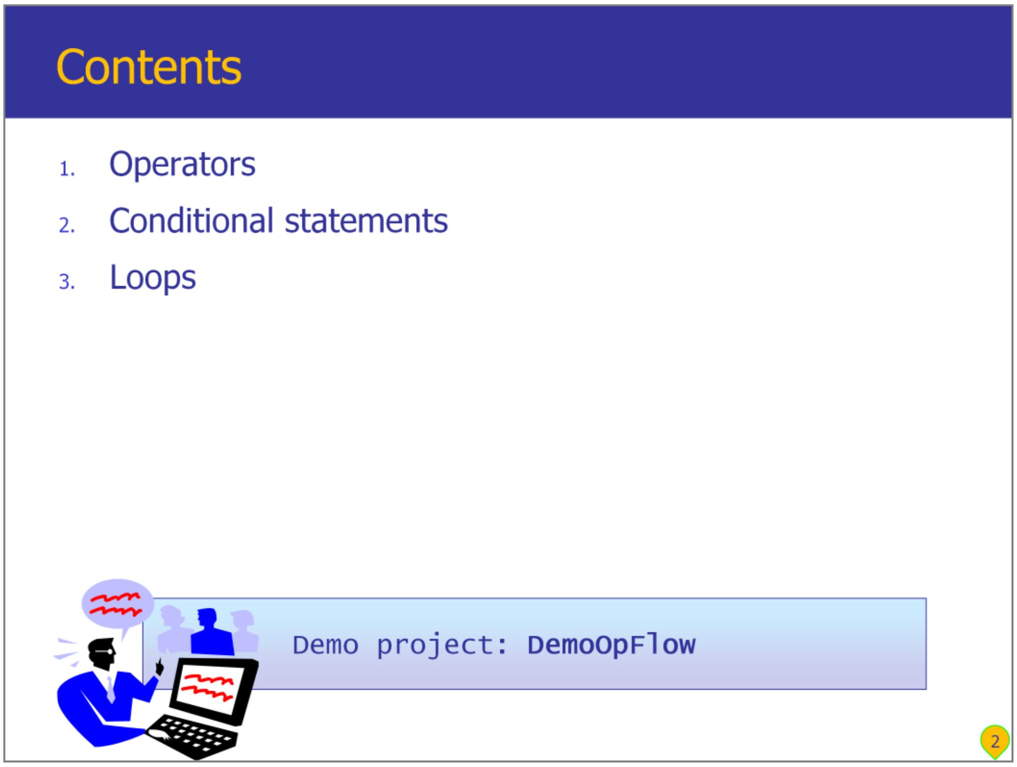
Operators and Flow Control

olsen
software ltd

This is quite a short and simple chapter, where we formally introduce the operators and flow-of-control mechanisms in Java. Most of this chapter will probably be familiar(ish) to you, from your experience with other programming languages.



Contents

1. Operators
2. Conditional statements
3. Loops

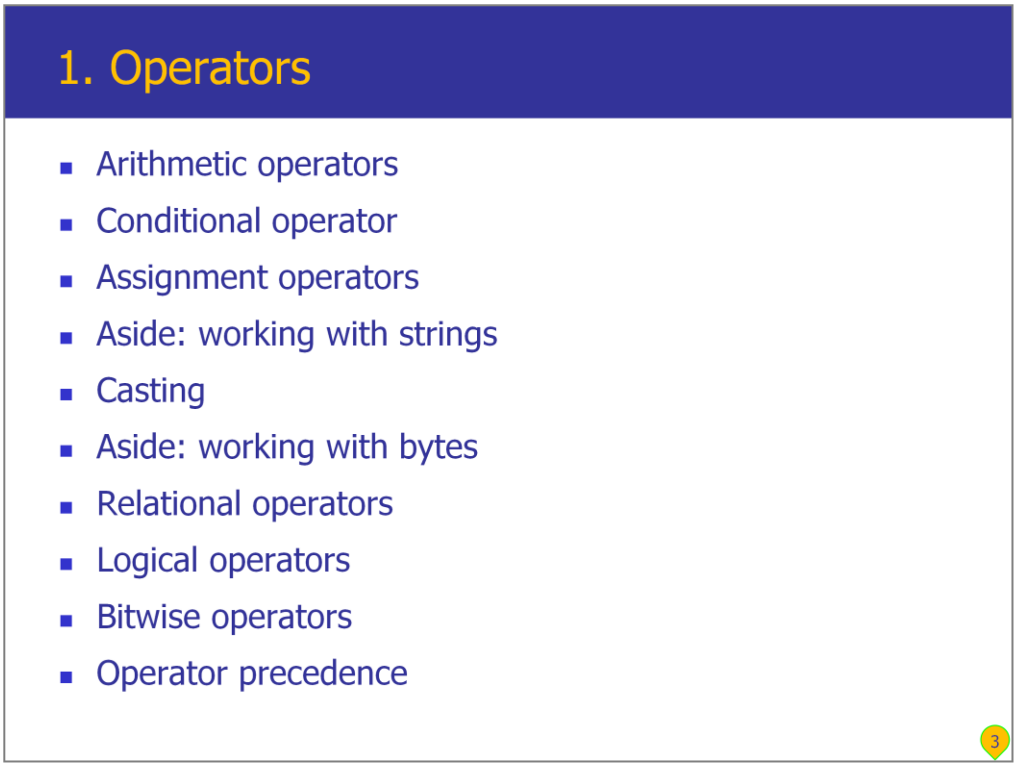
Demo project: `DemoOpFlow`

Section 1 in this chapter covers the operators available in Java. Most of these operators are intuitive, but there are a few gotchas to be aware of.

Section 2 shows how to write conditional statements, using `if/else` and `switch` constructs.

Section 3 shows how to write loops, using `for`, `while`, and `do/while` constructs. We also show how to use the `break` and `continue` keywords to perform unconditional jumps.

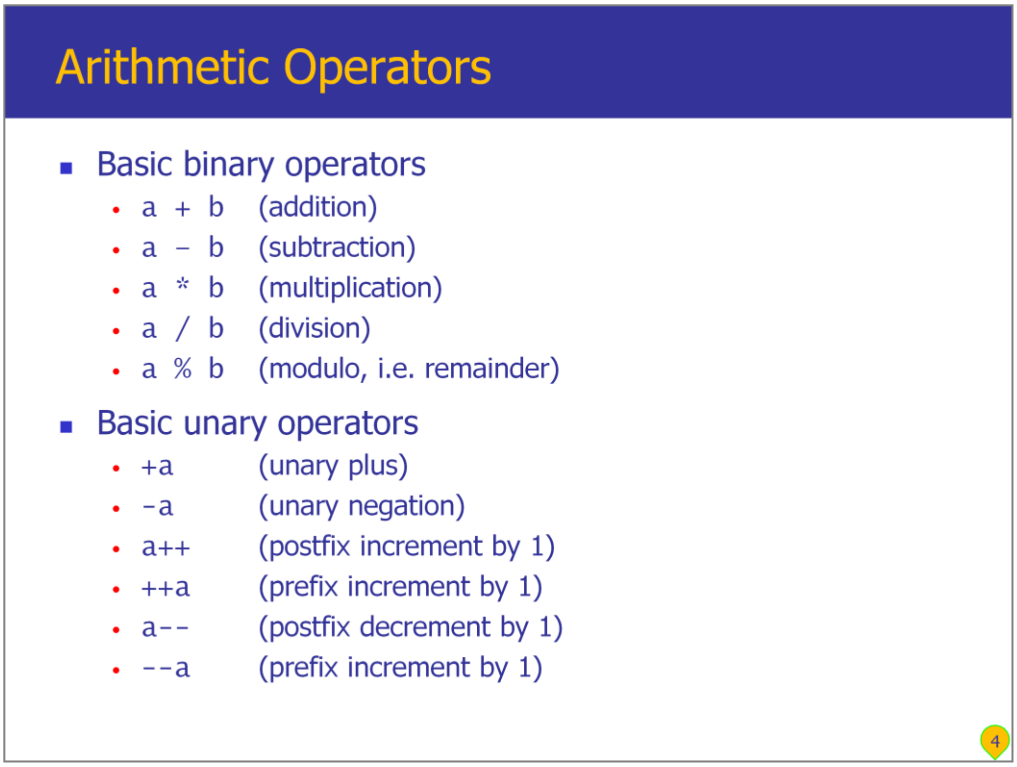
The demos for chapter are located in the `DemoOpFlow` project.



1. Operators

- Arithmetic operators
- Conditional operator
- Assignment operators
- Aside: working with strings
- Casting
- Aside: working with bytes
- Relational operators
- Logical operators
- Bitwise operators
- Operator precedence

This section takes you on a guided tour through Java's operators.



Arithmetic Operators

- Basic binary operators

`a + b`	(addition)
`a - b`	(subtraction)
`a * b`	(multiplication)
`a / b`	(division)
`a % b`	(modulo, i.e. remainder)

- Basic unary operators

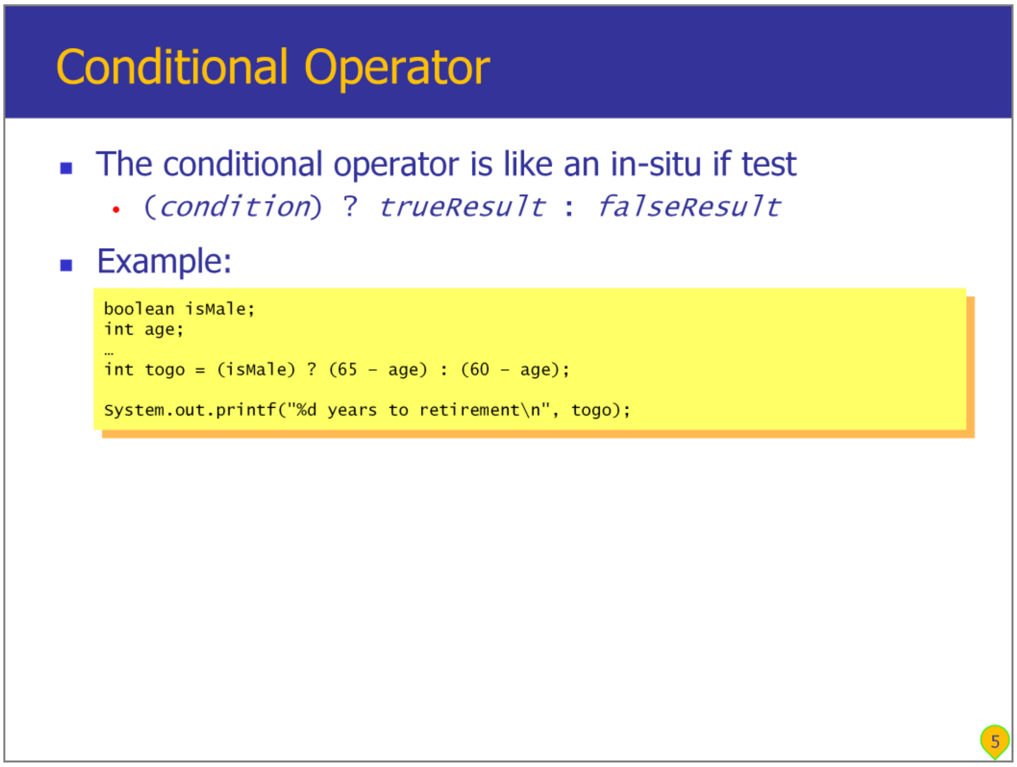
`+a`	(unary plus)
`-a`	(unary negation)
`a++`	(postfix increment by 1)
`++a`	(prefix increment by 1)
`a--`	(postfix decrement by 1)
`--a`	(prefix increment by 1)

4

Java provides all the arithmetic operators you might expect. Note the following points in particular:

- If you use / to divide two integral values, the result is also an integer. The result is rounded downwards. For example, 7/4 gives the result 1 (not 1.75).
- The % operator gives you the remainder from an integer division. For example, 7%4 gives the result 3 (4 goes into 7 once, with a remainder of 3).
- The ++ and -- operators can be used in the prefix or postfix positions. The following example explains the difference:

```
int a = 1;
int b = ++a;   // Increments a, then assigns a to b.
System.out.printf("%d %d\n", a, b);  // Prints 2 2

int a = 1;
int b = a++;   // Assigns a to b, then increments a.
System.out.printf("%d %d\n", a, b);  // Prints 2 1
```



Conditional Operator

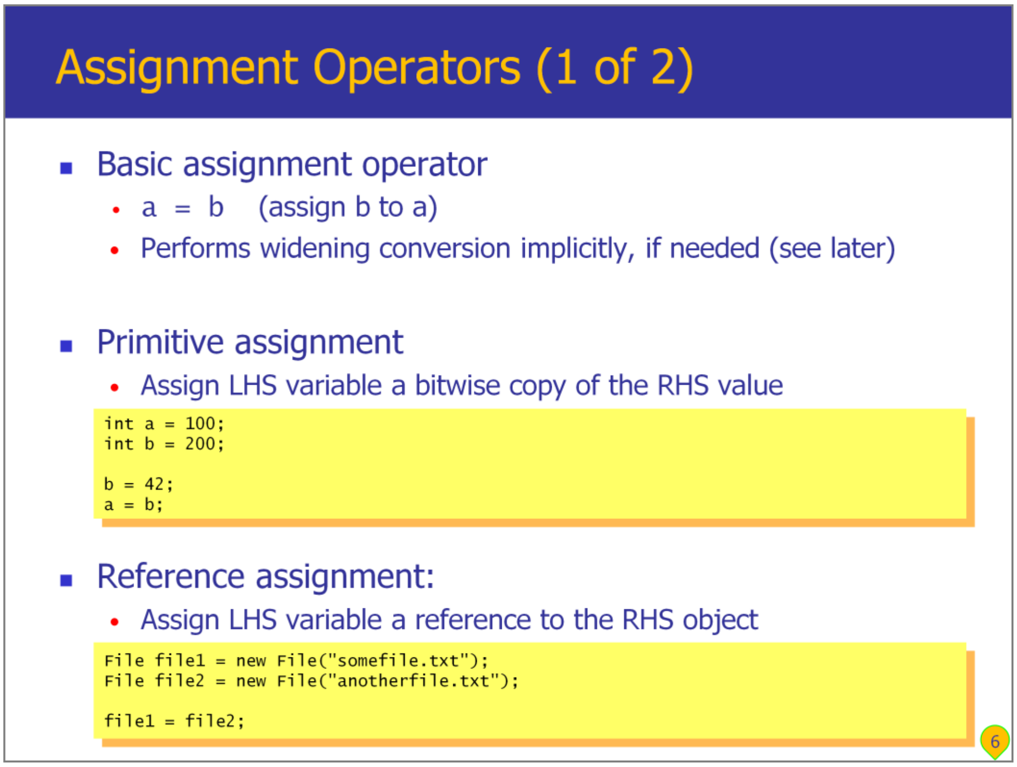
- The conditional operator is like an in-situ if test
  - *(condition) ? trueResult : falseResult*
- Example:

```
boolean isMale;
int age;
…
int togo = (isMale) ? (65 - age) : (60 - age);

System.out.printf("%d years to retirement\n", togo);
```

5

The conditional operator allows you to perform if-else tests in a single expression. This is handy if you want to embed a decision inside another expression, as shown in the example in the slide. You could achieve the same effect (less elegantly) using an if-else statement as follows:

```
boolean isMale;
int age;
…
int togo;
if (isMale)
    togo = 65 - age;
else
    togo = 60 - age;

System.out.printf("%d years to retirement\n", togo);
```



Assignment Operators (1 of 2)

- Basic assignment operator
  - a = b (assign b to a)
  - Performs widening conversion implicitly, if needed (see later)
- Primitive assignment
  - Assign LHS variable a bitwise copy of the RHS value

```
int a = 100;
int b = 200;

b = 42;
a = b;
```

- Reference assignment:
  - Assign LHS variable a reference to the RHS object

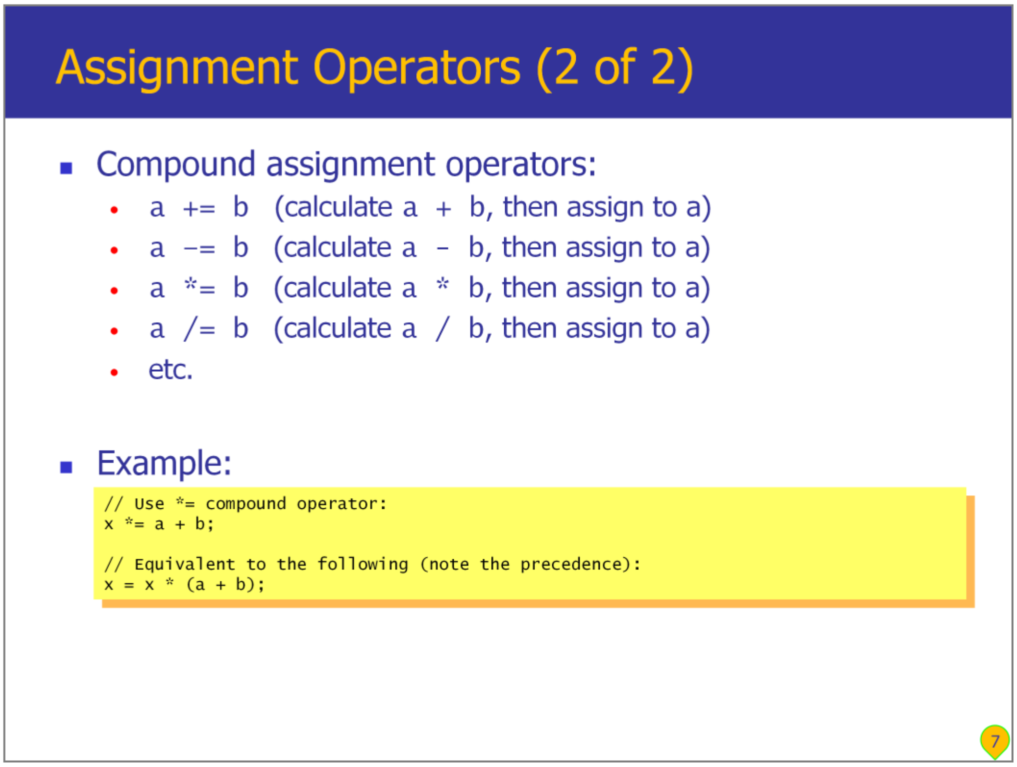
```
File file1 = new File("somefile.txt");
File file2 = new File("anotherfile.txt");

file1 = file2;
```

6

The assignment operator assigns the operand on the right-hand-side to the variable on the left-hand-side.

If you assign primitive types, e.g. integers, floats etc., the assignment operator copies the value on the right-hand-side to the variable on the left-hand-side. This is just a copy; the variables are still independent of each other.

If you assign object references, the assignment operator causes both object references to refer to the same object in memory. For example, in the second code box in the slide, `file1` and `file2` both point to the same `File` object in memory.



Assignment Operators (2 of 2)

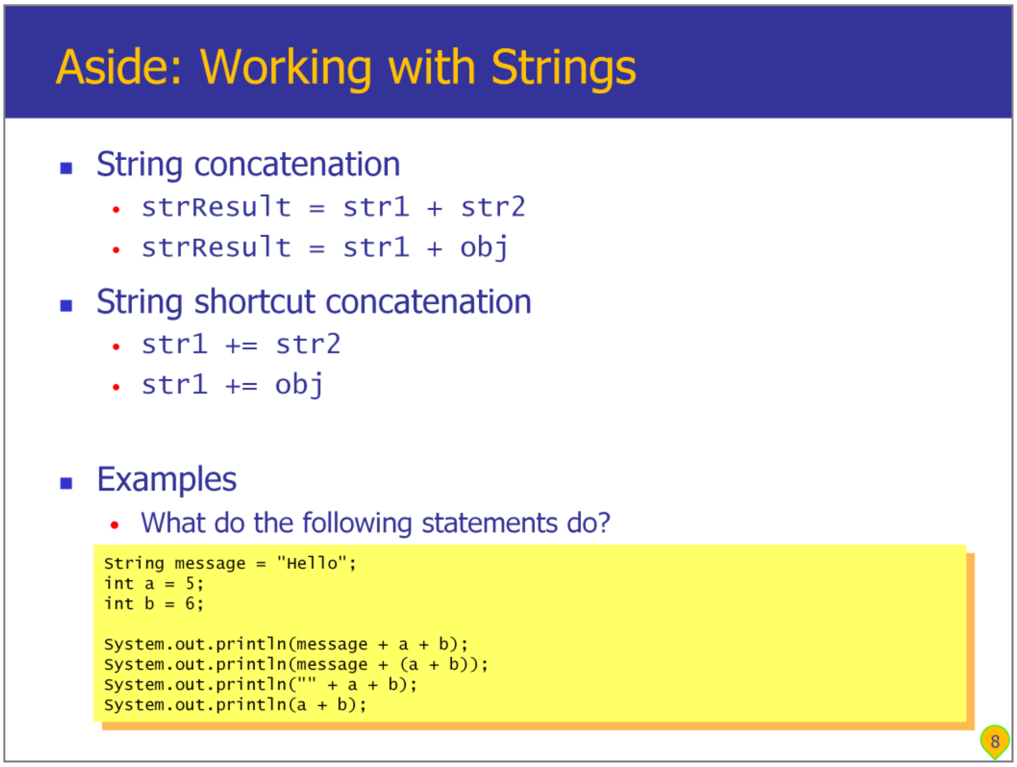
- Compound assignment operators:
  - a += b (calculate a + b, then assign to a)
  - a -= b (calculate a - b, then assign to a)
  - a *= b (calculate a * b, then assign to a)
  - a /= b (calculate a / b, then assign to a)
  - etc.
- Example:

```
// Use *= compound operator:
x *= a + b;

// Equivalent to the following (note the precedence):
x = x * (a + b);
```

7

Most binary operators (e.g. +, -, etc.) have a corresponding compound assignment operator (e.g. +=, -=, etc.). The compound assignment operators are equivalent to adding, subtracting etc. a value from a variable, and then storing the result back in the same variable.

The example in the slide shows how to use compound assignment operators, and how they equate to long-hand syntax.



Aside: Working with Strings

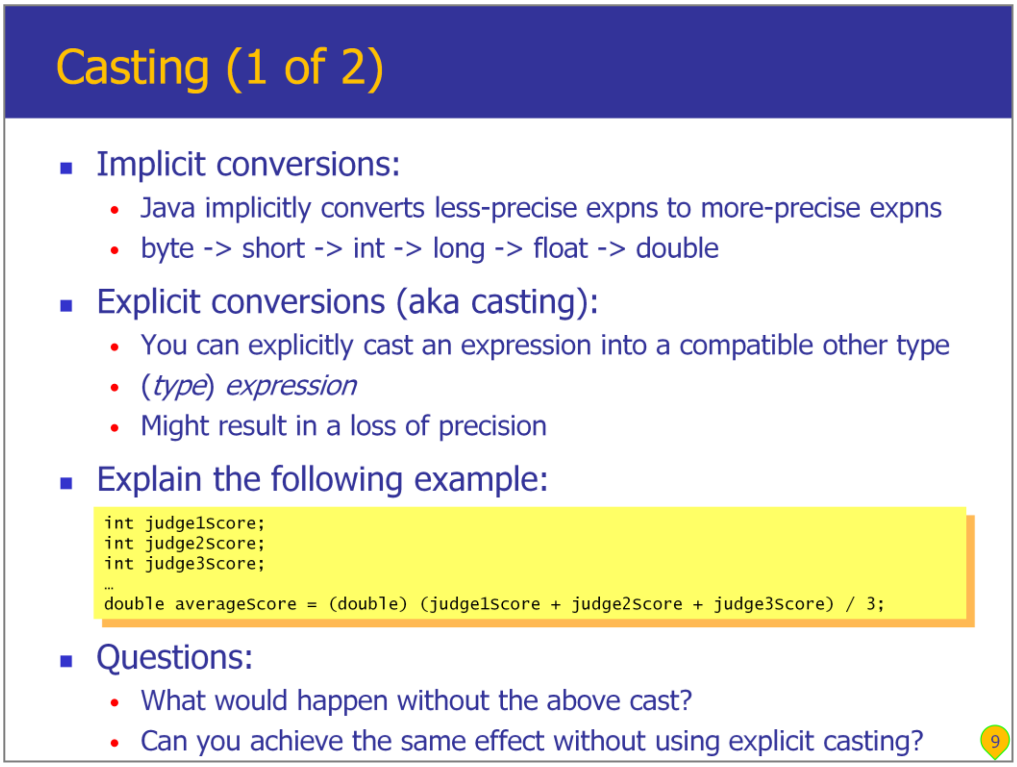
- String concatenation
  - `strResult = str1 + str2`
  - `strResult = str1 + obj`
- String shortcut concatenation
  - `str1 += str2`
  - `str1 += obj`
- Examples
  - What do the following statements do?

```
String message = "Hello";
int a = 5;
int b = 6;

System.out.println(message + a + b);
System.out.println(message + (a + b));
System.out.println("" + a + b);
System.out.println(a + b);
```

8

The previous slide introduced compound assignment operators. While we're on the subject, it's worth spending a moment considering how the += operator works in conjunction with strings.

Basically, `String` implements += as a concatenation mechanism. The += operator appends the right-hand-side string to the current string. Note that this is quite inefficient - you might prefer to use the `StringBuilder` class instead.



Casting (1 of 2)

- Implicit conversions:
  - Java implicitly converts less-precise expns to more-precise expns
  - byte -> short -> int -> long -> float -> double
- Explicit conversions (aka casting):
  - You can explicitly cast an expression into a compatible other type
  - (*type*) *expression*
  - Might result in a loss of precision
- Explain the following example:

```
int judge1Score;
int judge2Score;
int judge3Score;
…
double averageScore = (double) (judge1Score + judge2Score + judge3Score) / 3;
```

- Questions:
  - What would happen without the above cast?
  - Can you achieve the same effect without using explicit casting?

9

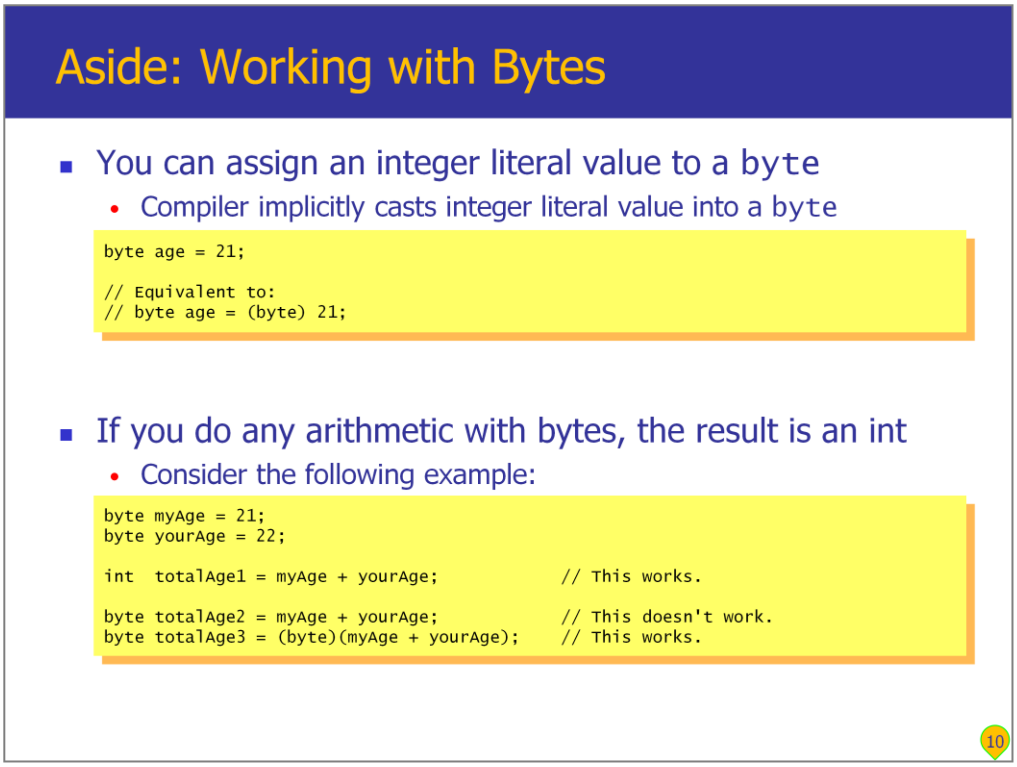
Casting is a mechanism that allows you to tell the compiler to interpret an expression as a different data type temporarily. For example, the following code snippet tells the compiler to temporarily treat `myAge` as a `double`, in order to perform floating-point arithmetic:

```
int myAge = 21;
double halfMyAge = (double)myAge / 2;
```

When you cast an expression, Java creates a copy of the original value in the specified type. The original value is not changed in any way.

In most cases, you won't need to perform casting. Java is quite capable of performing widening conversions implicitly (e.g. convert a `short` to an `int`, convert an `int` to a `long`, etc.). It's only when you need to perform a narrowing conversion (e.g. convert an `int` to a `short`, convert a `long` to an `int`, etc.) that you need an explicit cast.

You should not need to perform many casts in Java. If you continually find yourself having to cast a variable to a different type, then maybe you should have declared the variable with that type in the first place.



Aside: Working with Bytes

- You can assign an integer literal value to a `byte`
  - Compiler implicitly casts integer literal value into a `byte`

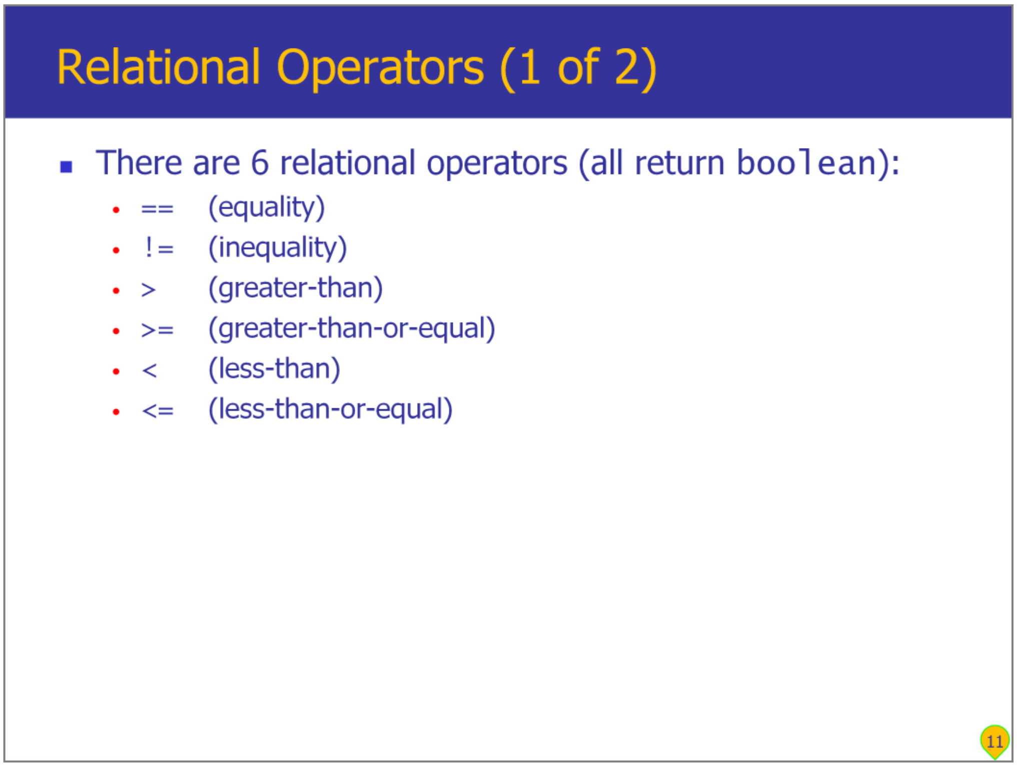
```
byte age = 21;

// Equivalent to:
// byte age = (byte) 21;
```

- If you do any arithmetic with bytes, the result is an int
  - Consider the following example:

```
byte myAge = 21;
byte yourAge = 22;

int  totalAge1 = myAge + yourAge;            // This works.

byte totalAge2 = myAge + yourAge;            // This doesn't work.
byte totalAge3 = (byte)(myAge + yourAge);    // This works.
```

10

This slide describes a particular scenario involving the `byte` data type. It's rather unlikely you'll hit this scenario in your code, unless you choose to use `byte` variables to hold very small integer values.



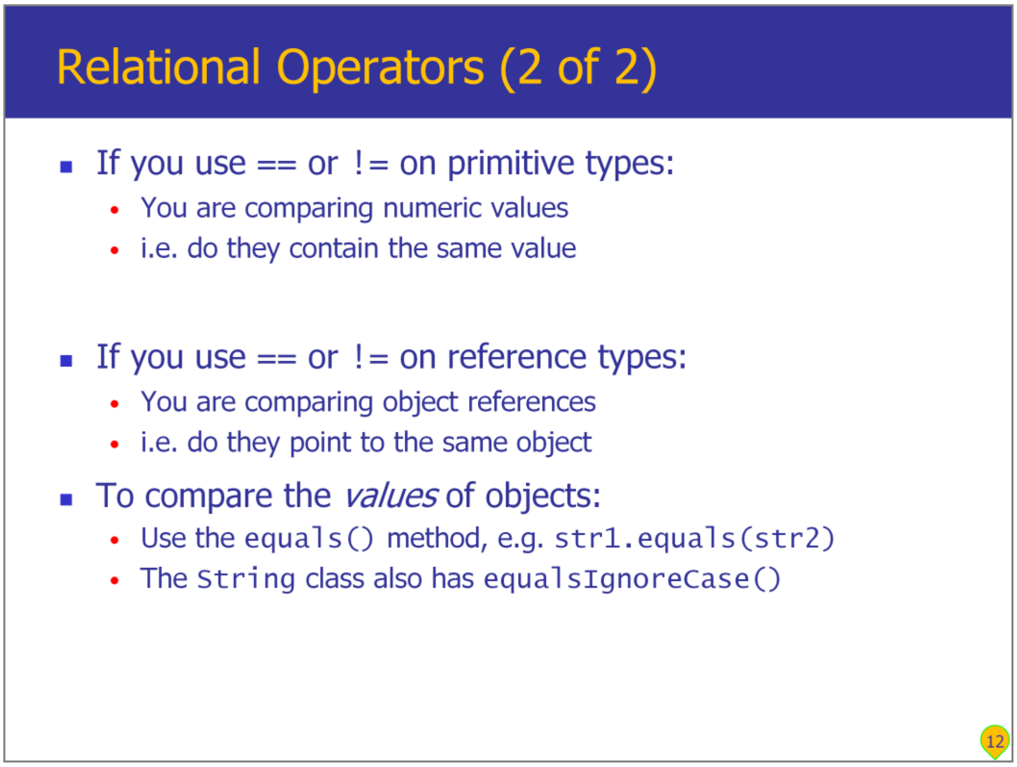
Relational Operators (1 of 2)

- There are 6 relational operators (all return `boolean`):
  - `==` (equality)
  - `!=` (inequality)
  - `>` (greater-than)
  - `>=` (greater-than-or-equal)
  - `<` (less-than)
  - `<=` (less-than-or-equal)

11

Java provides 6 relational operators, as listed in the slide. Note that the equality test is == (not = as in VB, or === as in JavaScript).

Note that Java also has an `instanceof` keyword. This is an operator that allows you to test whether a variable points to a particular type of object. This is an inheritance-related mechanism, so we'll defer discussion until the Inheritance chapter.



Relational Operators (2 of 2)

- If you use == or != on primitive types:
  - You are comparing numeric values
  - i.e. do they contain the same value
- If you use == or != on reference types:
  - You are comparing object references
  - i.e. do they point to the same object
- To compare the *values* of objects:
  - Use the `equals()` method, e.g. `str1.equals(str2)`
  - The `String` class also has `equalsIgnoreCase()`

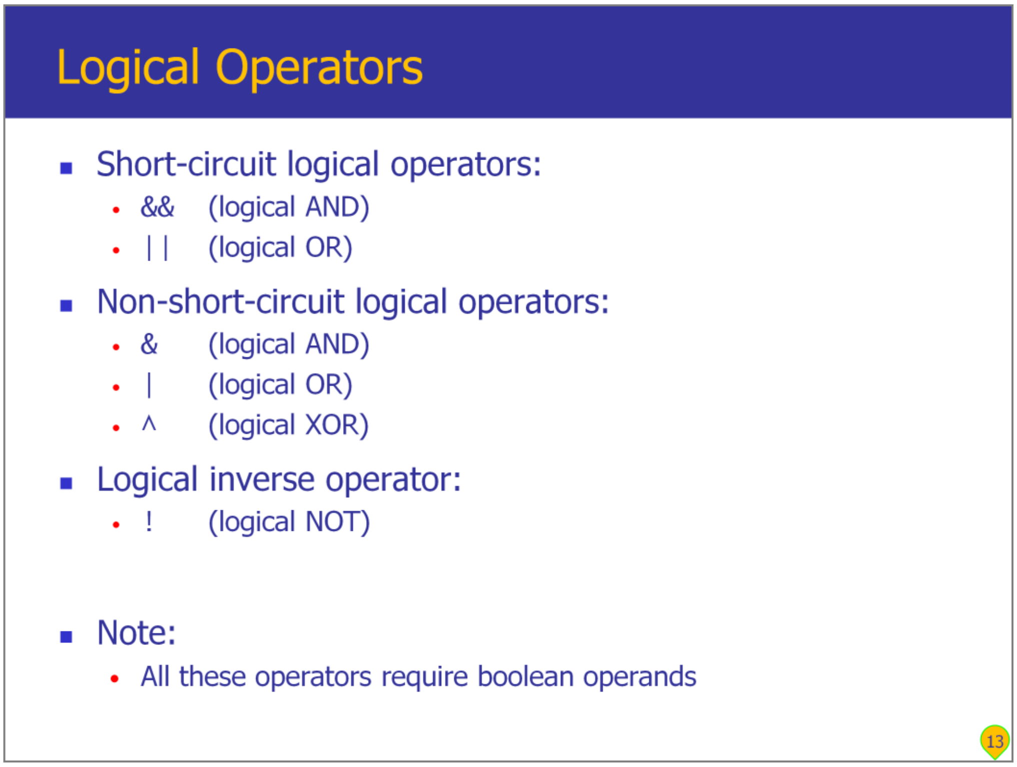
12

In addition to the points on the slide, note how equality works for wrapper classes such as `Integer` and `Double`:

- Wrapper classes work with != and ==
- Wrapper classes also work with `equals()`
- If you compare a wrapped object with a primitive, the wrapped object is automatically unboxed, and primitive-primitive comparison takes place (see later for boxing/unboxing)

Also note how equality for enums (see later for a full discussion of enums):

- Enums work with != and ==
- Enums also work with `equals()`



Logical Operators

- Short-circuit logical operators:
  - && (logical AND)
  - || (logical OR)
- Non-short-circuit logical operators:
  - & (logical AND)
  - | (logical OR)
  - ^ (logical XOR)
- Logical inverse operator:
  - ! (logical NOT)
- Note:
  - All these operators require boolean operands

13

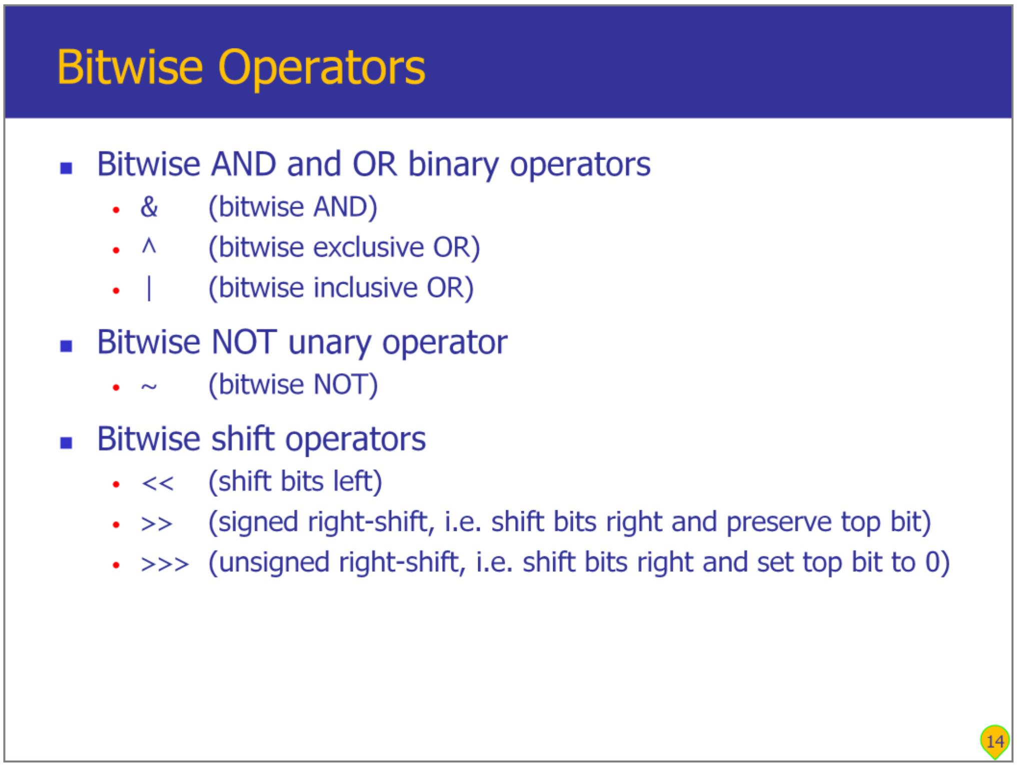
Java allows you to combine multiple boolean expressions together, by using the logical operators shown in the slide. Note the following points:

- The short-circuit logical operators do not perform the second test if the first test has already established the final outcome. For example, in the following code snippet, if `test1` evaluates to `false`, then `test2` will not be evaluated:

  ```
  if (test1 && test2)
  ```

- The non-short-circuit logical operators do perform the second test, even if the first test has already established the final outcome. For example, in the following code snippet, even if `test1` evaluates to `false`, `test2` will still always be evaluated. This is significant if `test2` has some side-effects, such as a function call:

  ```
  if (test1 & test2)
  ```

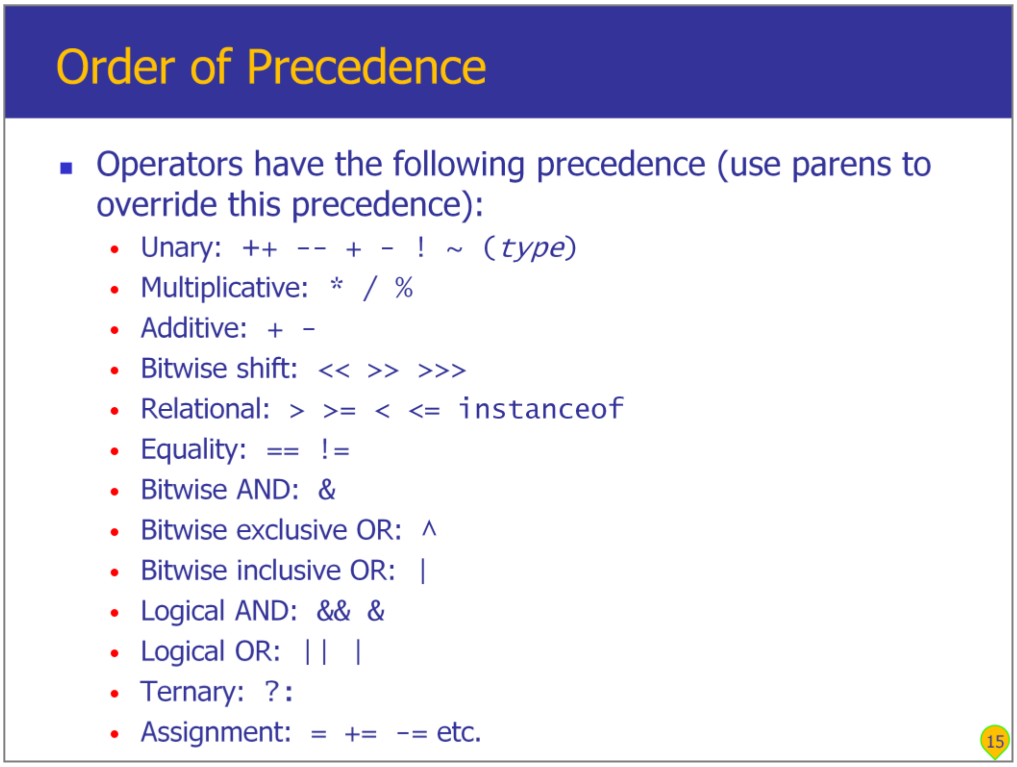


Bitwise Operators

- Bitwise AND and OR binary operators
  - & (bitwise AND)
  - ^ (bitwise exclusive OR)
  - | (bitwise inclusive OR)
- Bitwise NOT unary operator
  - ~ (bitwise NOT)
- Bitwise shift operators
  - << (shift bits left)
  - >> (signed right-shift, i.e. shift bits right and preserve top bit)
  - >>> (unsigned right-shift, i.e. shift bits right and set top bit to 0)

14

Java allows you to manipulate individual bits in integral values. This is a bit of a niche sport - the majority of commercial applications don't need to worry about individual bits. However, if you do need to perform bit-handling operations, you'll be familiar with the kinds of facilities provided by the bitwise operators in Java.



Order of Precedence

- Operators have the following precedence (use parens to override this precedence):
  - Unary: `++ -- + - ! ~ (`*`type`*`)`
  - Multiplicative: `* / %`
  - Additive: `+ -`
  - Bitwise shift: `<< >> >>>`
  - Relational: `> >= < <= instanceof`
  - Equality: `== !=`
  - Bitwise AND: `&`
  - Bitwise exclusive OR: `^`
  - Bitwise inclusive OR: `|`
  - Logical AND: `&& &`
  - Logical OR: `|| |`
  - Ternary: `?:`
  - Assignment: `= += -=` etc.

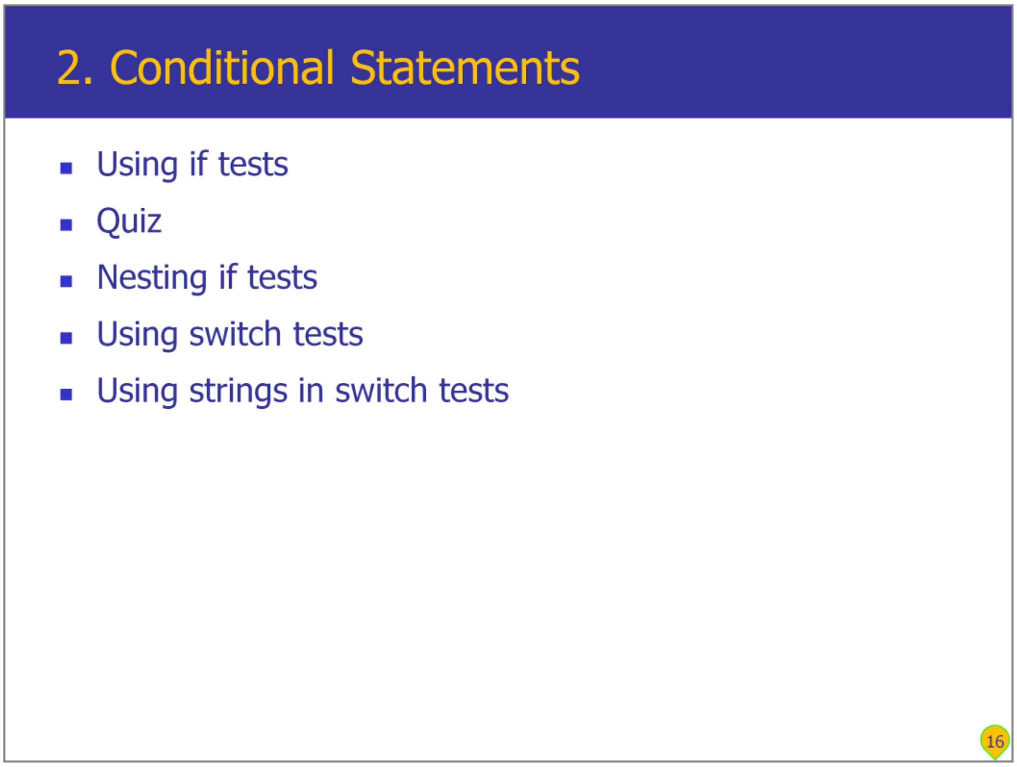
Java defines a precedence table, which identifies how complex expressions are parsed. Operators higher in the table have a higher precedence than operators lower in the table.

Consider the following example, for our French readers ☺:

```
int ninetyeight = 18 + (20 * 4);
```

This is equivalent to the following code. Note that the parentheses are optional, because * has a higher precedence than +:

```
int ninetyeight = 18 + 20 * 4;
```

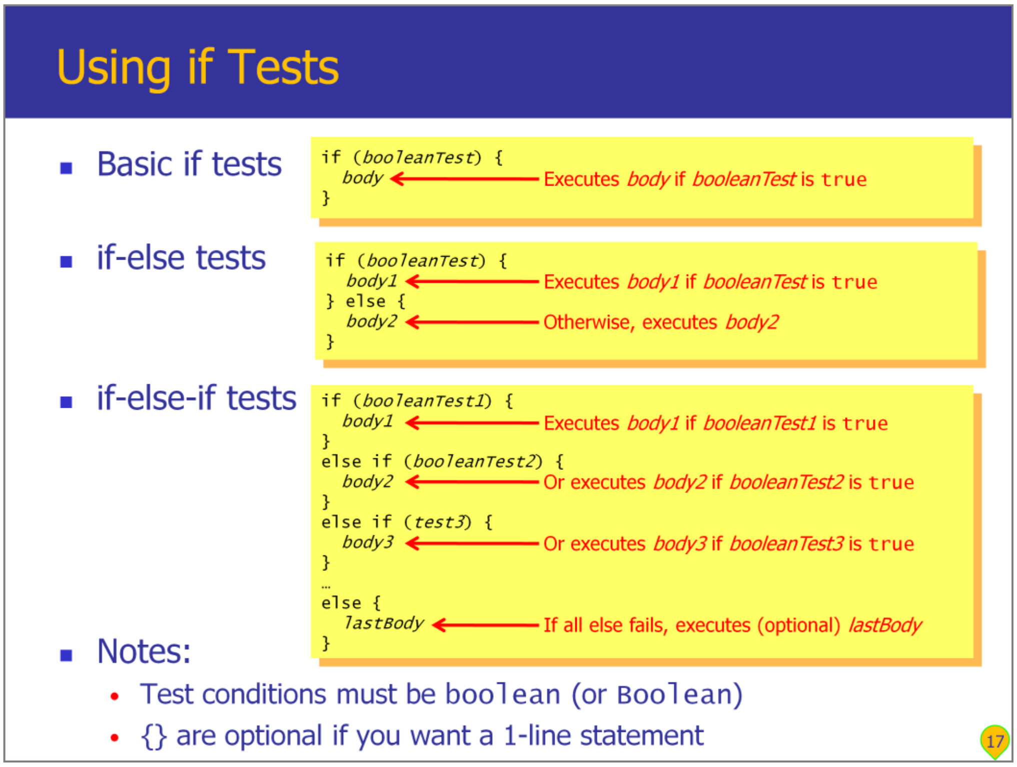


2. Conditional Statements

- Using if tests
- Quiz
- Nesting if tests
- Using switch tests
- Using strings in switch tests

16

This section takes you on a whirlwind tour of conditional statements in Java (i.e. `if` tests and `switch` statements). There are no great surprises here, but there are a few points to note…



Using if Tests

- Basic if tests

```
if (booleanTest) {
  body        <--- Executes body if booleanTest is true
}
```

- if-else tests

```
if (booleanTest) {
  body1       <--- Executes body1 if booleanTest is true
} else {
  body2       <--- Otherwise, executes body2
}
```

- if-else-if tests

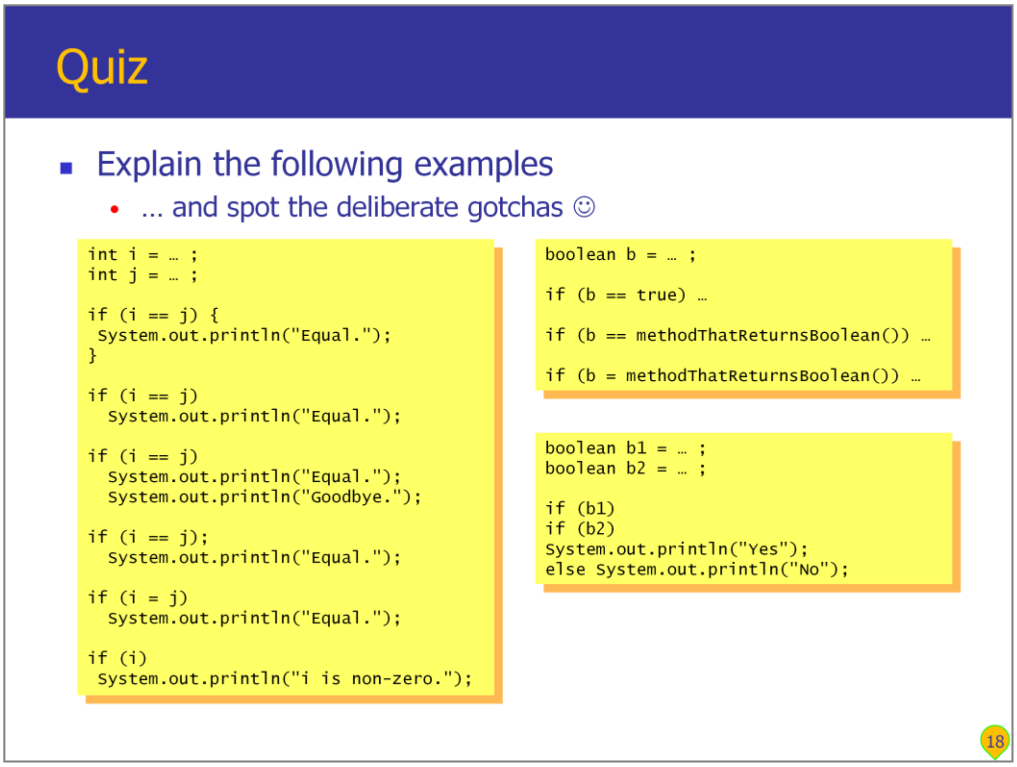
```
if (booleanTest1) {
  body1       <--- Executes body1 if booleanTest1 is true
}
else if (booleanTest2) {
  body2       <--- Or executes body2 if booleanTest2 is true
}
else if (test3) {
  body3       <--- Or executes body3 if booleanTest3 is true
}
…
else {
  lastBody    <--- If all else fails, executes (optional) lastBody
}
```

- Notes:
  - Test conditions must be `boolean` (or `Boolean`)
  - {} are optional if you want a 1-line statement

17

This slides describes all the rules for formulating if and if-else statements in Java.

The only point that we need to stress here is that you must specify a boolean condition - you can't use integers, object references, etc. directly. This is something for C/C++/JavaScript developers to bear in mind!



Quiz

- Explain the following examples
  - … and spot the deliberate gotchas ☺

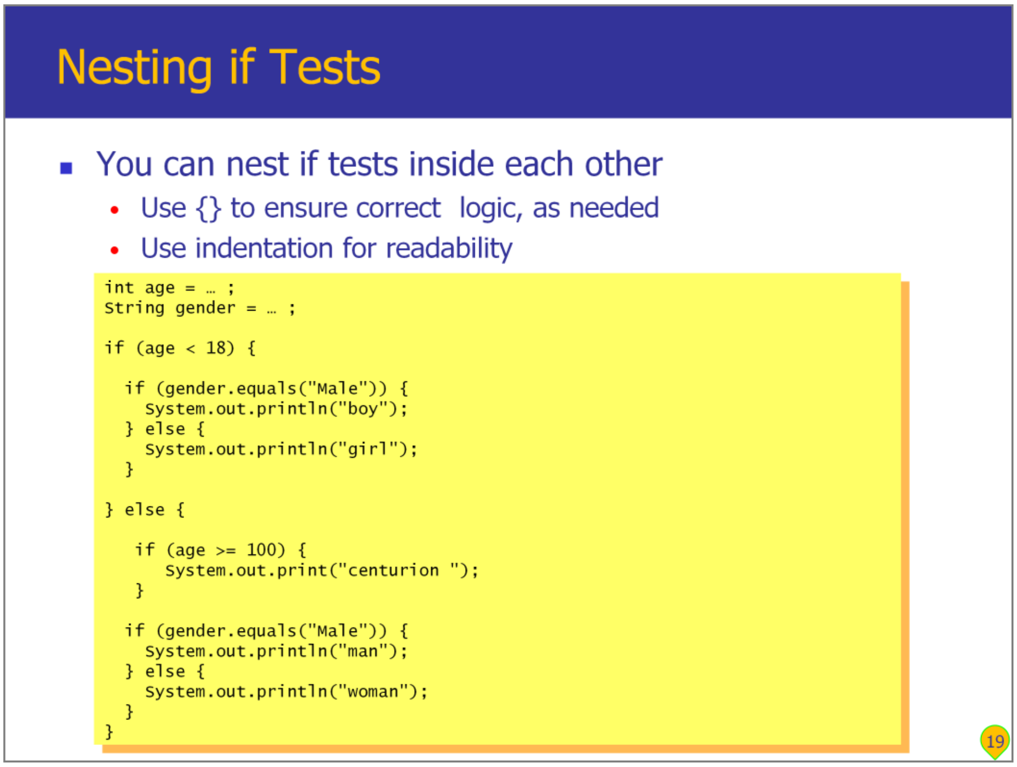
```
int i = … ;
int j = … ;

if (i == j) {
 System.out.println("Equal.");
}

if (i == j)
  System.out.println("Equal.");

if (i == j)
  System.out.println("Equal.");
  System.out.println("Goodbye.");

if (i == j);
  System.out.println("Equal.");

if (i = j)
  System.out.println("Equal.");

if (i)
 System.out.println("i is non-zero.");
```

```
boolean b = … ;

if (b == true) …

if (b == methodThatReturnsBoolean()) …

if (b = methodThatReturnsBoolean()) …
```

```
boolean b1 = … ;
boolean b2 = … ;

if (b1)
if (b2)
System.out.println("Yes");
else System.out.println("No");
```

18

This slide contains some deliberate syntax errors, logical bloopers, and bad style. Analyse the code carefully and see if you can spot the problems…



Nesting if Tests

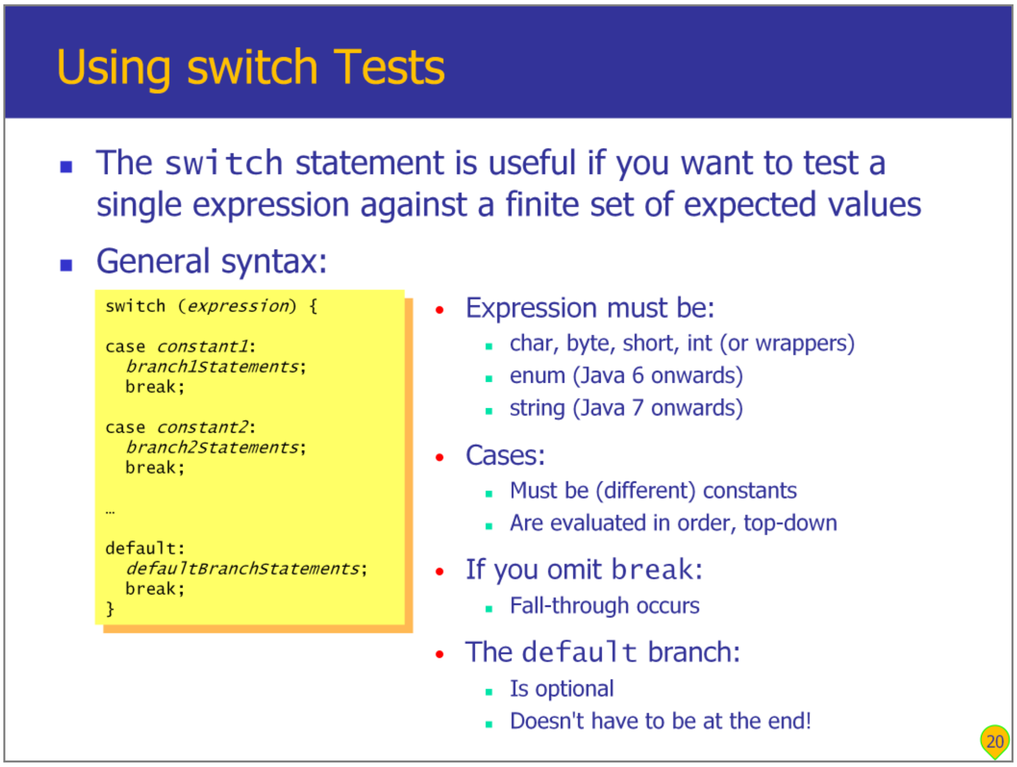
- You can nest if tests inside each other
  - Use {} to ensure correct logic, as needed
  - Use indentation for readability

```
int age = … ;
String gender = … ;

if (age < 18) {

  if (gender.equals("Male")) {
    System.out.println("boy");
  } else {
    System.out.println("girl");
  }

} else {

   if (age >= 100) {
      System.out.print("centurion ");
   }

  if (gender.equals("Male")) {
    System.out.println("man");
  } else {
    System.out.println("woman");
  }
}
```

19

As you might expect, you can nest `if` statements inside other `if` statements. Here are some recommendations:

- You might want to include `{}` even if they aren't strictly needed, in order to emphasize the logical flow through the code.
- Don't overdo it. The code in the slide is already getting quite tricky. It might be a better idea to shovel some of this logic off into a separate method, to make your intentions clearer.



Using switch Tests

- The `switch` statement is useful if you want to test a single expression against a finite set of expected values
- General syntax:

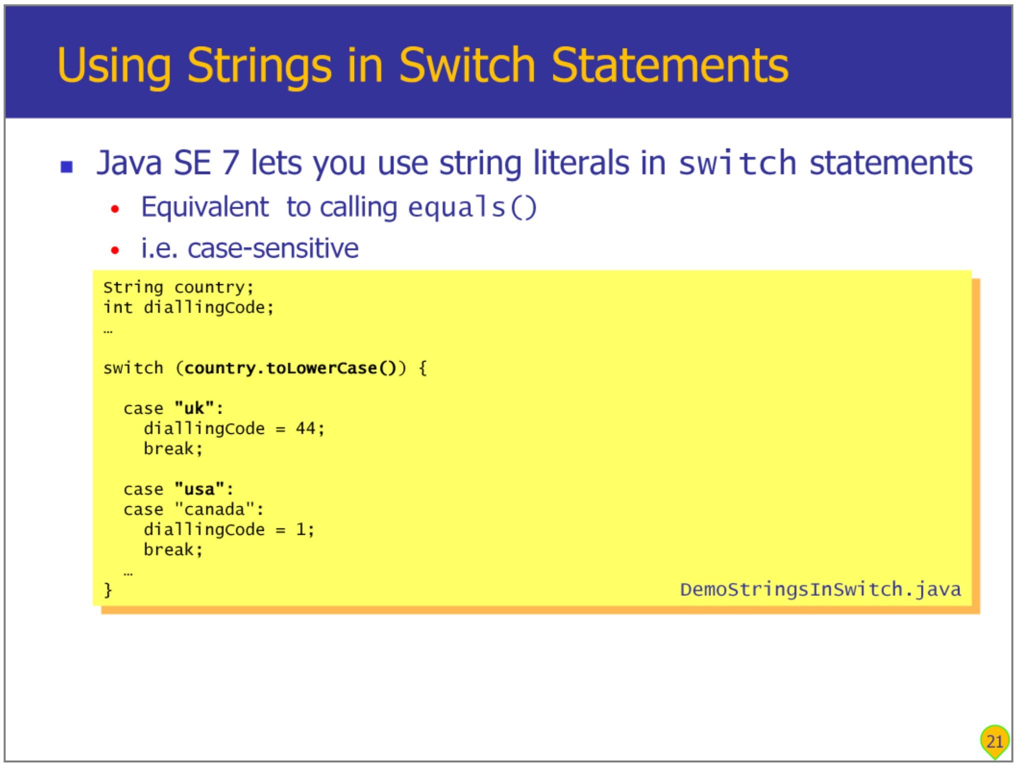
```
switch (expression) {

case constant1:
  branch1Statements;
  break;

case constant2:
  branch2Statements;
  break;

…

default:
  defaultBranchStatements;
  break;
}
```

- Expression must be:
  - char, byte, short, int (or wrappers)
  - enum (Java 6 onwards)
  - string (Java 7 onwards)
- Cases:
  - Must be (different) constants
  - Are evaluated in order, top-down
- If you omit `break`:
  - Fall-through occurs
- The `default` branch:
  - Is optional
  - Doesn't have to be at the end!

20

The `switch` construct is useful if you want to test a single integral expression against a finite and discrete set of possible values. Java 7 onwards also allows you to switch on string variables.

The pseudo-example on the slide shows the basic syntax and summarizes all the rules. It's not the cleanest construct in Java, but it has its place!



Using Strings in Switch Statements

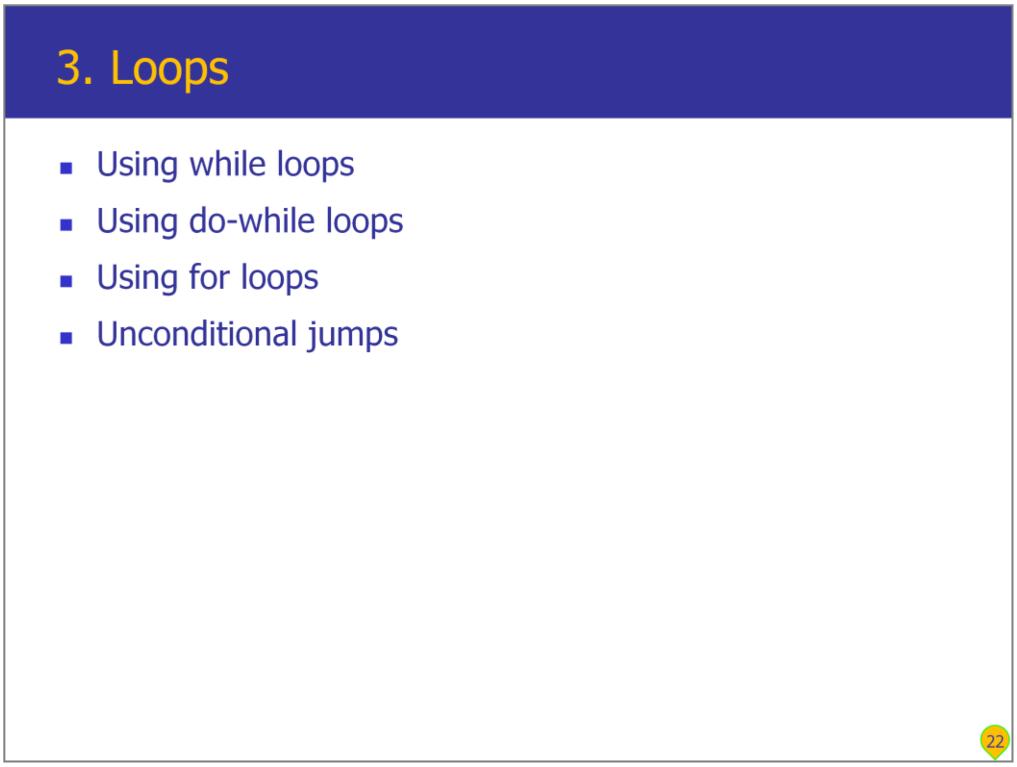
- Java SE 7 lets you use string literals in `switch` statements
  - Equivalent to calling `equals()`
  - i.e. case-sensitive

```
String country;
int diallingCode;
…

switch (country.toLowerCase()) {

  case "uk":
    diallingCode = 44;
    break;

  case "usa":
  case "canada":
    diallingCode = 1;
    break;
  …
}
```

DemoStringsInSwitch.java

21

Java SE 7 allows you to use string literals (e.g. "hello") in switch statements. Java does an `equals()` test to compare for matches.

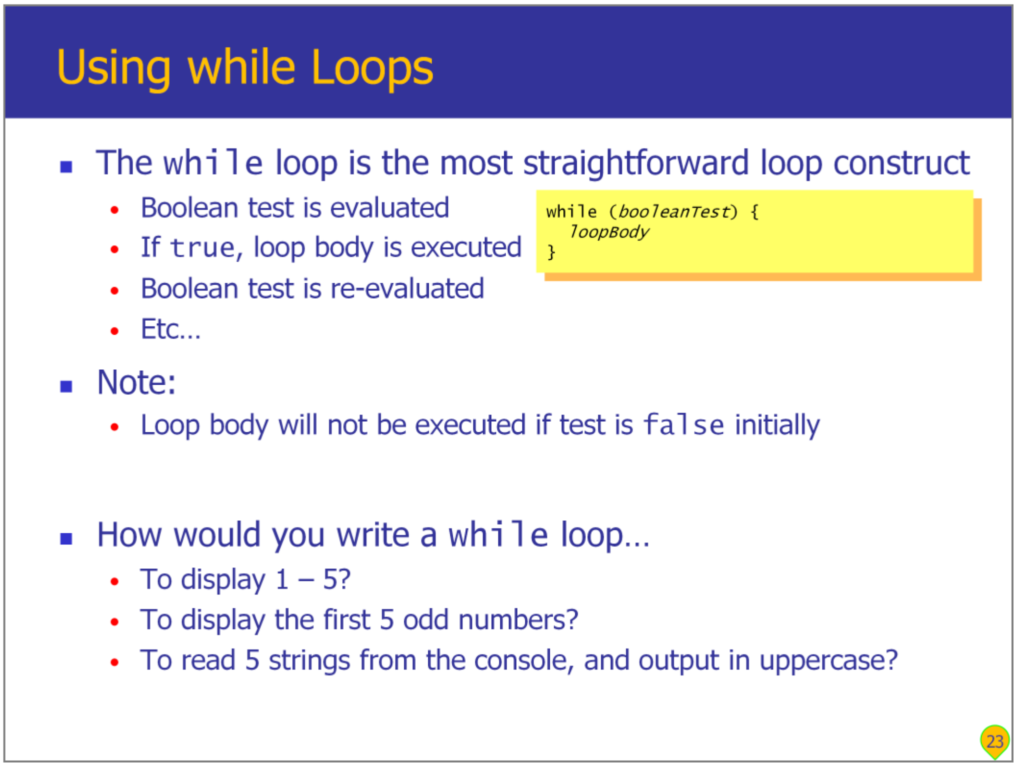


3. Loops

- Using while loops
- Using do-while loops
- Using for loops
- Unconditional jumps

22

This section describes how to write loops. There are three ways to do this, and we'll show each construct in turn. We'll also describe how to perform unconditional jumps by using the `break` and `continue` keywords.



Using while Loops

- The `while` loop is the most straightforward loop construct
  - Boolean test is evaluated
  - If `true`, loop body is executed
  - Boolean test is re-evaluated
  - Etc…

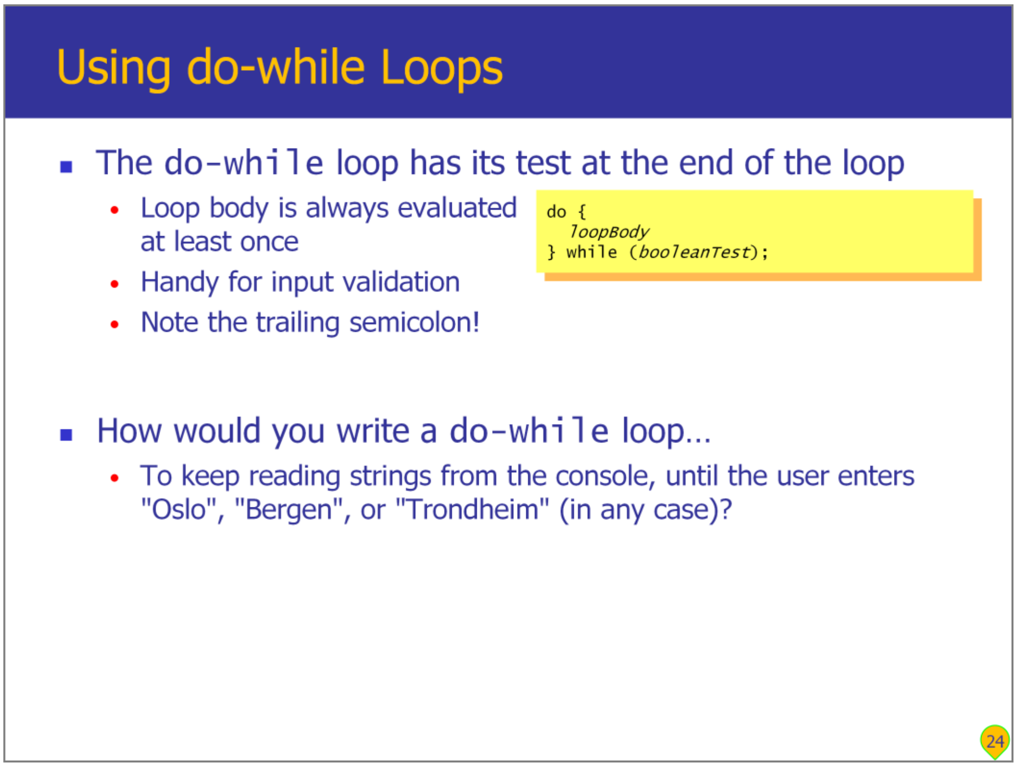
```
while (booleanTest) {
   loopBody
}
```

- Note:
  - Loop body will not be executed if test is `false` initially
- How would you write a `while` loop…
  - To display 1 – 5?
  - To display the first 5 odd numbers?
  - To read 5 strings from the console, and output in uppercase?

23

The most fundamental loop construct in Java is the `while` loop. Note that the condition is testing for truth (there's no equivalent of the "repeat until" construct that you find in Visual Basic).

Have a think about the questions in the slide. How would you implement these algorithms by using a `while` loop?



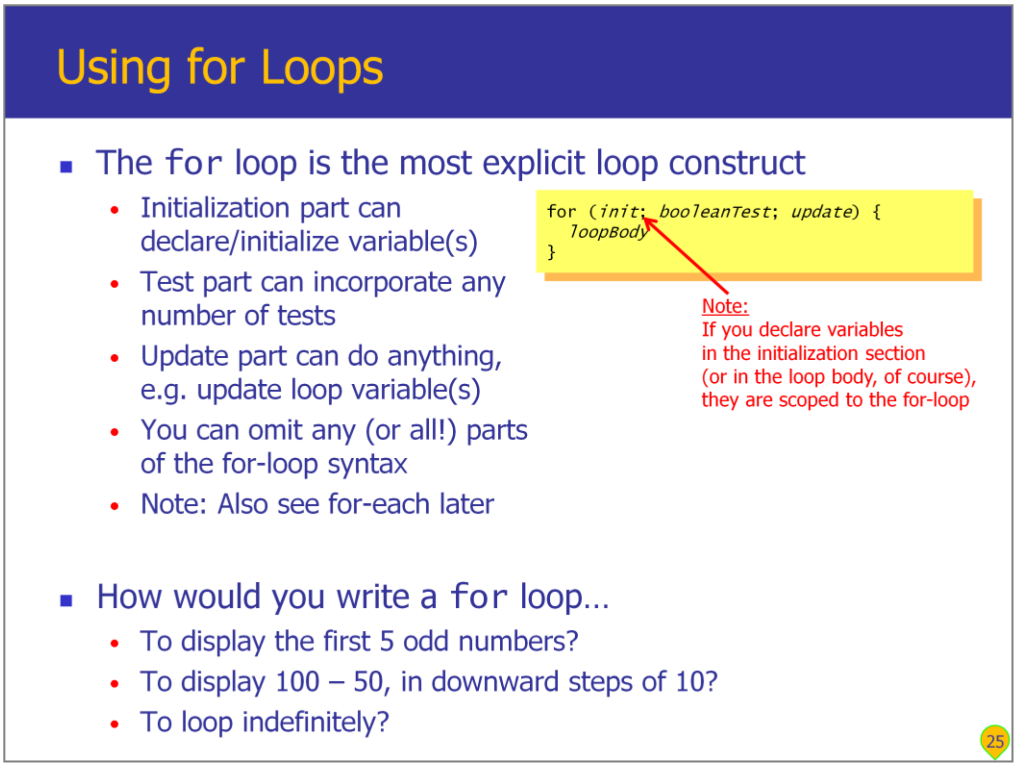
Using do-while Loops

- The `do-while` loop has its test at the end of the loop
  - Loop body is always evaluated at least once
  - Handy for input validation
  - Note the trailing semicolon!

```
do {
   loopBody
} while (booleanTest);
```

- How would you write a `do-while` loop…
  - To keep reading strings from the console, until the user enters "Oslo", "Bergen", or "Trondheim" (in any case)?

The `do-while` loop is similar to the `while` loop, except that with `do-while` the test is performed at the end of each iteration rather than at the start. This means that a `do-while` loop ensures at least one loop iteration will occur, before the test condition is encountered for the first time.



Using for Loops

- The `for` loop is the most explicit loop construct
  - Initialization part can declare/initialize variable(s)
  - Test part can incorporate any number of tests
  - Update part can do anything, e.g. update loop variable(s)
  - You can omit any (or all!) parts of the for-loop syntax
  - Note: Also see for-each later

```
for (init; booleanTest; update) {
   loopBody
}
```

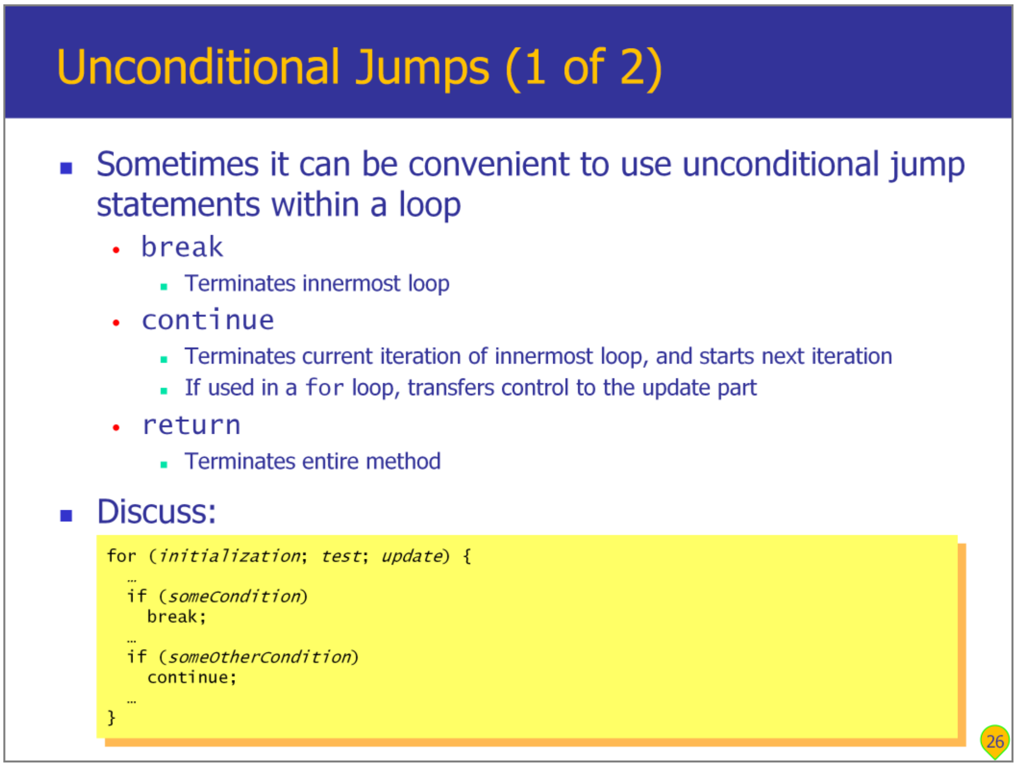
Note:
If you declare variables in the initialization section (or in the loop body, of course), they are scoped to the for-loop

- How would you write a `for` loop…
  - To display the first 5 odd numbers?
  - To display 100 – 50, in downward steps of 10?
  - To loop indefinitely?

25

The `for` loop is probably the most widely-used loop mechanism in Java, because it gets all the administrative tasks out of the way right at the start of the statement (i.e. loop variable initialization, test condition, and update logic).

Make sure you understand exactly how it works. The notes in the slide give full information.



Unconditional Jumps (1 of 2)

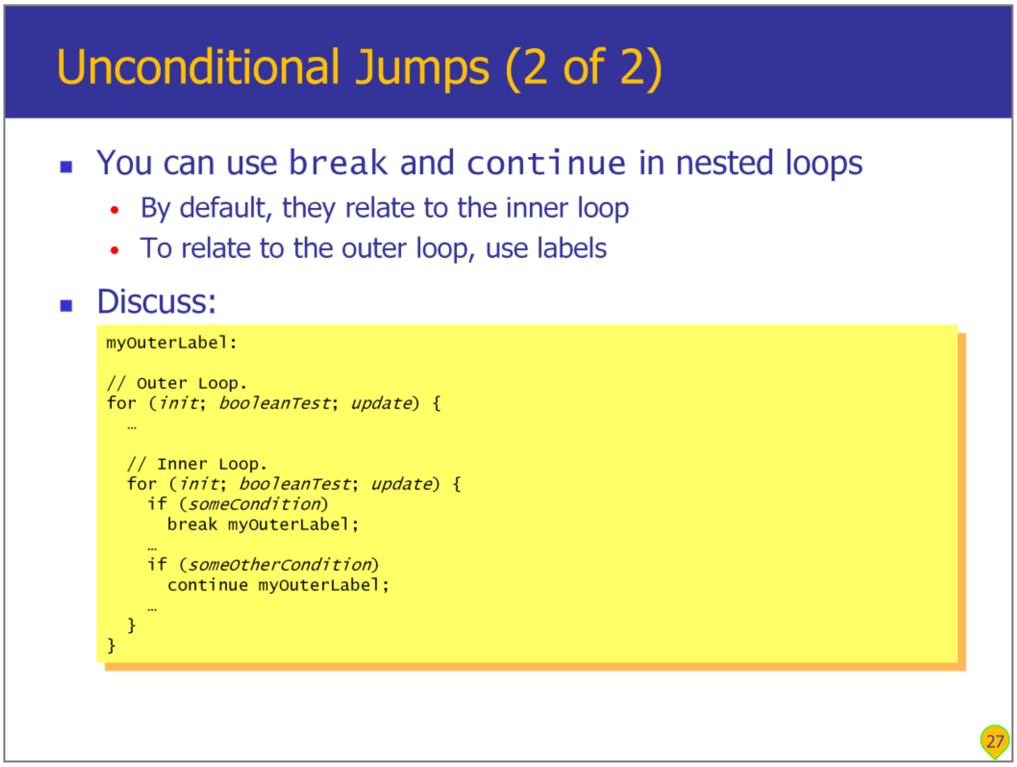
- Sometimes it can be convenient to use unconditional jump statements within a loop
  - `break`
    - Terminates innermost loop
  - `continue`
    - Terminates current iteration of innermost loop, and starts next iteration
    - If used in a `for` loop, transfers control to the update part
  - `return`
    - Terminates entire method
- Discuss:

```
for (initialization; test; update) {
  …
  if (someCondition)
    break;
  …
  if (someOtherCondition)
    continue;
  …
}
```

26

The `break` statement terminates a loop immediately. This is useful if you decide you've found the record you were looking for, or if something has gone horribly wrong and there's no point carrying on with the loop.

The `continue` statement abandons the current loop iteration, and resumes at the top of the loop ready for the next iteration. This is useful if you decide the current record is invalid or irrelevant, and you want to skip it and move on to the next record.



Unconditional Jumps (2 of 2)

- You can use `break` and `continue` in nested loops
  - By default, they relate to the inner loop
  - To relate to the outer loop, use labels
- Discuss:

```
myOuterLabel:

// Outer Loop.
for (init; booleanTest; update) {
  …

  // Inner Loop.
  for (init; booleanTest; update) {
    if (someCondition)
      break myOuterLabel;
    …
    if (someOtherCondition)
      continue myOuterLabel;
    …
  }
}
```

27

If you have nested loops, the `break` and `continue` statements just pertain to the inner loop by default. If you need to break out of the outer loop, or continue the next iteration of the outer loop, you can use labelled `break` and `continue` statements as shown in this slide.

Any Questions?

28