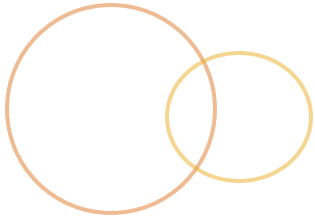
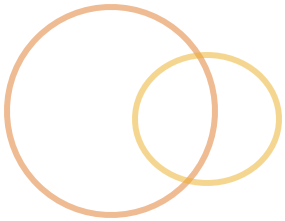
Control Structures in Java





Objectives

At the end of this module, you should be able to

- Describe what statements and blocks are
- Describe what a local variable is and its scope
- Describe the flow of control of a Java program
- Use `if` statements, `switch` statements

Statements and Blocks

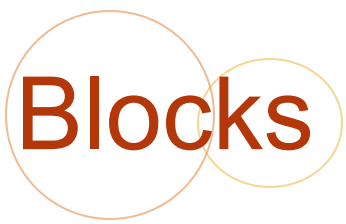
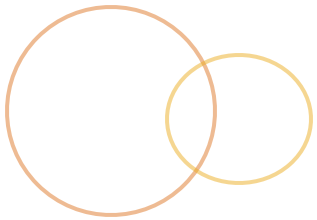
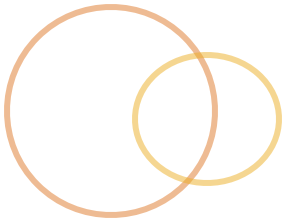
- Program execution is controlled by statements
  - Statements can extend over any number of lines
  - Are terminated by a semi-colon (`;`)
  - Executable statements *only* exist inside blocks (typically methods)
- Statements are often made up of expressions, but not always
  - Expressions evaluate to a result, but statements don't have to –
  - The `while` loop is a statement that is not an expression
- A statement can also be an expression
  - Called an expression statement
  - The result of the expression is discarded
  - e.g., the statement `x = 3;` returns and discards the value `3`

Syntax of Statements

Example 3-1: Expressions and statements

```
x = 7 + 1 // expression – not a statement
x = 7 + 1; // adding a semi-colon makes this a statement.
int x;     // statement – not an expression
x++;       // expression and a statement

// a statement and complex expression
int y = x++ / ( z * 2.0);

// the empty statement – legal but pointless.
;

// following: a line containing three statements
x = 7 + 1; x++; System.out.println(x);

// following is one statement on three lines
x =
    7 +
        1;
```

Blocks


- A block is a sequence of statements enclosed in braces { }
- A block can occur anywhere in a Java program that a statement can
- Any statement in the block could itself be replaced by a block
- Nested blocks are allowed

Blocks and Nested Blocks Example

Example 3-2: Blocks and nested blocks

```
public class Ex4_2  {
  public static void main(String []args) {
    int x = 0;
   x++;
    int y = 0;
    { // Start of a user defined block
      { // Start of a nested user defined block
        System.out.println(x);
        System.out.println(y);
      } // End of a nested user defined block
      y = x % 3;
    } // End of the outer user defined block
    y++;
    {  // Start of a second user defined block
      System.out.println(x);
      System.out.println(y);
    } // End of the second user defined block
  } // End of the block that makes up the body of the main method
} // end of the block that makes up the class definition
```

Local Variables

- Local variables exist within the scope of the defining block
  - Are **not** automatically initialized
  - Used as temporary or “working” variables within the body of a block
- Local variables have a set lifetime, they
  - Come into existence when the flow of control passes through their declaration
  - Cease to exist when the flow of control passes out the defining block

Scope of a Local Variable Example I

Example 3-4: Scope of local variable `outerVar`

```
// scope of outerVar
void someMethod(int x) {
  System.out.println("Entering someMethod..");
  int outerVar = 1; //local to someMethod
  { //"inner block"
    System.out.println("Entering inner block…");
    int innerVar = 4; //local to "inner block"
    innerVar = outerVar + x;
    System.out.println("innerVar is "+ innerVar);
  }
  System.out.println("outerVar is "+ outerVar);
}
```

Scope of a Local Variable Example II

Example 3-5: Scope of local variable `innerVar`

```
// scope of innerVar
void someMethod(int x) {
  System.out.println("Entering someMethod..");
  int outerVar = 1; //local to someMethod
  { //”inner block”
    System.out.println("Entering inner block…");
    int innerVar = 4; //local to “inner block”
    innerVar = outerVar + x;
    System.out.println("innerVar is "+ innerVar);
  }
  System.out.println("outerVar is "+ outerVar);

}
```

Common Local Variable Errors Example

Example 3-6: Common local variable errors

```
// scope of innerVar
void someMethod(int x) {
  System.out.println("Entering someMethod..");
  int outerVar = 1;
  {
    System.out.println("Entering inner block…");
    // Error 1: referencing innerVar before it is declared
    System.out.println(innerVar);
    int innerVar = 4;
    innerVar = outerVar + x;
    System.out.println("innerVar is "+ innerVar);
    // Error 2: trying to declare a variable with a name
    // used by another local variable in the same scope
    int outerVar = 10;
  }
  System.out.println("outerVar is "+ outerVar);
  // Error 3: Trying to reference innerVar outside of its scope
  System.out.println(innerVar);
}
```

Flow Control

Basic if Statement Syntax

- The `if` statement looks like:

```
if (test-expression){ }
```

- The test-expression is any expression that evaluates to one of the `boolean` values `true` or `false`
  - The expression must be contained in parentheses
  - The body of the `if` statement can be either a single statement or a block
  - If the body is a single statement, it must be terminated by a semi-colon

Basic if Statement Syntax Example

Example 3-7: Examples of if-then

```
// if statement with a single statement as a then-clause,
if (x == 3)
  System.out.println("x is, in fact, three");
x = 24;

// if statement with a body as a then-clause
boolean test = (x > 23);
if (test) {
  System.out.println("x is out of range");
  x = x - 10;
  System.out.println("value of x is reset to" + x);
}
x = 24;
```

if-else Statements

- The `if-else` form allows two mutually exclusive paths of execution
- The `if-else` form of the `if` statement looks like

```
if (test-expression) {
  then-clause
}
else {
  else-clause
}
```

- If the test expression is `true`, the then-clause executes exactly as we just saw in the previous section
  - If the test condition is `false`, the else-clause executes instead
  - Since the test-expression is `boolean`, one of the clauses will always execute

if-else Statements Example

Example 3-8: Examples of if-then-else

```
// if statement with a single statements in both then and else clauses.
if (x == 3)
  System.out.println("x is, in fact, three");
else
  System.out.println("x is NOT, in fact, three");
// if statement with a body in both then and else clauses
if (x > 23) {
  System.out.println("x is out of range");
  x = x - 10;
  System.out.println("value of x is reset to" + x);
}
else {
  System.out.println("x is in range");
  x++;
}
```

if-else Statements Example (cont.)

Example 3-8: Examples of if-then-else (continued)

```
// if statement with a body in then and a statement else clauses
if (x > 23) {
  System.out.println("x is out of range");
  x = x - 10;
  System.out.println("value of x is reset to" + x);
}
else
  System.out.println("x is in range");

// if statement with a statement in then and a body else clauses
if (test)
  System.out.println("x is out of range");
else {
  System.out.println("x is in range");
  x++;
}
```

The Dangling else Problem

Example 3-10: Dangling else

```
if (test1)
  if (test2)
    System.out.println("test1 and test2 true);
else
  System.out.println("test1 if false");

// what the programmer meant was
if (test1) {
  if (test2)
    System.out.println("test1 and test2 true);
}
else
  System.out.println("test1 if false");

// what the compiler saw was
if (test1) {
  if (test2)
    System.out.println("test1 and test2 true);
  else
    System.out.println("test1 if false");
}
```

if-elseif-else Statements

Example 3-11: Nested conditional -- multiple test values

```
int status = getStatus();
if (status == 0) {
  /* stuff to do if status is 0 */
}
else {
  if (status == 1) {
    /* stuff to do if status is 1 */
  }
  else {
    if (status == 2) {
      /* stuff to do if status is 2 */
    }
    else {
      /* stuff to do if status is anything else */
    }
  }
}
```

if-elseif-else Statements

- Most programming languages provide an alternate form to nested if statements
- In Java the syntax is

The if-else if-else construct

```
if (test1) first-then-clause
else if (test2) second-then-clause
else if (test3) third-then-clause
else else-clause /* the else clause is optional */
```

if-else if-else Example

Example 3-12: if-else if-else for Example 3-11

```
int status = getStatus();
if (status == 0) {
  /* stuff to do if status is 0 */
}
else if (status == 1) {
  /* stuff to do if status is 1 */
}
else if (status == 2) {
  /* stuff to do if status is 2 */
}
else {
  /* stuff to do if status is anything else */
}
```

The switch Statement

- Sometimes, it is necessary to perform conditional behavior based on the differing numeric values

```
if(x == 2) //do something
else if(x == 3) // do something else
else if(x == 4) //do something else
else //do something default
```

- Using if, else if, else can be
  - Tedious (especially if you have many conditions)
  - Cause unwanted overhead (every each condition is evaluated until the right one is found)
- There is a control construct that helps with this – `switch`
- The `switch` construct only works on integers and characters

The switch Statement

The switch construct

```
// testvar is a variable of some type.
switch(testvar) {
  case value1:
    /* code to execute when testvar has the value value1 */
   break;
  case value2:
    /* code to execute when testvar has the value value2 */
   break;
  case value3:
    /* code to execute when testvar has the value value3 */
   break;

  /*--- more cases --*/
  case value_n:
    /* code to execute when testvar has the value value_n */
   break;
  default:
    /* code to execute when testvar none of the above values */
    break;
}
```

The switch Statement Example

Example 3-13: switch case example

```
char status = 'a';

switch (status) {
  case '*':
    System.out.println("Asterisk");
    break;
  case 'a':
    System.out.println("letter a");
    break;
  case 'z':
    System.out.println("letter z");
    break;
  default:
    System.out.println("Unrecognized character");
    break;
}
```

The switch Statement (cont.)

Example 3-13: switch case example (continued)

```
System.out.println("And continuing…");

status = 'g';
switch (status) {
  case '*':
    System.out.println("Asterisk");
    break;
  case 'a':
    System.out.println("letter a");
    break;
  case 'z':
    System.out.println("letter z");
    break;
  default:
    System.out.println("Unrecognized character");
    break;
}
System.out.println("And continuing…");
```

The switch Statement

- Ordering of `case` statements is entirely up to the programmer
  - For example, the `default` case does not have to be the last `case` statement
  - For example, the order of `case` does not have to follow the logical ordering of integers
  - The `default` case is optional
- Execution will continue into the next `case` statement if a `break` statement is not encountered
  - This called *falling-through*
  - Fall through behavior allows the same code to applied to a set of test cases
- Applying falling-through with fancy you can achieve some advanced solutions

switch Statement Fall-through Example

Example 3-14: switch case fall through

```
char status = 'z';
switch (status) {
  case '*':
  case 'a':
    System.out.println("letter a or an asterisk");
    break;
  case 'z':
    System.out.println("letter z");
  case 'w':
    System.print("letter w");
    break;
  default:
    System.out.println("Unrecognized character");
    break;
}
```

The Ternary Operator

- There is one ternary operator in Java
  - Inherited from C and C++
  - The same as an `if-else` statement but the result is an expression

- The conditional operator is `? :`

```
test-expression ? then-expression : else-expression
```

- The test-expression must be a `boolean` expression
  - If test-expression evaluates to `true`, the then-expression is evaluated and the result returned
  - If the test-expression is `false`, then the else-expression is evaluated and returned

The Ternary Operator Example

Example 3-15: Conditional operator

```
class Ex3_15 {
  public static void main(String [] args) {
    String message;
    boolean test = false;
     // Using an if statement
    if (test){
      message= "Then clause";
    }
    else {
       message = "Else clause";
    }
    System.out.println(message);
   // Using a conditional operator
   System.out.println(true ? "Then expression" :
                                              "Else expression");
   System.out.println(false ? "Then expression" :
                                              "Else expression");
  }
}
```

Summary

We covered

- What statements and blocks are in Java
- What a local variable is and its scope is
- What the flow of control in Java program is
- `if` statements and `switch` statements

Exercises

- *Exercise 3-1:* Statements, Blocks and Local Variables
  - *In this lab you will investigate the correct usage of statements, blocks and how local variables behave.*

- *Exercise 3-2:* `if` and `switch` statements
  - *In this lab you will work with local variables and `if` statements and will solve a programming problem using a `switch` statement*

Control Structures in Java

Part II

Objectives

At the end of this section, you should be able to:

- Use `while`, `do-while` and `for` loops
- Use `break` and `continue` statements
- Use method overloading correctly

Loops in Java

- There are three looping structures in Java
  - `while` loops
  - `do-while` loops
  - `for` loops
- The `while` and `do-while` loops are more natural for iterating while some test condition is `true`
- The `for` loops are more natural when iterating over arrays or when it is convenient to have loop counter or index available
- The syntax for should be familiar

while

- The most basic looping structure
  - Condition is evaluated prior to executing the body
  - The body is executed as long as a condition is true
  - The `while` loop looks like this:

```
while(test-condition) {
  loop-body
}
```

- Remember test conditions must result in either a `boolean` `true` or `false`

while Example

Example 3-16: `while` loop example

```
int count = 0;

// while loop with a statement as a loop body
while (count < 10)
  System.out.println("count is "+ count++);

// while loop with a block as a loop body
count = 0;
while (count < 10) {
    System.out.println("count is "+ count++);
    count++;
}
```

do-while

- A variation on the `while` loop is the `do-while` loop
  - Always executes the body of the loop at least once
  - Determines subsequent execution based on condition
  - The `do-while` loop looks like this:

```
do {
  loop-body
}while (test-condition);
```

- The `do` indicates the start of the loop body and the test condition appears after the `while`
  - Notice that there is a semi-colon after the test condition
  - If the loop-body is a statement it must be terminated with a semi-colon

do-while Example

Example 3-18: while loops of Example 3-16 as do-while loops

```
int count = 0
// do-while loop with a statement as a loop body
do
  System.out.println("count is "+ count++);
while (count < 10);
// do-while loop with a block as a loop body
count = 0;
do {
  System.out.println("count is "+ count);
  count++;
}while (count < 10);
```

Command-line Input Example I

Example 3-19: Using a `while` loop

```
class DoWhileTest1 {
    public static void main(String [] args) {
   // Using a while loop
      try {
        char input;
        String output = "";
        input = (char)System.in.read();
        while(input != '\n') {
          output = input + output;
          input = (char)System.in.read();
        }
        System.out.println(output);
   }
   catch (Exception e) {
     System.out.println("IO Exception:" + e);
       }
    }
 }
```

Command-line Input Example II

Example 3-20: Using a `do-while` loop

```
class DoWhileTest2 {
  public static void main(String [] args) {
   //Using a do-while loop
      try {
        char input;
        String output = "";
        do {
          input = (char)System.in.read();
          output = input + output;
        } while(input != '\n');
        System.out.println(output);
      }
      catch (Exception e) {
        System.out.println("IO Exception:" + e);
      }
   }
}
```

for Loops

- Provide the same functionality as `while` loops
- Are intended to make iterations involving counters or indexes easier to program
- Has the following structure

```
for (initial-clause; test-clause; iteration-clause)
  loop-body
```

- It is not required that you provide a valid expression for each clause, we will see more on this later

for Example I

Example 3-23: Basic for loop

```
class ForLoop {
  public static void main(String [] args) {
    int sum = 0;
    int index;
    for (index = 1; index <=100; index++) {
      sum += index;
    }
    System.out.println("Sum ="+sum);
  }
}
```

for Example II

Example 3-24: More for loop

```
class ForLoop2 {
  public static void main(String [] args) {
    int [] forwards = {0,1,2,3,4,5,6,7,8,9};
    int [] backwards = {0,1,2,3,4,5,6,7,8,9};
    int idx1, idx2;
    for (idx1 = 0, idx2 = 9; idx1 <forwards.length; idx1++, idx2--){
      backwards[idx2]=forwards[idx1];
    }
    System.out.println("Backwards is now..");
    for (idx1 = 0; idx1 <backwards.length; idx1++){
         System.out.println(backwards[idx1]);
    }
  }
}
```

Local Variables

- Remember, local variables are defined within blocks
- Their existence is defined by their scope
- Local variables can be used in loops to hold temporary data
- Local variables can be defined as part of the loop test expression or as variables in the body

Local Variables in for Loops

Example 3-25: More for loop

```
class ForLoop3 {
  public static void main(String [] args) {
    int outer = 0; //local to main
    //middle is defined in the for construct
    for (int middle = 0; middle <10; middle++) {
      int inner = 0; //same scope as middle
      inner++;
      outer++;
      System.out.println("outer="+outer+" middle="+middle+
              " inner="+inner);
    }
    System.out.println(" outer="+outer);
    // This following two lines will not compile because
    // they are out of scope for middle and inner.
    //System.out.println(" inner="+inner);
    //System.out.println(" middle="+middle);
  }
}
```

Local Variables Example Output

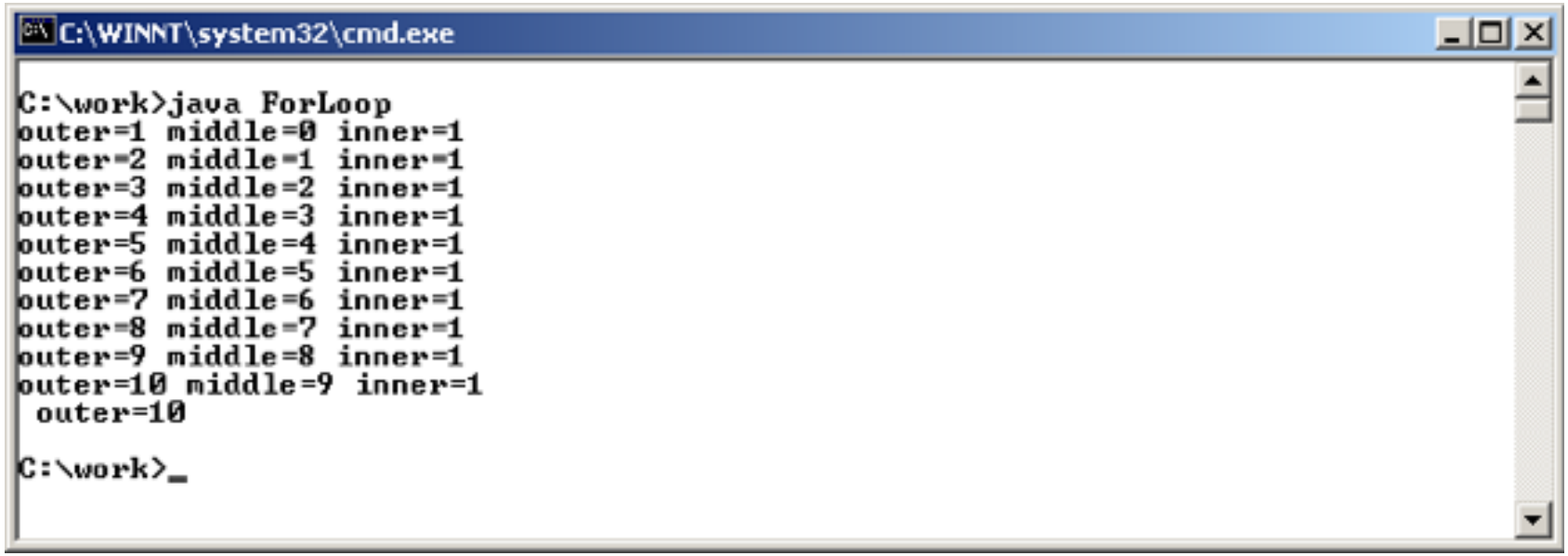


Fig. 4-2: **Output of Example 3-25**

More on Local Variables in Loops

Placement of local variables can sometimes cause unwanted results

```
// This is okay
int k; String s;
for (k=23, s="hi";;;)
{ /*body */ }

// As is this
for (int k=0, j=1;;)
{/* body */ }

// This doesn't work -- compiler thinks j is being redeclared.
int j;
for (int k =0, j=1;;)
{ /* body */ }

// Nor does this -- compiler does not expect to find "String"
// type must be consistent across all declared variables
for (int k=0, String s="hi";;) {/* body */}
```

Variations on the for Loop

Example 3-26: Variations on the `for` loop

```
// Variation one – an infinite for loop
for (;;) {
   // Only way to end this one is with a break
   if (conditions) break;
}
// Variation two – only a test expression
// Equivalent to a while loop
int k = 0;
for (; k < 10;) {
  k++
}
// Variation three – just an initialization clause
int z;
for (int k = 0, z=35;;) {
  if (++j >0) break;
}
// Variation four – just an iteration clause
int z=35;
for (;;z++) {
  if (z >100) break;
}
```

The break Statement

- The preceding flow control constructs executed until some condition fails
- In some cases, it is necessary to stop the execution of the loop structure due to some “local” condition
- The break statement in Java produces an abrupt termination of control
- As soon as the break statement is encountered then the processing breaks out of the loop and goes to the first statement after the loop body
- It is possible to utilize nested breaks within nested loops

For-each Loop

- What is it?
  - A variation of a for loop
  - Simplified notation targeted at collections and arrays
- Why does it exist?
  - Remove redundant iteration code
  - Simplify iteration over collections

For-each Loop [cont.]

- How does it work?
  - Functions like a for loop . .
    - Iterate over collection
    - Access each collection element individually
  - . . but has a different syntax
    - Has an initialization “clause” and “expression” clause
      - Initialization clause holds current element in collection
      - Expression clause defines collection
    - Doesn’t have a “test” clause or “increment” clause
    - Clauses separated by `:` instead of `;`
  - Translated into a formal for loop at compile time

Simple For-each Loop Example

```
1   package examples.foreach;
2
3   /**...*/
10  public class SimpleForEach {
11
12    public static void main(String[] args) {
13      //initialization clause - String s
14      //expression clause - args
15      for(String s : args) {
16        System.out.println(s);
17      }
18    }
19  }
20
```

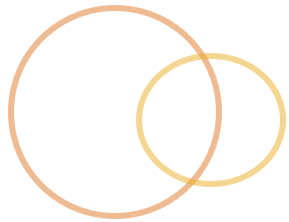
Limitations of For-each Loop

- Limited type support
  - Supports arrays
  - Supports `java.lang.Iterable`
    - Collections support `Iterable` through class hierarchies
    - `Iterable` provides `java.util.Iterator` to the for-each loop
- Limited insight
  - No way to determine “where am I” during iteration
  - No access to iterator or index
- All clauses are required - no “endless loop” abilities

Iterable For-each Loop Example

```
1   package examples.foreach;
2
3   import java.util.Arrays;
4   import java.util.List;
5
6   /**
7    * The following illustrates using the for-each
8    * loop with a collection through the Iterable
9    * interface.
10   */
11  public class IterableForEachExample {
12
13    public static void main(String[] args) {
14      //convert the array into a list
15      List argList = Arrays.asList(args);
16
17      //iterate over the list
18      for(Object arg : argList) {
19        System.out.println(arg);
20      }
21    }
22  }
23
```

The break Statement



Example 3-27: The break statement

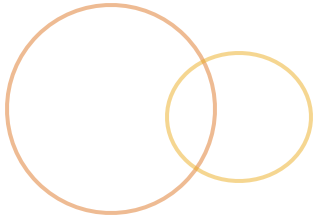
```
class BreakTest {
  public static void main(String [] args) {
    int [] values = {9,45,1,0,98,102,-34};
    int idx = 0;
    while (idx < values.length) {
      if (values[idx ] == 98)
        break; //break out of while
      idx++;
    }
    if (idx == values.length) {
      System.out.println("98 was not found");
    }
    else {
      System.out.println("98 found at index "+ idx +" in values");
    }
  }
}
```

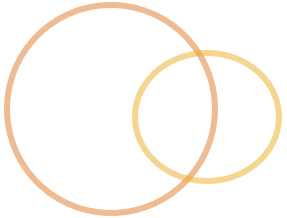
Nested loop break Example

Example 3-28: The break statement in nested loops

```
class BreakTest2 {
  public static void main(String [] args) {
    int [] target = {9,45,1,0,98,102,-34};
    int [] test =  { 9, 30, 102, 14 };
    int idx1 = 0;
    int found = -1;
    // Outer loop
    while (idx1 < test.length) {
      int idx2 = 0;
      while (idx2 < target.length) {// Inner loop
        if (test[idx1] == target[idx2])  {
          found = idx2;
          break; //break inner loop
        }
        idx2++;
      }
      idx1++;
    }
    if (found == -1) {
      System.out.println("No test values found");
    }
    else {
      System.out.println("Found test value "+ target[found] +
        " at index "+ found +" in target");
    }
  }
}
```

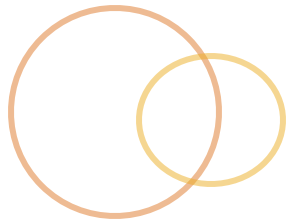
Labels





- When dealing with nested loops, using break may not provide the level of termination precision required
- For example, you may want to break out of the entire looping structure when something fatal occurs
- Java provides a supporting construct called *labels*
- Anything in Java can be labeled; however labels really only make sense in the context of loops
- The basic syntax of a label is
  `label_name:statement`
- Labels used with `break`s tell the JVM specifically where to terminate

The break Statement



Example 3-29: The labeled break statement in nested loops -- this works

```
class BreakTest3 {
  public static void main(String [] args) {
    int [] target = {9,45,1,0,98,102,-34};
    int [] test =  { 9, 30, 102, 14 };
    int idx1 = 0;
    int found = -1;
    // Outer loop with label zippy
    zippy: while (idx1 < test.length) {
      int idx2 = 0;
        // Inner loop
        while (idx2 < target.length) {
          if (test[idx1] == target[idx2])  {
            found = idx2;
            break zippy; //stop the execution of zippy:while
          }
          idx2++;
        }
        idx1++;
     }
     if (found == -1) {
         System.out.println("No test values found");
     }
     else {
       System.out.println("Found test value "+ target[found] +
            " at index "+ found +" in target");
     }
  }
}
```

The continue Statement

- In some cases, breaking out of a loop may not be desired
- Instead of breaking out of the loop, the current iteration is cancelled and the next iteration is started
- Just like `break` statements, `continue` statements can be labeled or unlabeled

continue Statement Example I

- In Example 3-30, if the remainder after division by 2 (the modulus operator) is not 0, then we have an odd number so we just start the next iteration and skip over the output statement

Example 3-30: The continue statement

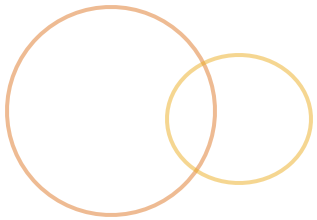
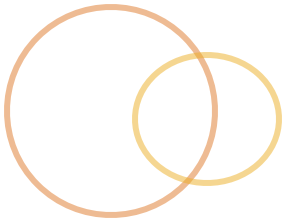
```
class ContinueTest {
  public static void main(String [] args) {
    int [] target = {9,45,1,0,98,102,-34};
    for (int idx = 0; idx < target.length; idx++) {
      if (target[idx] %2 != 0) {
        continue;
      }
      System.out.println(target[idx]+" is even");
    }
  }
}
```

continue Statement Example II

Example 3-31: The labeled continue statement in nested loops

```
class ContinueTest {
  public static void main(String [] args) {
    int [] target = {9,45,1,0,98,102,-34};
    int [] test =  { 9, 30, 102, 14 };
    // Outer loop with label zippy
    zippy: for (int idx1= 0; idx1 < test.length; idx1++) {
      for (int idx2 = 0; idx2 < target.length; idx2++) {
        if (test[idx1] == target[idx2])  {
          System.out.println("Found "+ test[idx1] +
                 " at " + idx2);
          continue zippy;
        }
      }
    }
  }
}
```

Methods


- Methods are equivalent to functions in structured programming.
- A method consists of three parts: a return value, a signature and a body.
- Methods define the behaviors of the Objects created from the class templates

Example 3-32: Some methods

```
class Test {
  int method1() {/* method body */}
  void method2(int x) {}
  void method2(float x) {}
}
```

Return Values of Methods

- All Methods have a return value
  - A method can have only a single return value
  - There are three common types of return values:
    - `void` - nothing is returned
    - Primitive - `int`, `char`, `long`, etc.
    - Reference Value (object) - `String`, `BankAccount`, etc

Example 3-32: Some methods

```
class Test {
  int method1() {/* method body */}
  void method2(int x) {}
  void method2(float x) {}
}
```

Method Signatures and Overloading

- The signature of a method is the method name (indentifier) plus the list of parameter types
- All method signatures in a class must be unique, which means that all methods either have:
  - Different names or
  - The same name, but different argument lists
- Methods the same name but differing in argument lists are referred to as *overloaded* methods
- The uniqueness of signatures only applies *within* a class definition

Method Signatures and Overloading

Example 3-34: Classes and methods

```
class Bob {
  static void print(int x) {
    System.out.println("Integer: "+ x);
  }
  static void print(float x) {
    System.out.println("Float: "+ x);
  }

class Fred {
  static void print(int x) {
     System.out.println("Integer: "+ x);
  }
}
```

Method Signatures and Overloading

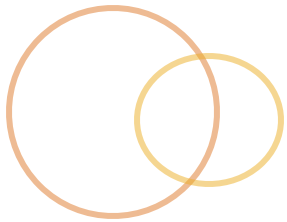
A look at `java.io.PrintStream` would reveal overloading of `println`

```
void println()          Write line separator string.
void println(boolean x)   Print a boolean
void println(char x)  Print a character.
void println(char[] x)    Print an array of characters.
void println(double x)    Print a double.
void println(float x)     Print a float.
void println(int x)       Print an integer.
void println(long x)      Print a long.
void println(Object x)    Print an Object.
void println(String x)    Print a String
```

Method Invocation

- In order to perform some operation, a method invocation must occur
- In fact, an initial method invocation is required in order for our application to execute - main
- In the definition of applications, objects will interact with other objects using method invocation
- In Java, parameters declared in methods have scope local to the method
- Currently, Java does not support optional parameters or default parameters to methods

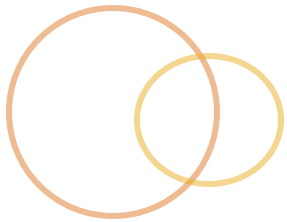
Calling a Method



Example 3-35: Calling a method

```
class Ex4_35 {
   static void print(int x) {
     System.out.println("Integer: "+ x);
   }
   // illegal because the method has the same signature as another
   // method in this class - only differs by return value.
   // static int void print(int x) { return x;}
   public static void main(String [] args) {
     int z = 3;
     print(z);
   }
 }
```

Method Parameters



Example 3-36: Argument promotion

```
class Ex4_36 {
  static void print(int x) {
    System.out.println("Integer: "+ x);
  }
  static void print(byte x) {
    System.out.println("Byte: "+ x);
  }
  public static void main(String [] args) {
    int z = 3;
    print(z);
    byte b = 9;
    print(b);
    short s = 12;
    print(s);
  }
}
```

Method Parameters

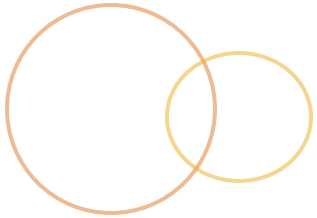
Fig 4-4: **Output of Example 3-36**

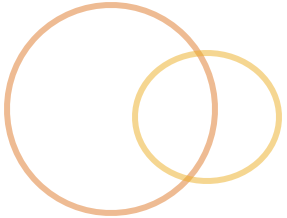
Optional Arguments and Default Parameters

Example 3-37: Default Parameter

```
class Ex4_37 {
  static void print(int x, char language) {
    switch (language) {
      case 'E':
      case 'e':
        System.out.println("The number is: "+ x);
        break;
      case 'F':
      case 'f':
        System.out.println("Le numeral est:"+ x);
        break;
         /* -- more cases here -- */
    }
    return;
  }
  // The version where language defaults to English
  static void print(int x) {
    print(x,'E');
    return;
  }
  public static void main(String [] args) {
    print(3);
    print(4,'e');
    print(5,'f');

  }
}
```

Varargs





- Why does it exist?
  - Simplifies passing flexible number of arguments
  - Typical argument list notation is inflexible
    - Certain scenarios require flexibility
    - Flexibility was supported through an `Object[]` argument
  - `Object[]` support was cumbersome
    - Required developer to declare method with `Object[]` argument
    - Required developer to convert parameters into an array before passing

Varargs Example [old way]

```
1  package examples.varargs;
2
3  /**...*/
13 public class VarArgsOldWay {
14
15   public static void main(String[] args) {
16     String name = "John Doe";
17     String book1 = "Hooked On Java";
18     String book2 = "The Java Language Specification";
19     //convert arguments into array
20     String [] titles = {book1, book2};
21     //pass arguments as array
22     listBooks(name, titles);
23   }
24
25   private static void listBooks(String name, String[] titles) {
26     System.out.print(name + " likes: ");
27     for(int i=0;i<titles.length;i++) {
28       System.out.print("\"" + titles[i] +"\"");
29       if(i+1 < titles.length)
30         System.out.print(",");
31     }
32     System.out.flush();
33   }
34 }
35
```

Using Varargs

- How does it work?
  - Change method signature to support varargs
    - Include ***ellipse*** notation
    - Must be last argument in argument list
  - Pass varargs either as:
    - An array of objects (like old way)
    - Like normal arguments (comma separated)
  - Varargs automatically converted
    - Signature converted to support [ ]
    - Arguments converted into an array
    - All performed by compiler
  - Work with varargs like any other array

Simple varargs Example

```
1  package examples.varargs;
2
3  /**...*/
11 public class VarArgsNewWay {
12
13   public static void main(String[] args) {
14     String name = "John Doe";
15     String book1 = "Hooked On Java";
16     String book2 = "The Java Language Specification";
17     //pass arguments as arguments
18     listBooks(name, book1, book2);
19   }
20
21   private static void listBooks(String name, String... titles) {
22     System.out.print(name + " likes: ");
23     for(int i=0;i<titles.length;i++) {
24       System.out.print("\"" + titles[i] +"\"");
25       if(i+1 < titles.length)
26         System.out.print(",");
27     }
28     System.out.flush();
29   }
30 }
31
```

Returning From A Method

- A method ends when it encounters a `return` statement
  - A `return` statement usually has a type following it
  - The return type matches what is specified in the method declaration

- If a method is declared to return `void`, no `return` statement is needed

Summary

We covered

- `while`, `do-while` and `for` loops
- `break` and `continue` statements
- method overloading correctly

Exercises

- *Exercise 3-3: Loops*
  - *In this lab, you will work closely with both for and while loops.*
- *Exercise 3-4: Methods*
  - *In this lab, you will work with methods, including a recursive method.*