i ++) {
3    if (a[i] > max) {  max = a[i]; }
4  }
5  System.out.println("Maximum is " + max);
```

Q: What if we change the initialization in **L1** to `int max = 0`?

A: NO ∵ Contents of `a` may be all smaller than this initial value (e.g., all negatives).

Q: What if we change the initialization in **L2** to `int i = 1`?

A: YES ∵ `a[0] > a[0]` is always *false* anyway.

```
1  int[] a = {2, 1, 3, 4, -4, 10}
2  int max = a[0];
3  for(int i = 0; i < a.length; i ++) {
4    if (a[i] > max) {
5      max = a[i]; } }
6  System.out.println("Maximum is " + max);
```

i	a[i]	a[i] > max	update max?	max
0	–	–	–	2
0	2	false	N	2
1	1	false	N	2
2	3	true	Y	3
3	4	true	Y	4
4	-4	false	N	4
5	10	true	Y	10

```
1  int[] ns = {2, 3, -1, 4, 5};
2  boolean soFarOnlyPosNums = true;
3  int i = 0;
4  while (i < ns.length) {
5    soFarOnlyPosNums = soFarOnlyPosNums && (ns[i] > 0);
6    i = i + 1;
7  }
```

i	soFarOnlyPosNums	i < ns.length	stay?	ns[i]	ns[i] > 0
0	true	true	YES	2	true
1	true	true	YES	3	true
2	true	true	YES	-1	false
3	false	true	YES	4	true
4	false	true	YES	5	true
5	false	false	NO	–	–

- **Problem:** Given an array of numbers, determine if it contains *all* positive number.

```
1  int[] numbers = {2, 3, -1, 4, 5};
2  boolean soFarOnlyPosNums = true;
3  int i = 0;
4  while (i < numbers.length) {
5    soFarOnlyPosNums = soFarOnlyPosNums && (numbers[i] > 0);
6    i = i + 1;
7  }
8  if (soFarOnlyPosNums) { /* print a msg. */ }
9  else { /* print another msg. */ }
```

- Change **Line 5** to soFarOnlyPosNums = numbers[i] > 0; ?
- **Hints:** Run both versions on the following three arrays:
  1. {2, 3, 1, 4, 5, 6, 8, 9, 100}   [all positive]
  2. {2, 3, 100, 4, 5, 6, 8, 9, -1}   [negative at the end]
  3. {2, 3, -1, 4, 5, 6, 8, 9, 100}   [negative in the middle]

```
1  int[] ns = {2, 3, -1, 4, 5};
2  boolean soFarOnlyPosNums = true;
3  int i = 0;
4  while (i < ns.length) {
5    soFarOnlyPosNums = ns[i] > 0; /* wrong */
6    i = i + 1;
7  }
```

i	soFarOnlyPosNums	i < ns.length	stay?	ns[i]	ns[i] > 0
0	true	true	YES	2	true
1	true	true	YES	3	true
2	true	true	YES	-1	false
3	false	true	YES	4	true
4	true	true	YES	5	true
5	true	false	NO	–	–

Problem: Given an array of numbers, determine if it contains *all* positive number. Also, *for efficiency, exit from the loop as soon as you find a negative number*.

```
1  int[] numbers = {2, 3, -1, 4, 5};
2  boolean soFarOnlyPosNums = true;
3  int i = 0;
4  while (soFarOnlyPosNums && i < numbers.length) {
5    soFarOnlyPosNums = numbers[i] > 0;
6    i = i + 1;
7  }
8  if (soFarOnlyPosNums) { /* print a msg. */ }
9  else { /* print another msg. */ }
```

```
1  int[] ns = {2, 3, -1, 4, 5};
2  boolean soFarOnlyPosNums = true;
3  int i = 0;
4  while (soFarOnlyPosNums && i < ns.length) {
5    soFarOnlyPosNums = ns[i] > 0;
6    i = i + 1;
7  }
```

i	soFarOnlyPosNums	$i <$ ns.length	stay?	ns$[i]$	ns$[i] > 0$
0	true	true	YES	2	true
1	true	true	YES	3	true
2	true	true	YES	-1	false
3	false	true	NO	–	–

```
1  int[] ns = {2, 3, -1, 4, 5};
2  boolean soFarOnlyPosNums = true;
3  int i = 0;
4  while (soFarOnlyPosNums && i < ns.length) {
5    soFarOnlyPosNums = soFarOnlyPosNums && ns[i] > 0;
6    i = i + 1;
7  }
```

i	soFarOnlyPosNums	$i <$ ns.length	stay?	ns$[i]$	ns$[i] > 0$
0	true	true	YES	2	true
1	true	true	YES	3	true
2	true	true	YES	-1	false
3	false	true	NO	–	–

Four possible solutions (`posNumsSoFar` is initialized as *true*):
1. Scan the entire array and accumulate the result.

```
for (int i = 0; i < ns.length; i ++) {
  posNumsSoFar = posNumsSoFar && ns[i] > 0; }
```

2. Scan the entire array but the result is **not** accumulative.

```
for (int i = 0; i < ns.length; i ++) {
  posNumsSoFar = ns[i] > 0; } /* Not working.  Why?  */
```

3. The result is accumulative until the early exit point.

```
for (int i = 0; posNumsSoFar && i < ns.length; i ++) {
  posNumsSoFar = posNumsSoFar && ns[i] > 0; }
```

4. The result is **not** accumulative until the early exit point.

```
for (int i = 0; posNumsSoFar && i < ns.length; i ++) {
  posNumsSoFar = ns[i] > 0; }
```

Problem: Given an array `a` of integers, how do determine if it is sorted in a *non-decreasing* order?

e.g., Given $\{1, 2, 2, 4\}$, print *true*; given $\{2, 4, 3, 3\}$ print *false*.

```
1  boolean isSorted = true;
2  for(int i = 0; i <  a.length - 1 ; i ++) {
3    isSorted = isSorted && (a[i] <= a[i + 1]);
4  }
```

Alternatively (with early exit):

```
1  boolean isSorted = true;
2  for(int i = 0; isSorted && i <  a.length - 1 ; i ++) {
3    isSorted = a[i] <= a[i + 1];
4  }
```

```
1  int[] a = {1, 2, 2, 4}
2  boolean isSorted = true;
3  for(int i = 0; i <  a.length - 1 ; i ++) {
4    isSorted = isSorted && (a[i] <= a[i + 1]);
5  }
```

i	$a[i]$	$a[i + 1]$	$a[i] <= a[i + 1]$	*isSorted*	exit?
0	–	–	–	*true*	N
0	1	2	*true*	*true*	N
1	2	2	*true*	*true*	N
2	2	4	*true*	*true*	**Y**

```
1  int[] a = {2, 4, 3, 3}
2  boolean isSorted = true;
3  for(int i = 0; i <  a.length - 1 ; i ++) {
4    isSorted = isSorted && (a[i] <= a[i + 1]);
5  }
```

i	$a[i]$	$a[i + 1]$	$a[i] <= a[i + 1]$	*isSorted*	exit?
0	–	–	–	*true*	N
0	2	4	*true*	*true*	N
1	4	3	*false*	*false*	N
2	3	3	*true*	*false*	**Y**

```
1  int[] a = {2, 4, 3, 3}
2  boolean isSorted = true;
3  for(int i = 0; isSorted  &&  i <  a.length - 1 ; i ++) {
4    isSorted = a[i] <= a[i + 1];
5  }
```

i	$a[i]$	$a[i + 1]$	$a[i] <= a[i + 1]$	*isSorted*	exit?
0	–	–	–	*true*	N
0	2	4	*true*	*true*	N
1	4	3	*false*	*false*	**Y**

Checking Properties of Arrays (1)

- Determine if **all** elements satisfy a property.
- We need to repeatedly apply the logical **conjunction** .
- **As soon as** we find an element that *does not satisfy* a property, then we exit from the loop.

  e.g., Determine if all elements in array `a` are positive.

```
1  boolean allPos = true;
2  for(int i = 0; i < a.length; i ++) {
3    allPos = allPos && (a[i] > 0);
4  }
```

Alternatively (with early exit):

```
1  boolean allPos = true;
2  for(int i = 0; allPos && i < a.length; i ++) {
3    allPos = a[i] > 0;
4  }
```

Checking Properties of Arrays (2)

- Determine if **at least one** element satisfies a property.
- **As soon as** we find an element that *satisfies* a property, then we exit from the loop.

  e.g., Is there at lease one negative element in array `a`?

  Version 1: Scanner the Entire Array

```
1  boolean foundNegative = false;
2  for(int i = 0; i < a.length; i ++) {
3    foundNegative = foundNegative || a[i] < 0;
4  }
```

Version 2: Possible Early Exit

```
1  boolean foundNegative = false;
2  for(int i = 0; ! foundNegative && i < a.length; i ++) {
3    foundNegative = a[i] < 0;
4  }
```

Checking Properties of Arrays (1): Demo

```
1  int[] a = {2, 3, -1, 4, 5, 6, 8, 9, 100};
2  boolean allPos = true;
3  for(int i = 0; allPos && i < a.length; i ++) {
4    allPos = a[i] > 0;
5  }
```

i	a[i]	a[i] > 0	allPos	exit?
0	–	–	true	N
0	2	true	true	N
1	3	true	true	N
2	-1	false	false	**Y**

- **Question:** Why do we initialize `allPos` as *true* in Line 2?
- **Question:** What if we change the stay condition in Line 3 to only `i < a.length`?

  Intermediate values of `allPos` will be overwritten!

Checking Properties of Arrays (2) Demo

```
1  int[] a = {2, 3, -1, 4, 5, 6, 8, 9, 100};
2  boolean foundNegative = false;
3  for(int i = 0; ! foundNegative && i < a.length; i ++) {
4    foundNegative = a[i] < 0;
5  }
```

i	a[i]	a[i] < 0	foundNegative	!foundNegative	exit?
0	–	–	false	true	N
0	2	false	false	true	N
1	3	false	false	true	N
2	-1	true	true	false	**Y**

- **Question:** Why do we initialize `foundNegative` as *false* in Line 2?

- In some cases, you *must* iterate through the `entire` array in order to obtain the result.
  - e.g., max, min, total, *etc.*
- In other cases, you *exit* from the loop `as soon as` you obtain the result.
  - e.g., to know if all numbers positive, it is certainly *false* `as soon as` you find the first negative number
  - e.g., to know if there is at least one negative number, it is certainly *true* `as soon as` you find the first negative number

---

Arrays: Indexing and Short-Circuit Logic (2)

```
1   Scanner input = new Scanner(System.in);
2   System.out.println("How many integers?");
3   int howMany = input.nextInt();
4   int[] ns = new int[howMany];
5   for(int i = 0; i < howMany; i ++) {
6     System.out.println("Enter an integer");
7     ns[i] = input.nextInt(); }
8   System.out.println("Enter an index:");
9   int i = input.nextInt();
10  if(ns[i] % 2 == 0) {
11    System.out.println("Element at index " + i + " is even."); }
12  else { /* Error : ns[i] is odd */ }
```

- Does the above code work?   [*not* always!]
  - It *works* if `0 <= i && i < ns.length`
  - It *fails* on **L10** if `i < 0 || i >= ns.length`
                              [*ArrayIndexOutOfBoundException*]

Arrays: Indexing and Short-Circuit Logic (1)

Problem: Ask the user how many integers they would like to input, prompt them accordingly, then ask them for an integer index, and check if the number stored at that index is <u>even</u> (i.e., error if it is odd).

```
How many integers?
2
Enter an integer:
23
Enter an integer:
24
Enter an index:
1
24 at index 1 is even.
```

Arrays: Indexing and Short-Circuit Logic (3.1)

```
1   Scanner input = new Scanner(System.in);
2   System.out.println("How many integers?");
3   int howMany = input.nextInt();
4   int[] ns = new int[howMany];
5   for(int i = 0; i < howMany; i ++) {
6     System.out.println("Enter an integer");
7     ns[i] = input.nextInt(); }
8   System.out.println("Enter an index:");
9   int i = input.nextInt();
10  if( 0 <= i && i < ns.length && ns[i] % 2 == 0) {
11    println(ns[i] + " at index " + i + " is even."); }
12  else { /* Error: invalid index or odd ns[i] */ }
```

- Does the above code work?   [*always*!]
- Short-circuit effect of `conjunction` has L-to-R evaluations:
  - `ns[i] % 2 == 0` is evaluated only when the `guard` (i.e., `0 <= i && i < ns.length`) evaluates to *true*.

Arrays: Indexing and Short-Circuit Logic (3.2)

```
1  Scanner input = new Scanner(System.in);
2  System.out.println("How many integers?");
3  int howMany = input.nextInt();
4  int[] ns = new int[howMany];
5  for(int i = 0; i < howMany; i ++) {
6    System.out.println("Enter an integer");
7    ns[i] = input.nextInt(); }
8  System.out.println("Enter an index:");
9  int i = input.nextInt();
10 if( i < 0 || i >= ns.length || ns[i] % 2 == 1) {
11   /* Error: invalid index or odd ns[i] */ }
12 else { println(ns[i] + " at index " + i + " is even."); }
```

- Does the above code work?   [*always*!]
- Short-circuit effect of *disjunction* has L-to-R evaluations:

    ns[i] % 2 == 1 is evaluated only when the *guard*

    (i.e., i < 0 || i >= ns.length) evaluates to *false*.

53 of 75

Arrays: Indexing and Short-Circuit Logic (4)

- ∵ Short-circuit evaluations go from **left** to **right**.
  ∴ **Order** in which the operands are placed matters!
- Consider the following changes to **L10**:
  - ns[i] % 2 == 0 && 0 <= i && i < ns.length
    What if input i is s.t. i < 0?   [*crash*]
    What if input i is s.t. i >= ns.length?   [*crash*]
  - 0 <= i && ns[i] % 2 == 0 && i < ns.length
    What if input i is s.t. i < 0?   [*works*]
    What if input i is s.t. i >= ns.length?   [*crash*]
  - i < ns.length && ns[i] % 2 == 0 && 0 <= i
    What if input i is s.t. i < 0?   [*crash*]
    What if input i is s.t. i >= ns.length?   [*works*]
- When does each change to **L10** *work* and *crash*?
  - ns[i] % 2 == 1 || i < 0 || i >= ns.length
  - i < 0 || ns[i] % 2 == 1 || i >= ns.length
  - i >= ns.length || ns[i] % 2 == 1 || i < 0

54 of 75

Parallel Loops vs. Nested Loops

- *Parallel Loops* :
  Each loop completes an *independent* phase of work.
  e.g., Print an array from left to right, then right to left.

```
System.out.println("Left to right:");
for(int i = 0; i < a.length; i ++) {
  System.out.println(a[i]); }
System.out.println("Right to left:");
for(int i = 0; i < a.length; i ++) {
  System.out.println(a[a.length - i - 1]); }
```

- *Nested Loops* :
  Loop counters form *all combinations* of indices.

```
for(int i = 0; i < a.length; i ++) {
  for(int j = 0; j < a.length; j ++) {
    System.out.println("(" + i + ", " + j + ")");
  } }
```

55 of 75

Nested Loops: Finding Duplicates (1)

- Given an integer array a, determine if it contains any duplicates.
  e.g., Print *false* for {1, 2, 3, 4}. Print *true* for {1, 4, 2, 4}.
- **Hint:** When can you conclude that there are duplicates?
  As soon as we find that two elements at difference indices happen to be the same

```
1  boolean hasDup = false;
2  for(int i = 0; i < a.length; i ++) {
3    for(int j = 0; j < a.length; j ++) {
4      hasDup = hasDup || (i != j && a[i] == a[j]);
5    } /* end inner for */ } /* end outer for */
6  System.out.println(hasDup);
```

- **Question:** How do you modify the code, so that we exit from the loops *as soon as* the array is found containing duplicates?
  - **L2**: for(...; !hasDup && i < a.length; ...)
  - **L3**: for(...; !hasDup && j < a.length; ...)
  - **L4**: hasDup = (i != j && a[i] == a[j]);

56 of 75

Nested Loops: Finding Duplicates (2)

```
1  /* Version 1 with redundant scan */
2  int[] a = {1, 2, 3}; /* no duplicates */
3  boolean hasDup = false;
4  for(int i = 0; i < a.length; i ++) {
5    for(int j = 0; j < a.length; j ++) {
6      hasDup = hasDup || (i != j && a[i] == a[j]);
7    } /* end inner for */ } /* end outer for */
8  System.out.println(hasDup);
```

i	j	i != j	a[i]	a[j]	a[i] == a[j]	hasDup
0	0	false	1	1	true	false
0	1	true	1	2	false	false
0	2	true	1	3	false	false
1	0	true	2	1	false	false
1	1	false	2	2	true	false
1	2	true	2	3	false	false
2	0	true	3	1	false	false
2	1	true	3	2	false	false
2	2	false	3	3	true	false

Nested Loops: Finding Duplicates (3)

```
1  /* Version 1 with redundant scan and no early exit */
2  int[] a = {4, 2, 4}; /* duplicates: a[0] and a[2] */
3  boolean hasDup = false;
4  for(int i = 0; i < a.length; i ++) {
5    for(int j = 0; j < a.length; j ++) {
6      hasDup = hasDup || (i != j && a[i] == a[j]);
7    } /* end inner for */ } /* end outer for */
8  System.out.println(hasDup);
```

i	j	i != j	a[i]	a[j]	a[i] == a[j]	hasDup
0	0	false	4	4	true	false
0	1	true	4	2	false	false
0	2	true	4	4	true	true
1	0	true	2	4	false	true
1	1	false	2	2	true	true
1	2	true	2	4	false	true
2	0	true	4	4	true	true
2	1	true	4	2	false	true
2	2	false	4	4	true	true

Nested Loops: Finding Duplicates (4)

```
1  /* Version 2 with redundant scan */
2  int[] a = {1, 2, 3}; /* no duplicates */
3  boolean hasDup = false;
4  for(int i = 0; i < a.length && !hasDup ; i ++) {
5    for(int j = 0; j < a.length && !hasDup ; j ++) {
6      hasDup = i != j && a[i] == a[j] ;
7    } /* end inner for */ } /* end outer for */
8  System.out.println(hasDup);
```

i	j	i != j	a[i]	a[j]	a[i] == a[j]	hasDup
0	0	false	1	1	true	false
0	1	true	1	2	false	false
0	2	true	1	3	false	false
1	0	true	2	1	false	false
1	1	false	2	2	true	false
1	2	true	2	3	false	false
2	0	true	3	1	false	false
2	1	true	3	2	false	false
2	2	false	3	3	true	false

Nested Loops: Finding Duplicates (5)

```
1  /* Version 2 with redundant scan and early exit */
2  int[] a = {4, 2, 4}; /* duplicates: a[0] and a[2] */
3  boolean hasDup = false;
4  for(int i = 0; i < a.length && !hasDup ; i ++) {
5    for(int j = 0; j < a.length && !hasDup ; j ++) {
6      hasDup = i != j && a[i] == a[j] ;
7    } /* end inner for */ } /* end outer for */
8  System.out.println(hasDup);
```

i	j	i != j	a[i]	a[j]	a[i] == a[j]	hasDup
0	0	false	4	4	true	false
0	1	true	4	2	false	false
0	2	true	4	4	true	true

Nested Loops: Finding Duplicates (6)

The previous two versions scan all pairs of array slots, but with redundancy: e.g., `a[0] == a[2]` and `a[2] == a[0]`.

```
1  /* Version 3 with no redundant scan */
2  int[] a = {1, 2, 3, 4}; /* no duplicates */
3  boolean hasDup = false;
4  for(int i = 0; i < a.length && !hasDup; i ++) {
5    for(int j = i + 1; j < a.length && !hasDup; j ++) {
6      hasDup = a[i] == a[j];
7    } /* end inner for */ } /* end outer for */
8  System.out.println(hasDup);
```

i	j	a[i]	a[j]	a[i] == a[j]	hasDup
0	1	1	2	false	false
0	2	1	3	false	false
0	3	1	4	false	false
1	2	2	3	false	false
1	3	2	4	false	false
2	3	3	4	false	false

Nested Loops: Finding Duplicates (7)

```
1   /* Version 3 with no redundant scan:
2    * array with duplicates causes early exit
3    */
4   int[] a = {1, 2, 3, 2}; /* duplicates: a[1] and a[3] */
5   boolean hasDup = false;
6   for(int i = 0; i < a.length && !hasDup; i ++) {
7     for(int j = i + 1; j < a.length && !hasDup; j ++) {
8       hasDup = a[i] == a[j];
9     } /* end inner for */ } /* end outer for */
10  System.out.println(hasDup);
```

i	j	a[i]	a[j]	a[i] == a[j]	hasDup
0	1	1	2	false	false
0	2	1	3	false	false
0	3	1	2	false	false
1	2	2	3	false	false
1	3	2	2	true	true

Common Error (1): Improper Initialization of Loop Counter

```
boolean userWantsToContinue;
while (userWantsToContinue) {
  /* some computations here */
  String answer = input.nextLine();
  userWantsToContinue = answer.equals("Y");
}
```

The default value for an initialized boolean variable is *false*.
Fix?

```
boolean userWantsToContinue = true;
while (userWantsToContinue) {
  /* some computations here */
  String answer = input.nextLine();
  userWantsToContinue = answer.equals("Y");
}
```

Common Error (2): Improper Stay Condition

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

The maximum index for array `a` is `a.length − 1`
Fix?

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

Common Error (3): Improper Update to Loop Counter

Does the following loop print all slots of array `a`?

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

The indices used to print will be: 1, 2, 3, ..., a.length

Fix?

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

```
int i = 0;
while (i < a.length) {
  i ++;
  System.out.println(a[i - 1]);
}
```

65 of 75

Common Error (5): Improper Start Value of Loop Counter

```
int i = a.length - 1;
while (i >= 0) {
  System.out.println(a[i]); i --; }
while (i < a.length) {
  System.out.println(a[i]); i ++; }
```

The value of loop counter *i* after the first `while` loop is −1!

Fix?

```
int i = a.length - 1;
while (i >= 0) {
  System.out.println(a[i]); i --; }
i = 0;
while (i < a.length) {
  System.out.println(a[i]); i ++; }
```

67 of 75

Common Error (4): Improper Update of Stay Condition

```
1  String answer = input.nextLine();
2  boolean userWantsToContinue = answer.equals("Y");
3  while (userWantsToContinue) { /* stay condition (SC) */
4    /* some computations here */
5    answer = input.nextLine();
6  }
```

What if the user's answer in **L1** is simply `Y`?   An *infinite loop*!!

∵ **SC** never gets updated when a new answer is read.   Fix?

```
String answer = input.nextLine();
boolean userWantsToContinue = answer.equals("Y");
while (userWantsToContinue) {
  /* some computations here */
  answer = input.nextLine();
  userWantsToContinue = answer.equals("Y");
}
```

66 of 75

Common Error (6): Wrong Syntax

How about this?

```
while(int i = 0; i < 10; i ++) { ... }
```

You meant:

```
for(int i = 0; i < 10; i ++) { ... }
```

How about this?

```
for(i < 10) { ... }
```

You meant:

```
while(i < 10) { ... }
```

or

```
for( ; i < 10 ; ) { ... }
```

68 of 75

Common Error (7): Misplaced Semicolon

Semicolon (`;`) in Java marks *the end of a statement* (e.g., assignment, `if` statement, `for`, `while`).

```java
int[] ia = {1, 2, 3, 4};
for (int i = 0; i < 10; i ++); {
  System.out.println("Hello!");
}
```

Output?

```
Hello!
```

Fix?

```java
for (int i = 0; i < 10; i ++) {
  System.out.println("Hello!");
}
```

Index (2)

Index (1)

Index (3)

Index (4)

Index (5)

Index (6)