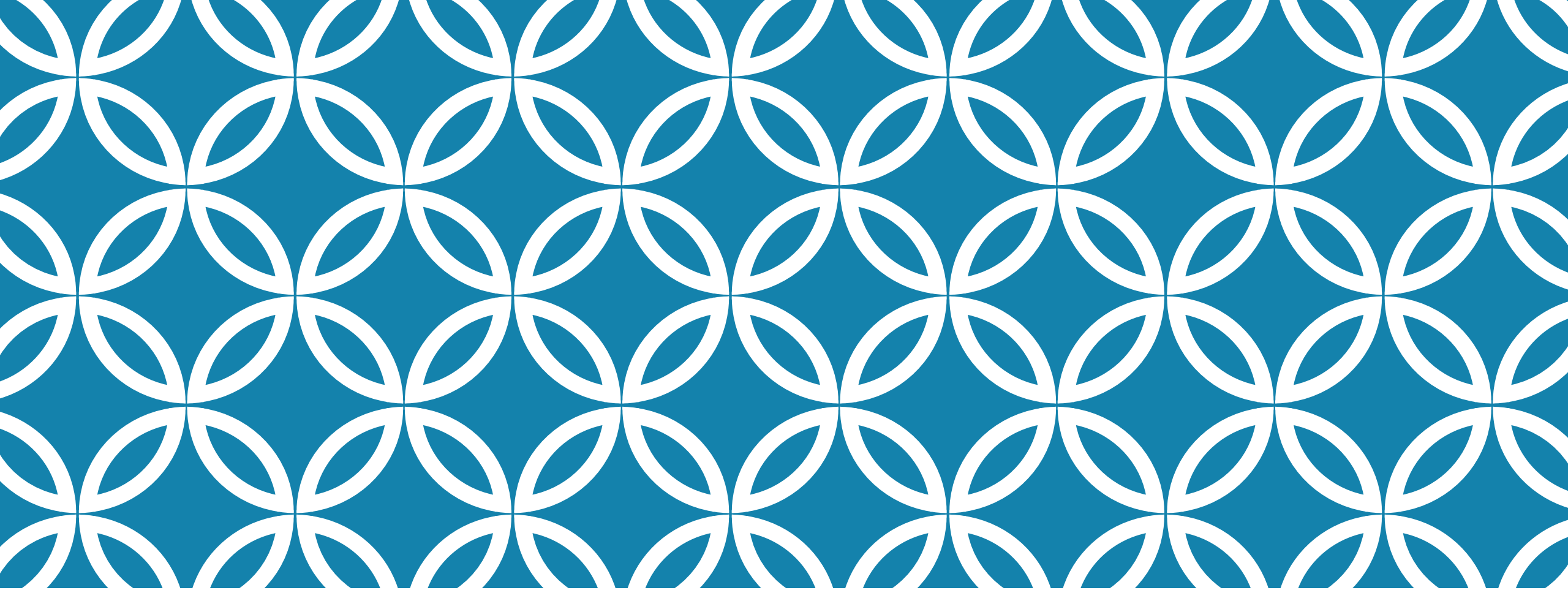

CHAPTER 5
CONTROL STATEMENTS

CONTROL STATEMENTS

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program.

Control statement categories:

- Selection
- Iteration
- Jump

JAVA'S SELECTION STATEMENTS

IF STATEMENT

if statement is Java's conditional branch statement

general form of the **if** statement:

if (condition)

statement1;

else

statement2;

The *condition* is any expression that returns a **boolean** value.

IF STATEMENT

Example 1,

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Example 2,

```
boolean dataAvailable;
// ...
if (dataAvailable)
      ProcessData();
else
      waitForMoreData();
```

IF STATEMENT

Example 3,

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else
    waitForMoreData();
```

NESTED IFS STATEMENT

When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

For Example,

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

IF-ELSE-IF LADDER STATEMENT

General Form:

```
if(condition)
        statement;
else if(condition)
        statement;
else if(condition)
        statement;
...
else
        statement;
```

IF-ELSE-IF LADDER STATEMENT

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

Here is the output produced by the program:

```
April is in the Spring.
```

SWITCH STATEMENT

The **switch** statement is Java's *multiway branch* statement.

General form of a **switch** statement:

```
switch (expression) {
case value1:
    // statement sequence
    break;
case value2:
    // statement sequence
    break;
...
case valueN:
    // statement sequence
    break;
default:
    // default statement sequence
}
```

expression must be of type **byte, short, int,** or **char;** each of the *values* specified in the **case** statements must be of a type compatible with the expression

SWITCH STATEMENT

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

SWITCH STATEMENT

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

SWITCH STATEMENT

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more");
      }
  }
}
```

This program generates the following output:

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

NESTED SWITCH STATEMENTS

For example,

```
switch(count) {
        case 1:
          switch(target) { // nested switch
                case 0:
                        System.out.println("target is zero");
                        break;
                case 1: // no conflicts with outer switch
                        System.out.println("target is one");
                        break;
                }
                break;
        case 2: // ...
        ......
}
```

SWITCH STATEMENT

In summary, there are three important features of the **switch** statement to note:

- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.
- A **switch** statement is usually more efficient than a set of nested **if**s.

Note: When it compiles a **switch** statement, the Java compiler will inspect each of the **case** constants and create a "***jump table***" that it will use for selecting the path of execution depending on the value of the expression.

ITERATION STATEMENTS

WHILE STATEMENT

General form:

```
while(condition)
{
        // body of loop
}
```

The ***condition*** can be any Boolean expression.

WHILE STATEMENT

```
// Demonstrate the while loop.
class While {
 public static void main(String args[]) {
  int n = 10;
  while(n > 0) {
     System.out.println("tick " + n);
      n--;
    }
 }
}
```

When you run this program, it will "tick" ten times:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

WHILE STATEMENT

The body of the **while** (or any other of Java's loops) can be empty. This is because a ***null statement*** (one that consists only of a semicolon) is syntactically valid in Java.

```
// The target of a loop can be empty.

class NoBody {

public static void main(String args[]) {
  int i, j;
  i = 100;
  j = 200;
  // find midpoint between i and j
  while(++i < --j); // no body in this loop
  System.out.println("Midpoint is " + i);
  }
}
```

This program finds the midpoint between i and j. It generates the following output:

```
Midpoint is 150
```

DO-WHILE STATEMENT

Sometimes it is desirable to execute the ***body of a loop at least once***, even if the conditional expression is false.

Its **general form** is

```
do {
// body of loop
} while (condition);
```

The **do-while** loop is especially useful when you process a ***menu selection***.

DO-WHILE STATEMENT

Example:

```
// Demonstrate the do-while loop.
class DoWhile {
public static void main(String args[]) {
  int n = 10;
  do {
        System.out.println("tick " + n);
        n--;
  } while(n > 0);
 }
}
```

FOR STATEMENT

the **traditional for** statement:

```
for(initialization; condition; iteration) {
// body
}
```

Example:

```
// Demonstrate the for loop.
class ForTick {
  public static void main(String args[]) {
  int n;
  for(n=10; n>0; n--)
  System.out.println("tick " + n);
  }
}
```

FOR STATEMENT

Declaring Loop Control Variables Inside the for Loop

```
// Declare a loop control variable inside the for.
class ForTick {
  public static void main(String args[]) {
  // here, n is declared inside of the for loop
  for(int n=10; n>0; n--)
  System.out.println("tick " + n);
  }
}
```

Another Example:

FOR STATEMENT

Using the Comma

Example:

```
// Using the comma.
class Comma {
public static void main(String args[]) {
  int a, b;
  for(a=1, b=4; a<b; a++, b--) {
  System.out.println("a = " + a);
  System.out.println("b = " + b);
  }
}
}
```

NOTE

If you are familiar with C/C++, then you know that in those languages the comma is an operator that can be used in any valid expression. However, this is not the case with Java.

In Java, the comma is a separator.

FOR STATEMENT

Some for Loop Variations

1)

```
boolean done = false;
for(int i=1; !done; i++) {
// ...
if(interrupted()) done = true;
}
```

FOR STATEMENT

Some for Loop Variations

2)

```
// Parts of the for loop can be empty.
class ForVar {
public static void main(String args[]) {
  int i;
  boolean done = false;
  i = 0;
  for( ; !done; ) {
  System.out.println("i is " + i);
  if(i == 10) done = true;
  i++;
}   }   }
```

FOR STATEMENT

Some for Loop Variations

3) Infinite loop

```
for( ; ; ) {
// ...
}
```

FOR STATEMENT

The For-Each Version of the for Loop (or enhanced for loop)

General Form:

for(*type itr-var : collection*) *statement-block*

Here collection is type of array, collections framework

With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained.

Example:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

Its iteration variable is "read-only" as it relates to the underlying array.

FOR STATEMENT

The For-Each Version of the for Loop (or enhanced for loop)

Example:

```
// Use a for-each style for loop.
class ForEach {
  public static void main(String args[]) {
    int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int sum = 0;

    // use for-each style for to display and sum the values
    for(int x : nums) {
      System.out.println("Value is: " + x);
      sum += x;
    }

    System.out.println("Summation: " + sum);
  }
}
```

FOR STATEMENT

Iterating Over Multidimensional Arrays

```
// Use for-each style for on a two-dimensional array.
class ForEach3 {
public static void main(String args[]) {
int sum = 0;
int nums[][] = new int[3][5];
// give nums some values
for(int i = 0; i < 3; i++)
for(int j=0; j < 5; j++)
nums[i][j] = (i+1)*(j+1);
// use for-each for to display and sum the values
for(int x[] : nums) {
for(int y : x) {
System.out.println("Value is: " + y);
sum += y;
}
}
System.out.println("Summation: " + sum);
}
}
```

FOR STATEMENT

The output from this program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

FOR STATEMENT

For-Each related applications:

- Searching
- computing an average
- finding the minimum or maximum of a set
- looking for duplicates
- etc

FOR STATEMENT

Nested Loops

```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<10; i++) {
for(j=i; j<10; j++)
System.out.print(".");
System.out.println();
}
}
}
```

The output produced by this program is shown here:

```
.........
........
.......
.......
......
.....
....
...
..
.
```

JUMP STATEMENTS

BREAK

Three uses:

- it terminates a statement sequence in a **switch** statement
- it can be used to exit a loop
- it can be used as a “civilized” form of goto

BREAK

Example:

```
// Using break to exit a loop.
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

BREAK

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

BREAK

When used inside a set of nested loops, the **break** statement will only break out of the innermost loop.

Example:

```
// Using break with nested loops.
class BreakLoop3 {
public static void main(String args[])
{
for(int i=0; i<3; i++) {
System.out.print("Pass " + i + ": ");
for(int j=0; j<100; j++) {
if(j == 10) break; // terminate loop if j is 10
System.out.print(j + " ");
}
System.out.println();
}
System.out.println("Loops complete.");
}
}
```

BREAK

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

BREAK - USING BREAK AS A FORM OF GOTO

Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner.

General Form:

break *label*;

Here, label is the name of a label that identifies a block of code.

BREAK - USING BREAK AS A FORM OF GOTO

```
// Using break as a civilized form of goto.
class Break {
  public static void main(String args[]) {
    boolean t = true;

    first: {
      second: {
        third: {
          System.out.println("Before the break.");
          if(t) break second; // break out of second block
          System.out.println("This won't execute");
        }
        System.out.println("This won't execute");
      }
      System.out.println("This is after second block.");
    }
  }
}
```

Running this program generates the following output:

```
Before the break.
This is after second block.
```

CONTINUE

You might want to continue running the loop but stop processing the remainder of the code in its body for this particular Iteration.

```
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

This code uses the % operator to check if i is even. If i
a newline. Here is the output from this program:

```
0 1
2 3
4 5
6 7
8 9
```

CONTINUE — WITH A LABEL

```
// Using continue with a label.
class ContinueLabel {
  public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
         for(int j=0; j<10; j++) {
           if(j > i) {
             System.out.println();
             continue outer;
           }
           System.out.print(" " + (i * j));
         }
       }
       System.out.println();
   }
}
```

CONTINUE – WITH A LABEL

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

RETURN

The **return** statement is used to explicitly return from a method.

That is, it causes program control to **transfer back to the caller of the method**.

```
// Demonstrate return.
class Return {
public static void main(String args[]) {
boolean t = true;
System.out.println("Before the return.");
if(t) return; // return to caller
System.out.println("This won't execute.");
}
}
```

DISCLAIMER

- These slides are not original and have been prepared from various sources for teaching purpose.

Sources:

- Herbert Schildt, Java™: The Complete Reference

THANK YOU....