Click to add text

LOOP TRICKS AND PITFALLS

Local Variable Inside `for`

- Possible to declare variables within a `for` statement

```
int sum = 0;
for ( int n = 1 ; n <= 10 ; n++ )
    sum = sum + n * n;
```

- Note that variable `n` is local to the loop, it means:
  - you can not use `n` after the loop.
  - `n` is undeclared outside the loop.

for Statement Variations

- Multiple Initializations are allowed:

```
for ( n = 1 , product = 1 ; n <= 10; n++ )
     product = product * n;
```

Use a comma , to separate multiple init. or actions

- Multiple update actions are allowed:

```
for ( n = 1, product = 1; n <= 10;
        product = product * n , n++ ) ;
```

for Statement Variations and Pitfalls

- A for loop ending with a ; does not have a body:

```
for (i = 1; i <= 5; i++);
{    System.out.println("Hello");
     System.out.println("*");
}
System.out.println(i);
```

- it is most likely a logical error,
  - except like example on previous slide:

```
for ( n = 1, product = 1; n <= 10;
     product = product * n , n++ );
```

`for` Statement Variations

- Only **one** boolean expression is allowed, but it can consist of `&&`s, `||`s, and `!`s.
- If **no** boolean expression is given, it is assumed to be `true`

```
for ( n = 1 ;   ; n++ )
{   sum = sum + n;
    if (n > 10) break;
}
```

- Omitting all three control statements ➔ infinite loop

```
for ( ;   ; )
{   sum = sum + n;
    if (n > 10) break;
}
```

The Loop Body

- To design the loop body, write out the actions the code must accomplish.
- Then look for a repeated pattern.
  - The pattern need not start with the first action.
  - The repeated pattern will form the body of the loop.
  - Some actions may need to be done after the pattern stops repeating.

Initializing Statements

- Some variables need to have a value before the loop begins.
  - Sometimes this is determined by what is supposed to happen after one loop iteration.
  - Often variables have an initial value of zero or one, but not always.
- Other variables get values only while the loop is iterating.

Controlling Number of Loop Iterations

- If the number of iterations is known before the loop starts, the loop is called a count-controlled loop*.*
  - Use a **`for`** loop.
- Asking the user before each iteration if it is time to end the loop is called the ask-before-iterating technique.
  - Appropriate for a small number of iterations
  - Use a **`while`** loop or a **`do-while`** loop.

Controlling Number of Loop Iterations

- For large input lists, a sentinel value can be used to signal the end of the list.
  - The sentinel value must be different from all the other possible inputs.
  - A negative number following a long list of nonnegative exam scores could be suitable.

```
90
0
10
-1
```

Loop control

```
int N = 5;    //input by user
               //or assigned
int counter = 0;
int sum = 0;
while (counter < N)
{



    counter++;
    sum = sum + counter;

}
System.out.println("Total= ",sum);
```

What if we put a ; here?

Infinite loop

What's the output?

Total = 15

What if we use `(counter<=N)` ?

Total = 21

infinite loop

What if we use `counter--;` ?

What if we switch the order of these two lines?

Total = 10

Loop control

```
int N = 5;    //input by user
               //or assigned
int counter = 1;
int sum = 0;
while (counter <= N)
{



    counter++;
    sum = sum + counter;

}
System.out.println(“Total= ”,sum);
```

Trace it at home
☺

Beware of infinite loops

- What about this loop?

```
for (i = 0;  i <= 5;  i++)
{
    System.out.println(i--);
}
System.out.println();
```

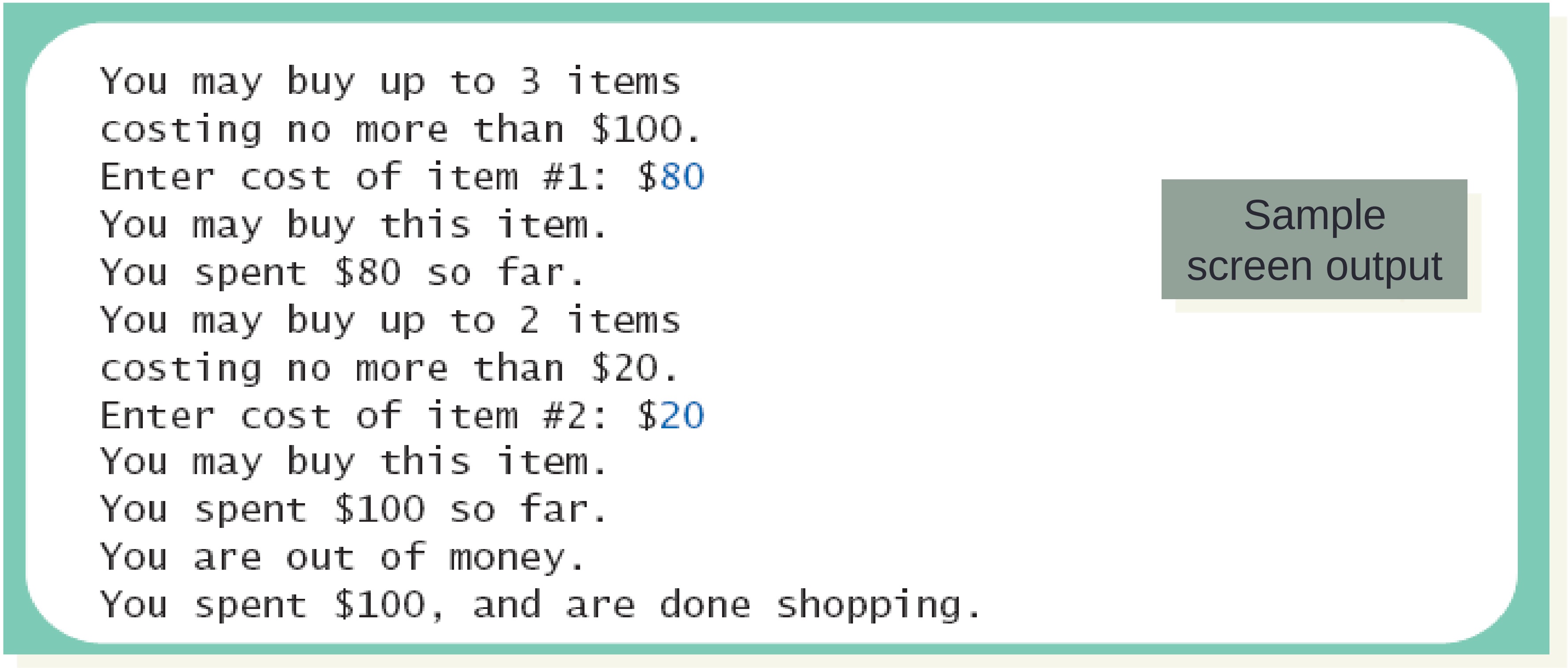
Tip: Avoid changing the loop counter inside the body

Programming Example

- Spending Spree
  - You have $100 to spend in a store
  - Maximum 3 items
  - Computer tracks spending and item count
  - When item chosen, computer tells you whether or not you can buy it
- Client wants adaptable program
  - Able to change amount and maximum number of items
- View sample algorithm

Programming Example

- View sample program, listing 4.7
  `class SpendingSpree`



```
You may buy up to 3 items
costing no more than $100.
Enter cost of item #1: $80
You may buy this item.
You spent $80 so far.
You may buy up to 2 items
costing no more than $20.
Enter cost of item #2: $20
You may buy this item.
You spent $100 so far.
You are out of money.
You spent $100, and are done shopping.
```

Sample screen output

The `break` and `continue` Statement

`break`

- Used to **exit a loop completely**.
- Skips remainder of loop body and continues AFTER the loop
- Ends only the **innermost** loop that contains the `break`.

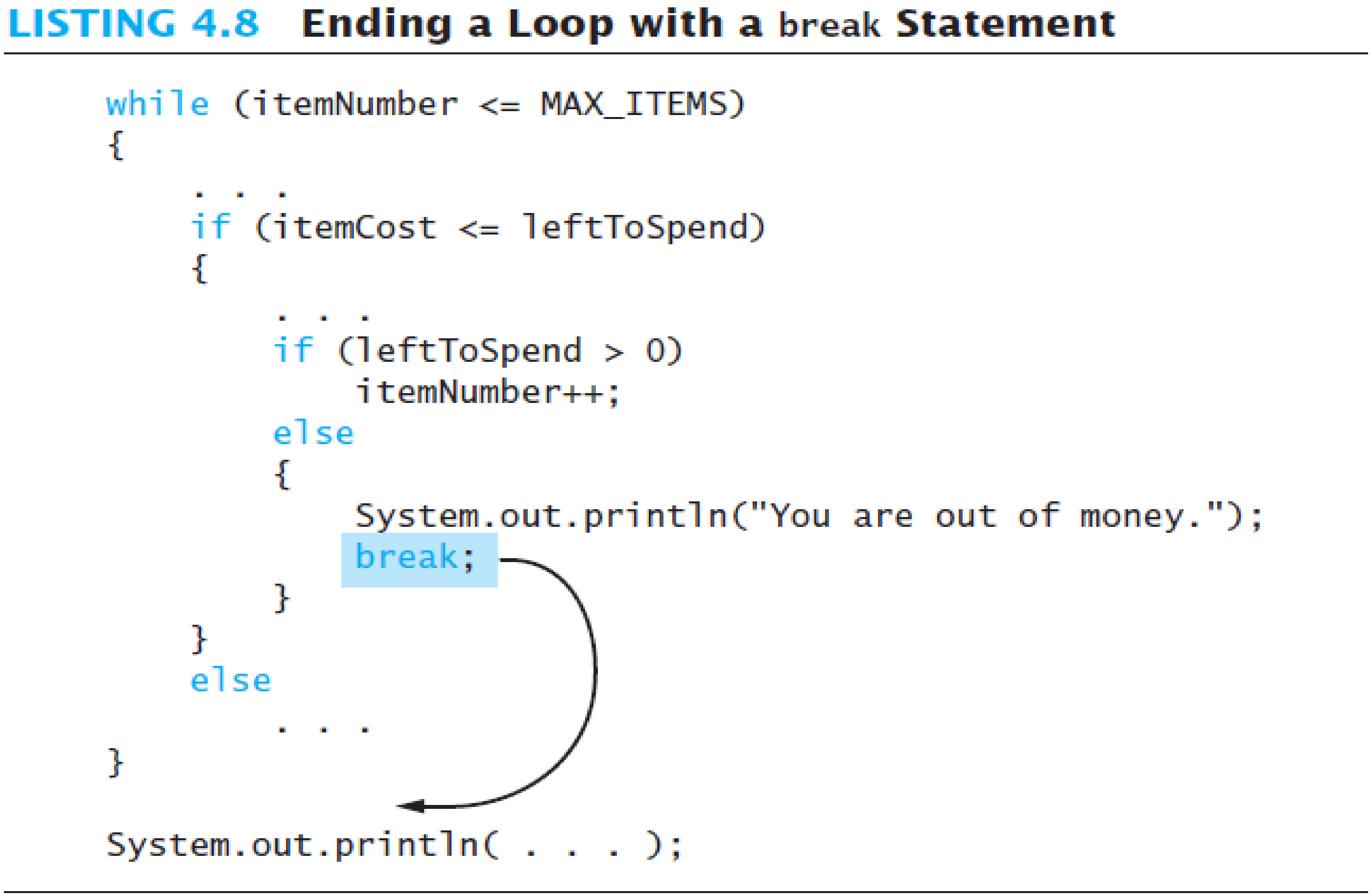
`continue`

- Used to **end current iteration only**.
- Skips remainder of loop body and continues with NEXT iteration if any:
  - In a `while` and `do…while`, the logical expression is evaluated immediately after `continue`
  - In a `for` statement, the counter is updated after the `continue` and then the logical expression is evaluated.

The `break` and `continue` Statement

- BOTH are associated with an `if` statement.
- BOTH affect only the **innermost** loop that contains the `break` or `continue`
- BOTH make loops more difficult to understand ➔ Avoid using them.

The **break** Statement in Loops



LISTING 4.8 Ending a Loop with a break Statement

```
while (itemNumber <= MAX_ITEMS)
{
    . . .
    if (itemCost <= leftToSpend)
    {
        . . .
        if (leftToSpend > 0)
            itemNumber++;
        else
        {
            System.out.println("You are out of money.");
            break;
        }
    }
    else
        . . .
}

System.out.println( . . . );
```

The `break` Statement in Loops

- The following code segment **sums up a set of positive** integers
- This program does **not** allow to sum **negative** numbers:

```
int num = -1, sum = 0; //initialize the accumulator sum
while (num != 0)
    {
      System.out.println (“Enter a positive integer, 0 to exit”);
      num = read.nextInt();
      if (num < 0)
         {
           System.out.println (“Negative numbers are not allowed”);
           break;
         } //end if
      sum =sum + num;
    } // end while
System.out.println (“Sum = “ + sum);
```

If the user enters a NON-ZERO POSITIVE number

The POSITIVE number is added

The `break` Statement in Loops

- The following code segment **sums up a set of positive** inte
- This program does **not** allow to sum **negative** numbers:

If the user enters a NEGATIVE number

```
int num = -1, sum = 0; //initialize the accumul   r sum
while (num != 0)
    {
       System.out.println (“Enter a positive integer, 0 to exit”);
       num = read.nextInt();
       if (num < 0)
          {
             System.out.println (“Negative numbers are not allowed”);
             break;
          } //end if
       sum =sum + num;
    } // end while
System.out.println (“Sum = “ + sum);
```

The –ve number is NOT added, because `break;` exits the loop

The `continue` Statement in Loop

- The following code segment **sums up a set of positive inte**
- **Negative** numbers are **skipped**, but the loop does **not st**

```
int num = -1, sum = 0; //initialize the accumul    r sum
while (num != 0)
    {
        System.out.println (“Enter a positive integer, 0 to exit”);
        num = read.nextInt();
        if (num < 0)
            {
                System.out.println (“Negative numbers are not allowed”);
                continue;
            } //end if
        sum =sum + num;
    } // end while
System.out.println (“Sum = “ + sum);
```

If the user enters a NON-ZERO POSITIVE number

The POSITIVE number is added

The `continue` Statement in Loops

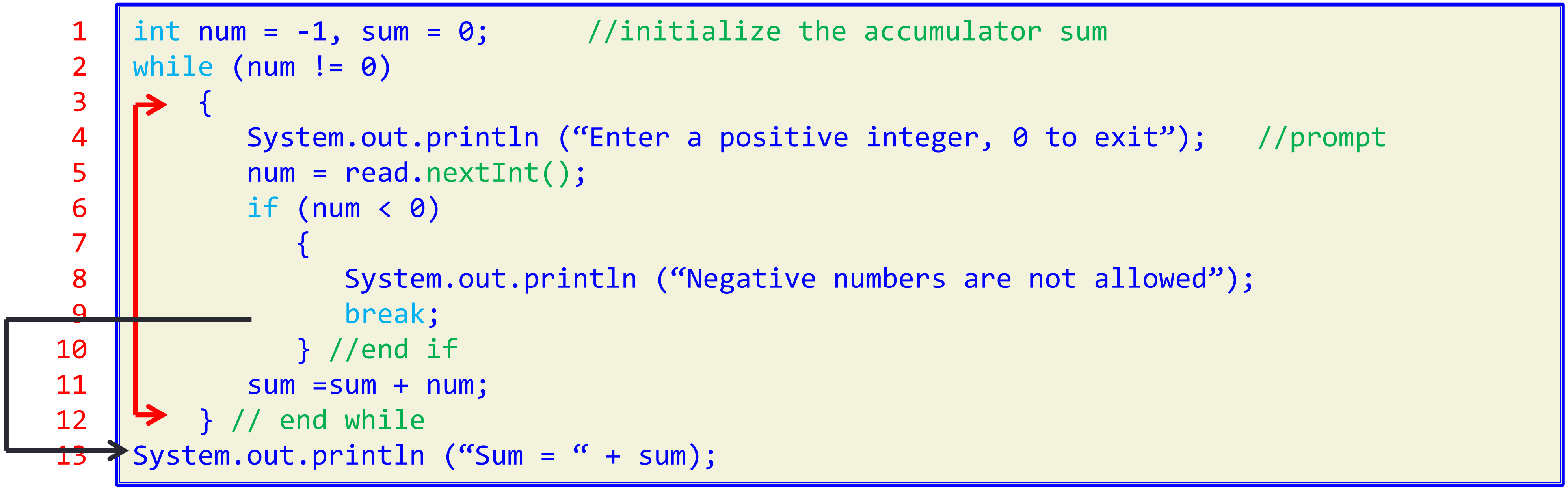
- The following code segment **sums up a set of positive** inte
- This program does **not** allow to sum **negative** numbers:

If the user enters a NEGATIVE number

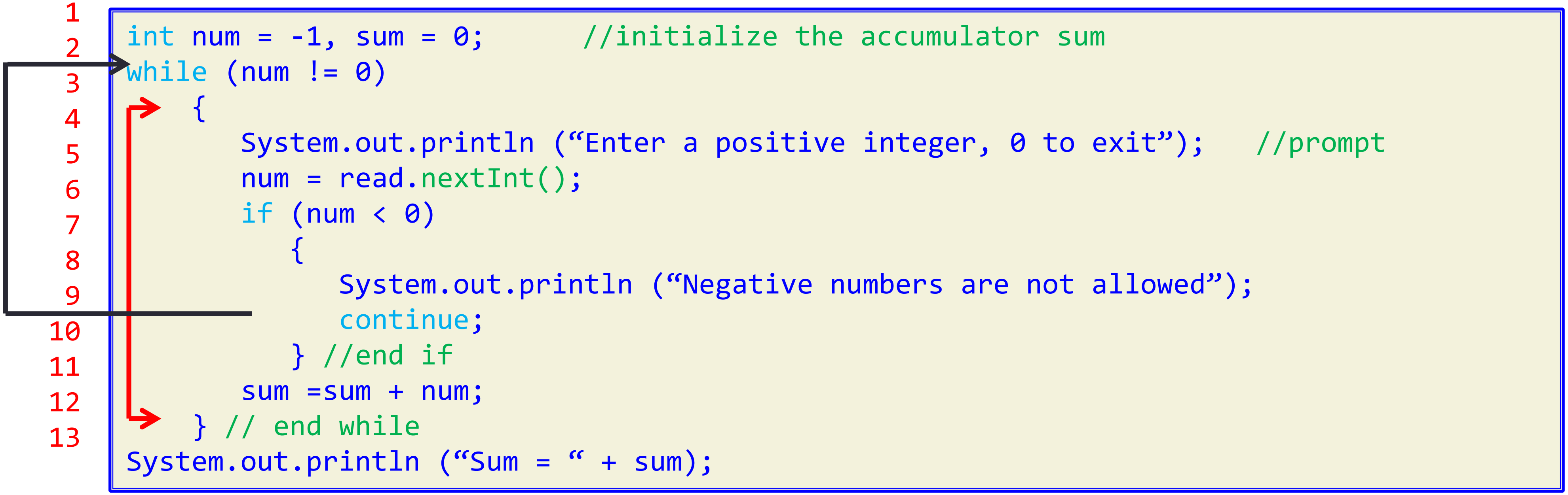
```java
int num = -1, sum = 0; //initialize the accumulator sum
while (num != 0)
    {
        System.out.println (“Enter a positive integer, 0 to exit”);
        num = read.nextInt();
        if (num < 0)
            {
                System.out.println (“Negative numbers are not allowed”);
                continue;
            } //end if
        sum =sum + num;
    } // end while
System.out.println (“Sum = “ + sum);
```

The –ve number is NOT added, because `continue;` exits the current iteration

break vs continue Statement

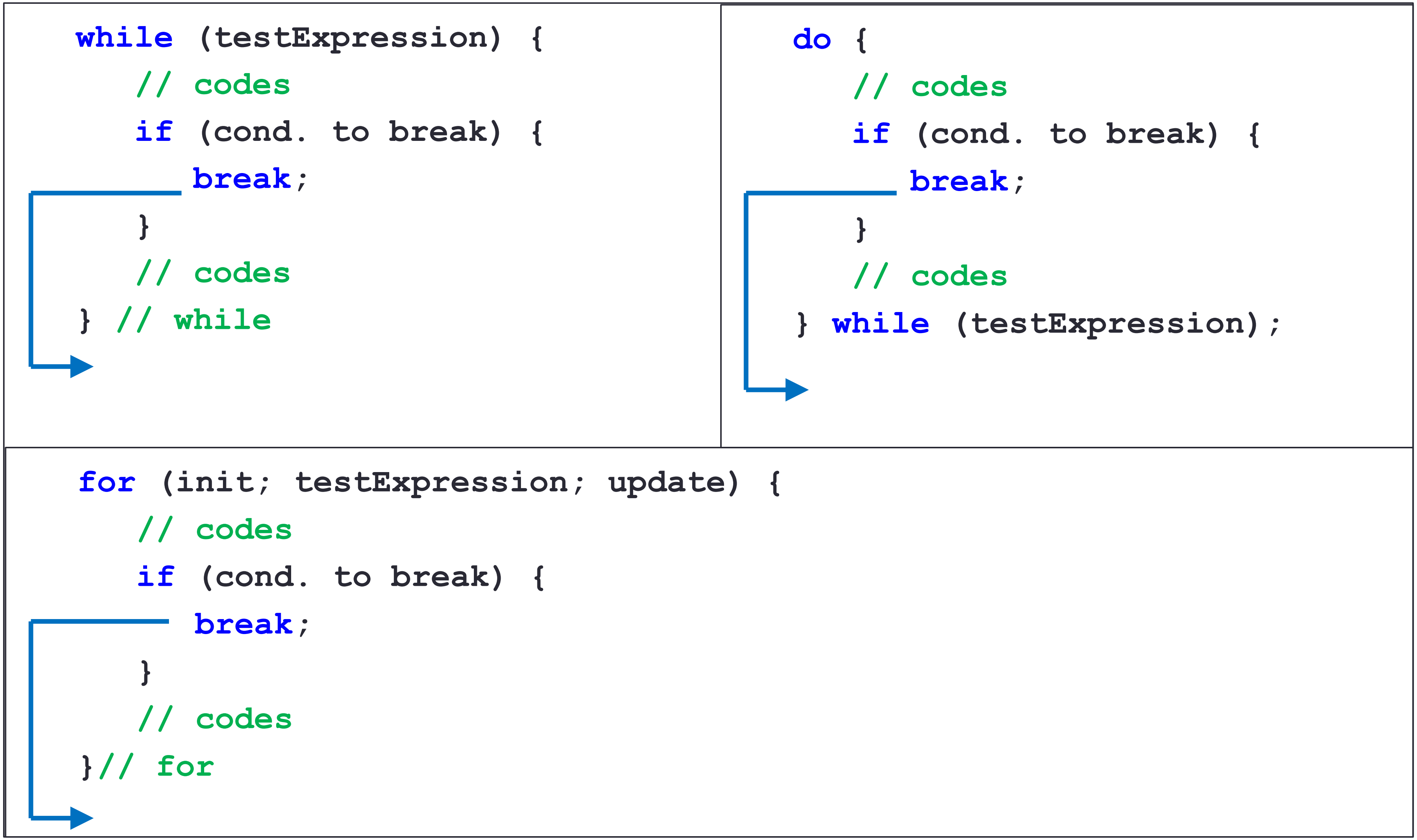


```
int num = -1, sum = 0;      //initialize the accumulator sum
while (num != 0)
    {
       System.out.println (“Enter a positive integer, 0 to exit”);   //prompt
       num = read.nextInt();
       if (num < 0)
          {
             System.out.println (“Negative numbers are not allowed”);
             break;
          } //end if
       sum =sum + num;
    } // end while
System.out.println (“Sum = “ + sum);
```



```
int num = -1, sum = 0;      //initialize the accumulator sum
while (num != 0)
    {
       System.out.println (“Enter a positive integer, 0 to exit”);   //prompt
       num = read.nextInt();
       if (num < 0)
          {
             System.out.println (“Negative numbers are not allowed”);
             continue;
          } //end if
       sum =sum + num;
    } // end while
System.out.println (“Sum = “ + sum);
```

How the **break** Statement works

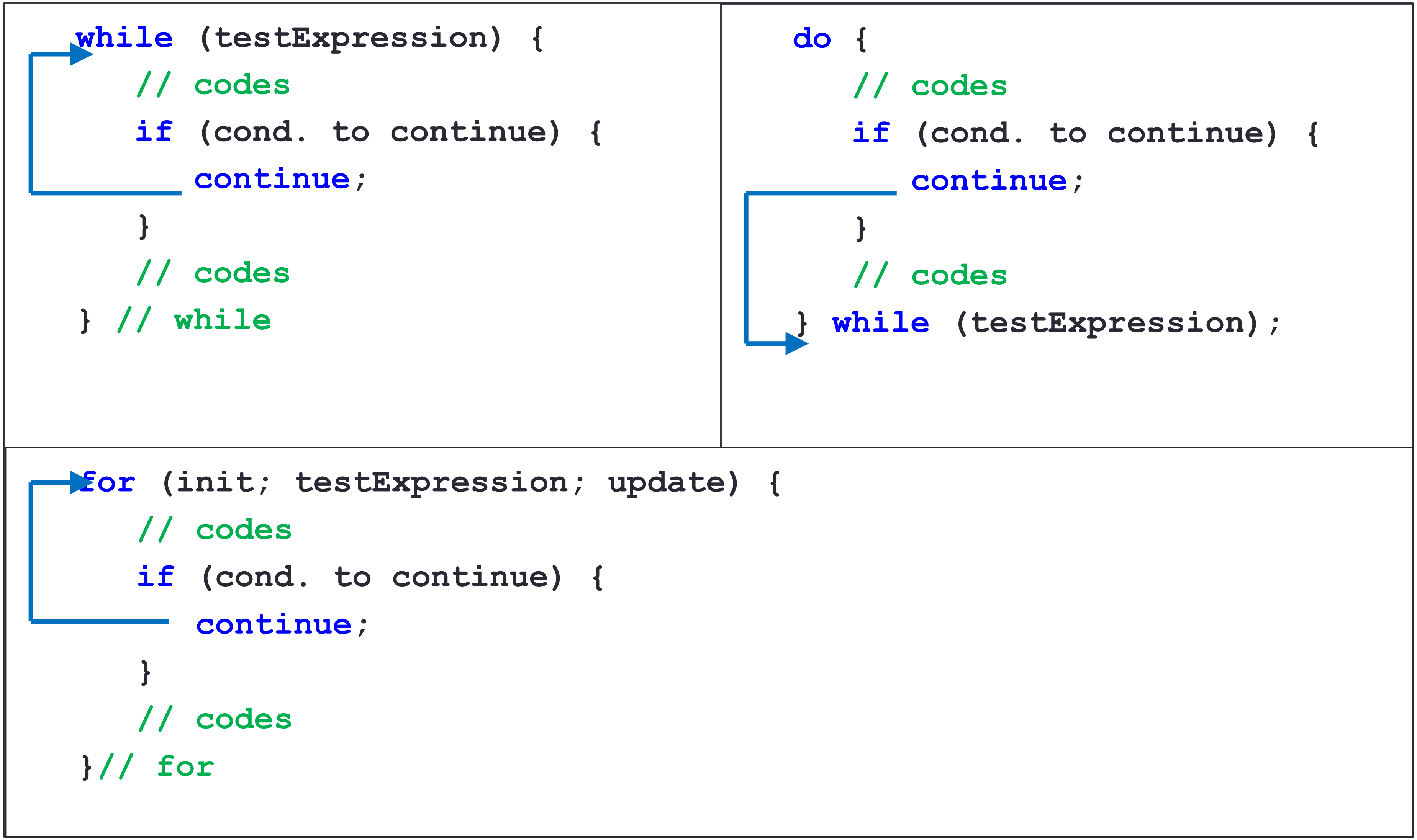


```
while (testExpression) {
   // codes
   if (cond. to break) {
      break;
   }
   // codes
} // while
```

```
do {
   // codes
   if (cond. to break) {
      break;
   }
   // codes
} while (testExpression);
```

```
for (init; testExpression; update) {
   // codes
   if (cond. to break) {
      break;
   }
   // codes
}// for
```

How the continue Statement works



```
while (testExpression) {
    // codes
    if (cond. to continue) {
       continue;
    }
    // codes
} // while
```

```
do {
    // codes
    if (cond. to continue) {
       continue;
    }
    // codes
} while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (cond. to continue) {
       continue;
    }
    // codes
}// for
```

Tracing Variables

- Tracing variables means watching the variables change while the program is running.
  - Simply insert temporary output statements in your program to print of the values of variables of interest
  - Or, learn to use the debugging facility that may be provided by your system.

Loop Bugs

- Common loop bugs
  - Unintended infinite loops
  - Off-by-one errors
  - Testing equality of floating-point numbers
- Subtle infinite loops
  - The loop may terminate for some input values, but not for others.
  - For example, you can’t get out of debt when the monthly penalty exceeds the monthly payment.