


CSC 1051 Algorithms & Data Structures I

Control Structures & Data Representation



CONDITIONALS & CONTROL FLOW

If Statement

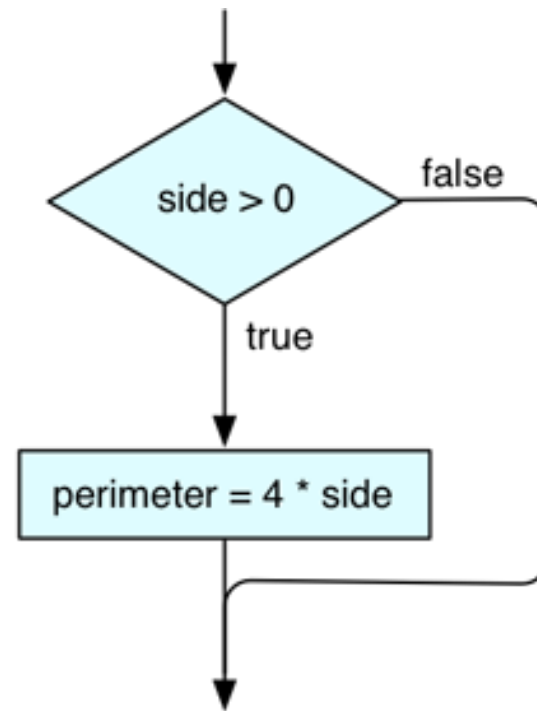
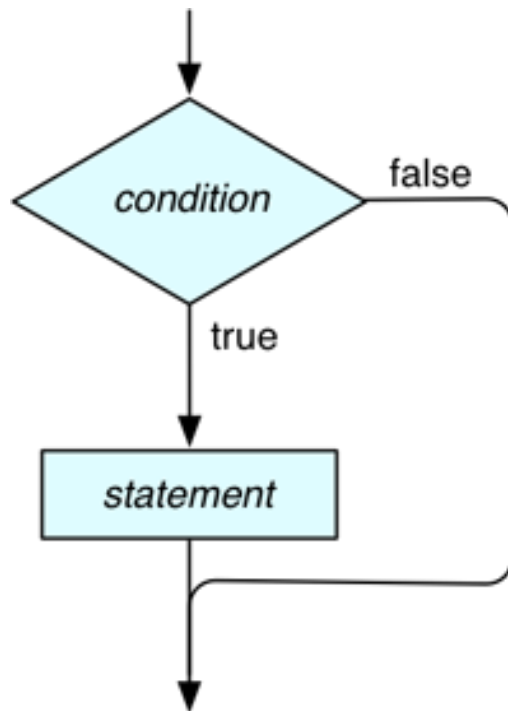
- An **if statement** is used to make a decision
- In particular, it is used to determine which program statement to execute next
- It evaluates a boolean **condition** and only executes the body of the if statement if the condition is true

```
if (side > 0)
    perimeter = 4 * side;
```

The if statement is sometimes referred to as a **selection** statement or a **conditional** statement

If Statement

- The logic of an if statement can be represented as a flowchart



If Statement

- Another example:

```
if (total < MAX)
    System.out.println("The total is within limits.");
```

The body of an if statement is often a **block statement** enclosed in braces

```
if (side > 0)
{
    perimeter = 4 * side;
    System.out.println("The perimeter is " + perimeter);
}
```

If Statement

- Syntactically, the body of an if statement is a single statement
- Use a **block statement** if needed



```
if (side > 0)
    perimeter = 4 * side;
    System.out.println("The perimeter is " + perimeter);
```



The `println` statement is not part of the `if statement` body, despite the indentation

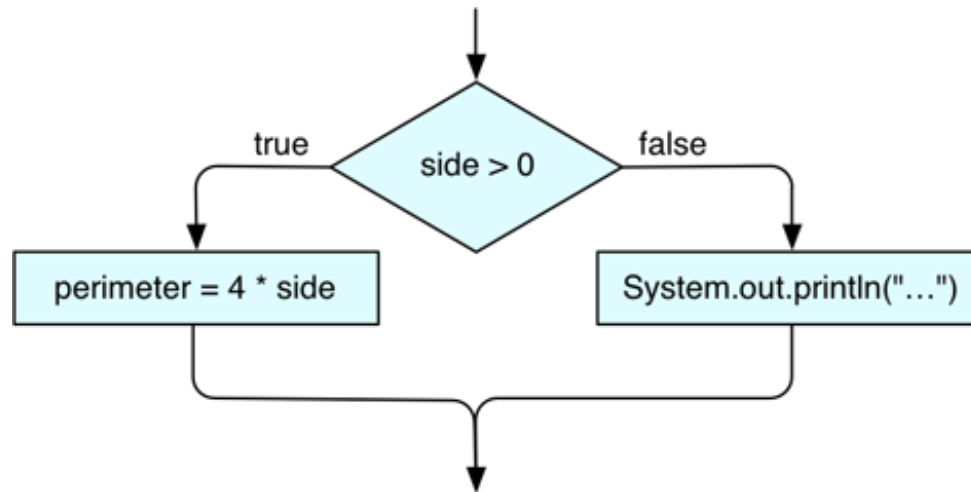
Some programming conventions recommend always using **braces**



If Statement

- An optional **else clause** specifies what should be done if the condition is not true

```
if (side > 0)
    perimeter = 4 * side;
else
    System.out.println("Invalid side value.");
```



If Statement

- The body of the if, the else, or both can be **block statements**, surrounded by braces.

```
if (side > 0)
{
    perimeter = 4 * side;
    System.out.println("The perimeter is " + perimeter);
}
else
    System.out.println("Invalid side value.");
```


If Statement

- A **nested if statement** occurs when the body of an if statement is itself an if statement


```
if (pendingOrders > 0)
    if (stock < MIN_STOCK)
        System.out.println("Reorder stock.");
```

Both conditions must be true for the println statement to be executed

If Statement

- An else clause is matched to the closest unmatched if
- That creates potential for the **dangling else problem** 

```
if (total >= 0)
    if (result >= 0)
        System.out.println("Both are positive.");
else
    System.out.println("Total is negative");
```



Despite the indentation, the else is matched to the second (nested) if statement

If Statement

- Braces can be used to ensure the else is matched with the correct if statement

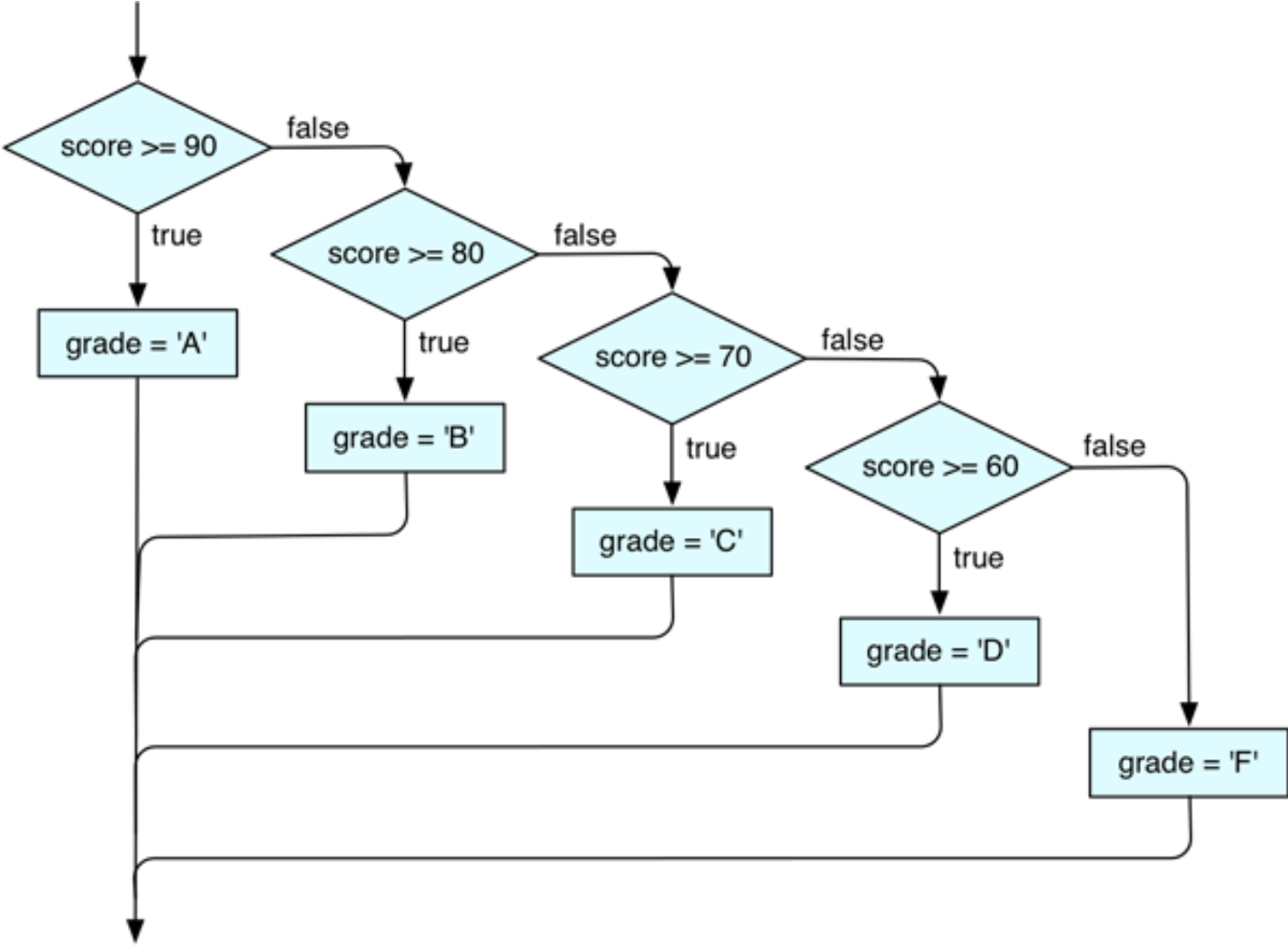
```
if (total >= 0)
{
    if (result >= 0)
        System.out.println("Both are positive.");
}
else
    System.out.println("Total is negative.");
```

If Statement

- A series of nested ifs are often formatted as follows to increase readability

```
if (score >= 90)
    grade = 'A';
else if (score >= 80)
    grade = 'B';
else if (score >= 70)
    grade = 'C';
else if (score >= 60)
    grade = 'D';
else
    grade = 'F';
```

If Statement



Example: Paycheck Calculator

- Walk-thru the Refactor topic [Example: Paycheck Calculator](#)

A Practical Use of an If Statement

- Let's say you have some number of things and you want to print a grammatically correct output statement. That's a perfect job for an if statement!

```
int count = 73;

if (count == 1)
    System.out.println(count + " gold bar");
else
    System.out.println(count + " gold bars");
```

This is a frequently needed thing to do. Remember it!

Why do we need Boolean Expressions?

```
public class GPACalculator
{
    public static void main(String[] args)
    {
        double qp, credits, gpa;
        Scanner scan = new Scanner(System.in);
        System.out.print ("Quality Points> ");
        qp = scan.nextDouble();
        System.out.print ("Credits> ");
        credits = scan.nextInt();

        // calculate GPA
        gpa = qp / credits;

        System.out.println ("\n\tGPA: " + gpa);
    }
}
```

What if credits is 0?

Quality Points> 22.5
Credits> 0

Runtime Error!

Boolean Expression to the Rescue!

We can prevent some badness by adding add logical conditions to protect certain statements. Here's the **pseudocode**:

If statement

```
If credits equals 0
    print "No GPA"
Else
    calculate GPA by formula
    print GPA results
```

Boolean Expression

Boolean Expressions

A **boolean expression** is an expression that evaluates to either **true** or **false**

The result could be assigned to a boolean variable:

```
boolean countExceeded = count > MAX;
```

More often, a boolean expression is used as the condition of an **if statement** or a loop

```
if (count > MAX)
    System.out.println("Maximum count exceeded.");
```

Boolean Expressions

A **relational operator** tests the relationship between two values

Java Operator	Math Symbol	Description	Example
<code>==</code>	<code>=</code>	Equal	<code>a == b</code>
<code>!=</code>	<code>≠</code>	Not Equal	<code>a != b</code>
<code><</code>	<code><</code>	Less Than	<code>a < b</code>
<code><=</code>	<code>≤</code>	Less Than or Equal To	<code>a <= b</code>
<code>></code>	<code>></code>	Greater Than	<code>a > b</code>
<code>>=</code>	<code>≥</code>	Greater Than or Equal To	<code>a >= b</code>

Evaluate these Boolean expressions

Evaluate the values of expressions, if no value, write down the reasons:

Expression	Value
<code>1 == 1.0</code>	True, the value equals
<code>5 < 11/2</code>	False, 11/2 is 5, 5 equals 5
<code>'a' == 'a'</code>	True
<code>'a' < 'A'</code>	False, Unicode: 'a' == 97, 'A' = 65
<code>"a" > "A"</code>	No value, String type cannot be applied
<code>3 >= 3</code>	True
<code>true != false</code>	True
<code>true > false</code>	No value, boolean type cannot be applied

Boolean Expressions

The relational operators have a lower **precedence** than the arithmetic operators

```
if (total > localCount + globalCount)
    System.out.println("Total exceeds full count.");
```

The two numbers are added first, and the result is compared to the value of total

Boolean Expressions

Characters can be compared using the relational operators

A character is less than another if it comes before it (has a lower value) in the [Unicode](#) character set

Most of the relational operators **cannot** be applied to **objects**

To put strings in order, use the [compareTo](#) method

The equal to (==) and not equal to (!=) operators can be used on objects, but only to determine if two references point to the same object

Boolean Expressions

It's unwise to test two floating-point values for equality using the == operator

```
if (d1 == d2)
    System.out.println("They are equal.");
```

That will only be true if the underlying binary representation of d1 and d2 are exactly the same

If they are the results of calculations, there is a good chance they won't be equal (even if they are very close)

Boolean Expressions

For example:

```
double num = Math.sqrt(2) * Math.sqrt(2);  
  
System.out.println(num);  
  
if (num == 2.0)  
    System.out.println("Equal!");
```

```
2.0000000000000004
```

The string "Equal!" is not printed, because they aren't

Boolean Expressions

A better approach is to see if they are "close enough"

Set up a **tolerance value** and compare it to the difference between the two values

```
final double TOLERANCE = 1E-14; // 0.000000000000001
double num = Math.sqrt(2) * Math.sqrt(2);
if (Math.abs(num - 2.0) < TOLERANCE)
    System.out.println("Close enough!");
```

Boolean Operators

Multiple conditions can be combined using **boolean operators** to create more complex boolean expressions

For example, to determine if a value is in the range 1 to 100:

```
if (num >= 1 && num <= 100)
    System.out.println("In range.");
```

Boolean operators take boolean operands and produce boolean results

The **and operator** (&&) produces a true result if both of its operands are true

Boolean Operators

Java has four boolean operators

Operator	Name	Use	Result
!	Not	!X	true if X is false and false if X is true
&&	And	X && Y	true if both X and Y are true and false otherwise
	Or	X Y	true if X or Y or both are true and false otherwise
^	Exclusive Or	X ^ Y	true if X or Y (but not both) are true, and false otherwise

The boolean operators are sometimes called **logical operators**

The results of a boolean operator can be shown in a **truth table**, which shows all possible true/false combinations

Boolean Operators

The **not operator** (!) is a **unary operator** – it has only one operand

It negates (reverses) the truth value of it's operand

X	!X
false	true
true	false

```
if (!found)
    System.out.println("Keep looking.");
```

```
if (!name1.equals(name2))
    System.out.println("Not the same name.");
```

Boolean Operators

The **and operator** (&&) produces a true result only if both of its operands are true

X	Y	X && Y
false	false	false
false	true	false
true	false	false
true	true	true

```
if (num > 0 && num % 2 == 0)
    System.out.println("Positive and even.");
```

Boolean Operators

The **or operator** (||) produces a true result if either or both of its operands are true

X	Y	X Y
false	false	false
false	true	true
true	false	true
true	true	true

```
if (width > 20 || height > 50)
    System.out.println("Too big.");
```

Boolean Operators

The **exclusive or operator** (^) produces a true result if either of its operands are true but not both

X	Y	X ^ Y
false	false	false
false	true	true
true	false	true
true	true	false

```
if (chooseTea ^ chooseCoffee)
    System.out.println("Good choice.");
```

Boolean Operators

The `&&` and `||` operators are **short-circuited** – if the result can be determined by the first operand, the second is not evaluated

If the first operand to a `&&` operator is false, it doesn't matter what the second operand is

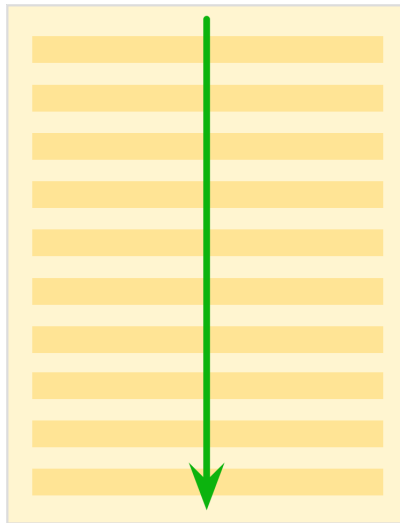
```
if (name != null && name.equals("Sam"))
    System.out.println("That's the guy.");
else
    System.out.println("It's not Sam.");
```

The name reference won't be used if it is null

REPETITION

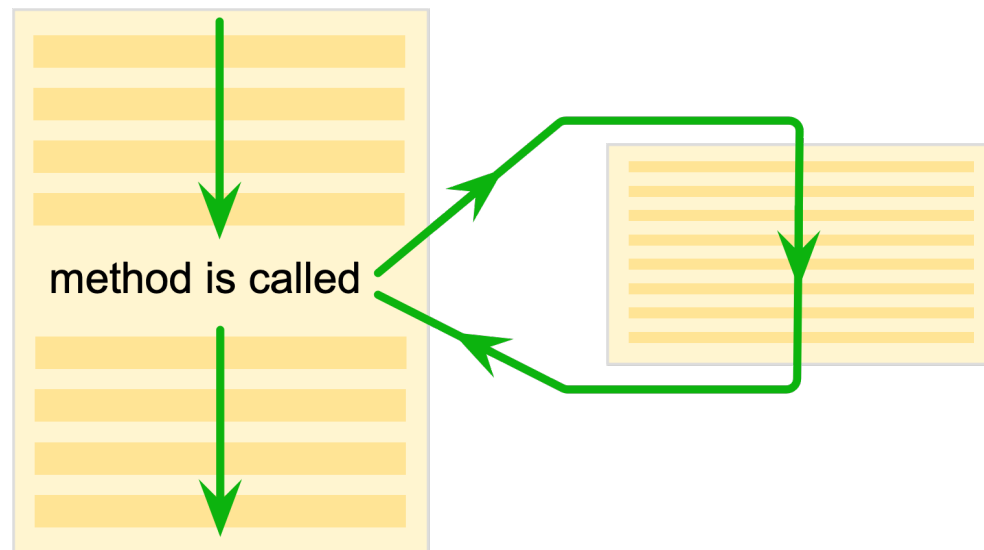
Flow of Control

- The order in which the statements of a program are executed is called the program's **flow of control**.
- The flow of control determines what happens next in a program.
- Unless told otherwise, a program executes statements in a **linear** fashion:



Flow of Control

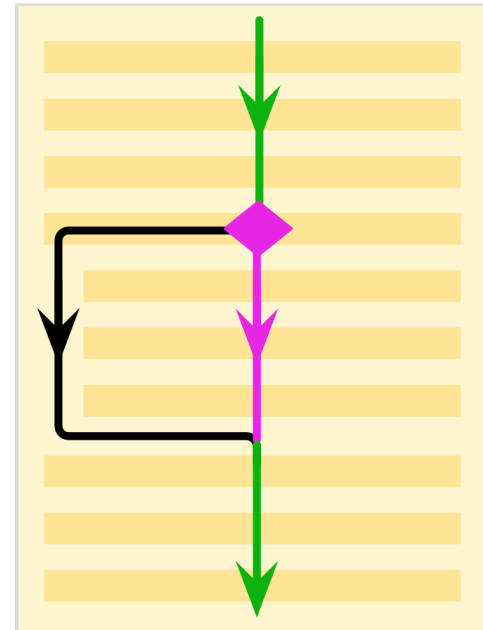
- When a **method** is called, the flow jumps to the method.
- When complete, the flow **returns** to the location where the method was called.



Flow of Control

- A **conditional** statement, such as an **if** statement, evaluates a condition to determine what to do next.

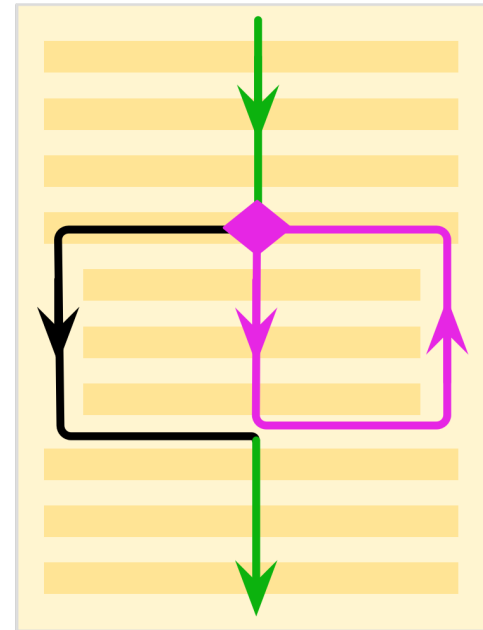
```
if (total > 25)
{
    maximum = total * 2;
    System.out.println(
        "The maximum is " +
        maximum);
}
```



Flow of Control

- A **loop**, such as a **while** statement, also evaluates a condition.
- It executes a block of code repeatedly.

```
while (count < 100)
{
    System.out.println(count);
    count = count + 1;
}
```



While Statement

A **while statement** is used to execute a set of program statements **multiple times**

It is called a loop or repetition statement

Like an if statement, it evaluates a boolean **condition** and only executes the body of the loop if the condition is true

But unlike an if, it then evaluates the condition again – if it's still true, the body is executed again

The body of the **while loop** is executed repeatedly until the condition becomes false

While Statement

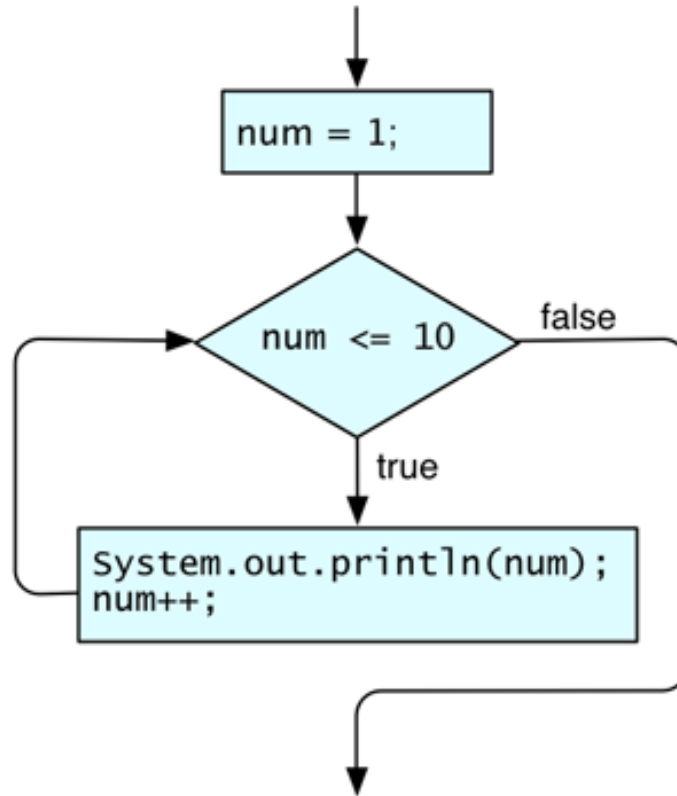
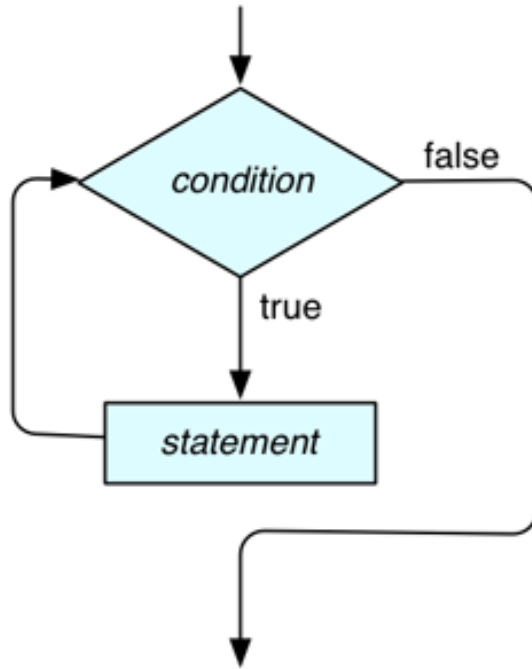
```
int num = 1;

while (num <= 5)
{
    System.out.println(num);
    num++;
}

System.out.println("Now here.");
```

```
1
2
3
4
5
Now here.
```

While Statement



Algorithm to Print Numbers from 7 to 100

Write an algorithm in **pseudocode** that prints the integers from 7 to 100.

Initialize **num** to 7

While **num** is less than or equal to 100

 print **num**

 increment **num**

The Code

```
int num = 7;

while (num <= 100)
{
    System.out.println(num);
    num++;
}
```

While Statement



An **infinite loop** is a loop that doesn't terminate normally – the condition doesn't ever become false

```
int num = 1;
while (num <= 10)
{
    System.out.println(num);
}
```



To terminate the program, press Control-C or Control-Break
Carefully check the logic involved to avoid them

While Statement

Here's an example that uses a while loop for **input validation**

```
double num = 0.0;

while (num <= 100)
{
    System.out.print("Enter a number > 100: ");
    num = in.nextDouble();
}
System.out.println("Moving on...");
```

If the user enters a valid number right away, the loop body only executes once

```
Enter a number > 100: 101
Moving on...
```

While Statement

The loop "traps" the user until a valid value is entered

```
Enter a number > 100: 70
Enter a number > 100: 35
Enter a number > 100: 99.99
Enter a number > 100: 500
Moving on...
```

Sometimes you can tell how many times a loop will execute by looking at the code, and sometimes you can't

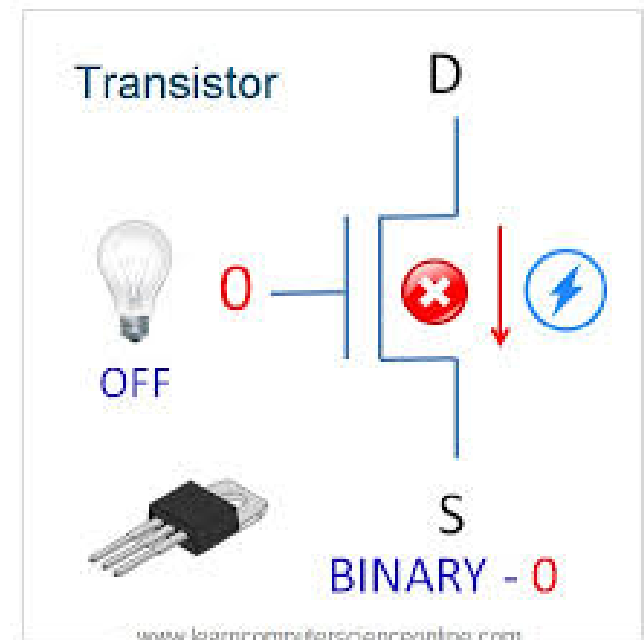
Example: Accumulating Interest

Walk-thru the Refactor topic [Example: Accumulating Interest](#)

DATA REPRESENTATION

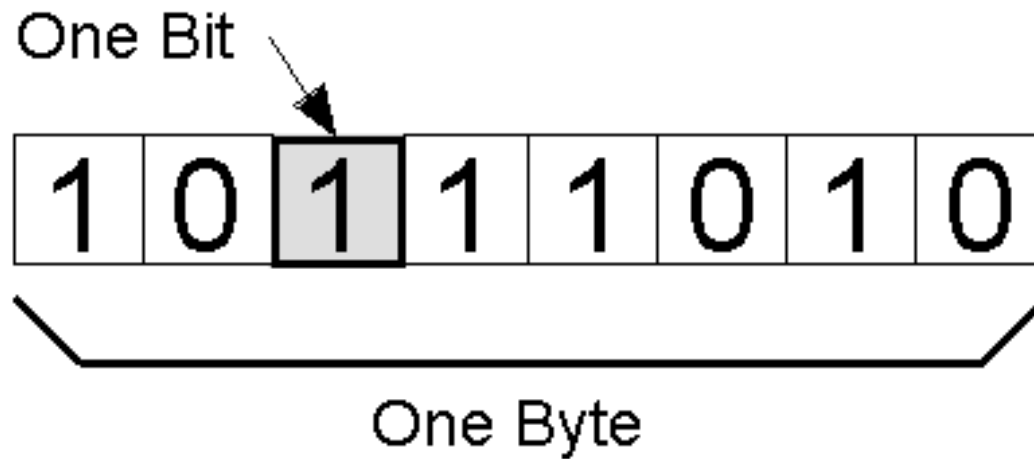
Why do Computers use Binary Numbers

- Humans have 10 fingers, so they use decimal numbers
- Computers only have 2 fingers to count with in the form of electronic switches that are either **on** or **off**.
- Computers use binary numbers
- One **byte** contains 8 bits



What's a Byte?

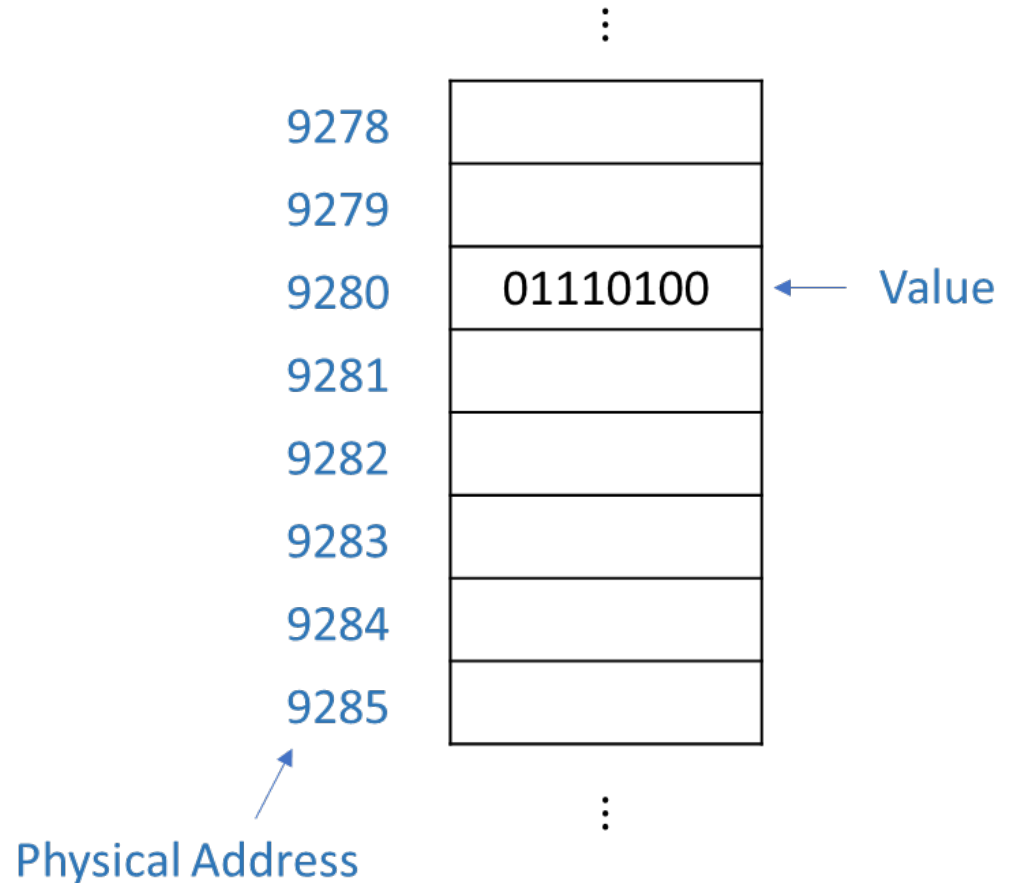
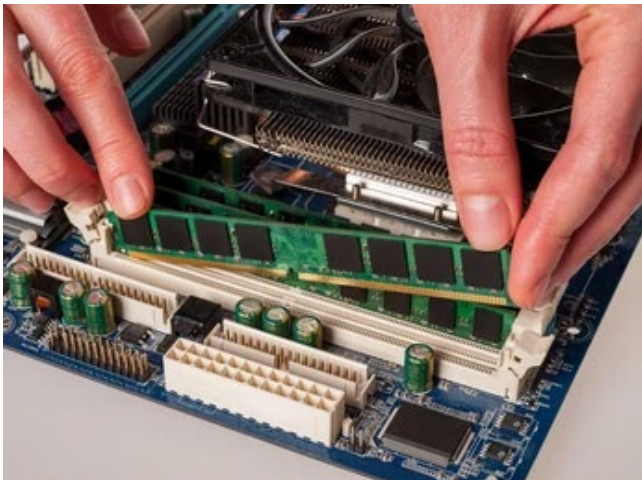
- We know that one **bit** is a binary digit
- One **byte** contains 8 bits



Storing Data in the Computer

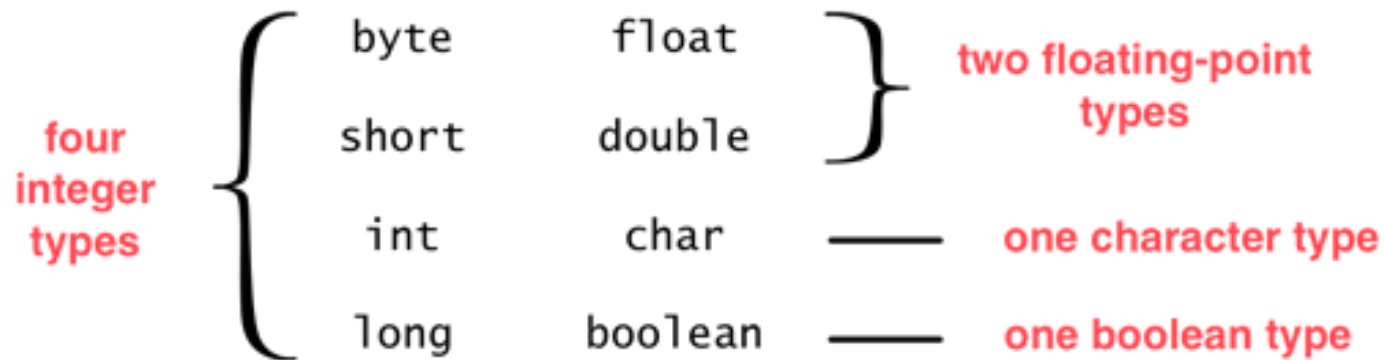
Computers store all information **digitally**, using **binary numbers**:

- numbers & text
- images, audio & video
- program instructions
- objects



Primitive Data Types

In Java, fundamental data is represented by one of the eight primitive data types



Everything else is represented as an object

Primitive Data Types

Six of the eight are numeric types: 4 integer and 2 floating-point
They differ by how much memory space they use to store a value,
which determines the possible range of values

Type	Size	Minimum	Maximum
byte	1 byte	-128	127
short	2 bytes	-32,768	32,767
int	4 bytes	-2,147,483,648	2,147,483,647
long	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
float	4 bytes	-3.4×10^{38} (7 sig. digits)	3.4×10^{38} (7 sig. digits)
double	8 bytes	-1.7×10^{308} (15 sig. digits)	1.7×10^{308} (15 sig. digits)

Primitive Data Types

You cannot use the comma grouping character in a program

```
int count = 150,765;
```



However, you can format your output to include a grouping character

You can also format output to print floating-point values to a certain precision

Primitive Data Types

The char data type represents a single character

Character literals are surrounded by single quotes

```
char terminator = ';' ;  
char middleInitial = 'R' ;  
char topGrade = 'A' ;
```

The character 'X' is different than the character string "X"

The character '7' is different than the number 7

The character 'a' is different than the character 'A'

Primitive Data Types

A **character set** is simply a list of characters

Java characters are represented by the **Unicode** character set

Unicode includes thousands of characters and symbols used in languages all over the world

The characters in a character set are in a particular order and each has its own numeric value

A char variable stores the numeric value of the character represented

Primitive Data Types

The boolean type is named after George Boole, an English mathematician and logician

It has only two possible values: true and false

A boolean variable can be used to represent a condition

```
boolean flag = true;  
boolean targetFound = false;  
boolean tooHigh, trialMode, gameOver;
```

Many Java operators return boolean results, which are often used as the conditions of if statements and loops

The Unicode Character Set

A **character set** is a list of characters used by a language

The way those characters are represented in memory is called a **character encoding**

The **ASCII** character set is still in use, but has limitations

ASCII – American Standard Code for Information Interchange

It was originally a 7-bit code, allowing only 128 characters to be represented

English letters, digits, punctuation

An extended 8-bit version allowed for accented characters and other symbols, but was still inadequate

The Unicode Character Set

The goal of the [Unicode character set](#) is to represent all the characters used in written languages across the world

Including Asian ideographs and symbols from special domains

It was originally specified as a straightforward 16-bit encoding, which allowed for 65,536 characters

But even that wasn't enough

Later, the encoding scheme was extended to include supplementary characters

The Unicode Character Set

Java supports the Unicode character set

A char variable stores the numeric code that represents a 16-bit Unicode character

By design, ASCII is a subset of Unicode – the first 128 characters of Unicode are the ASCII characters

Upper case and lower case letters are in order and contiguous

This allows characters and strings to be put in [lexicographic order](#), which is not quite alphabetical ordering

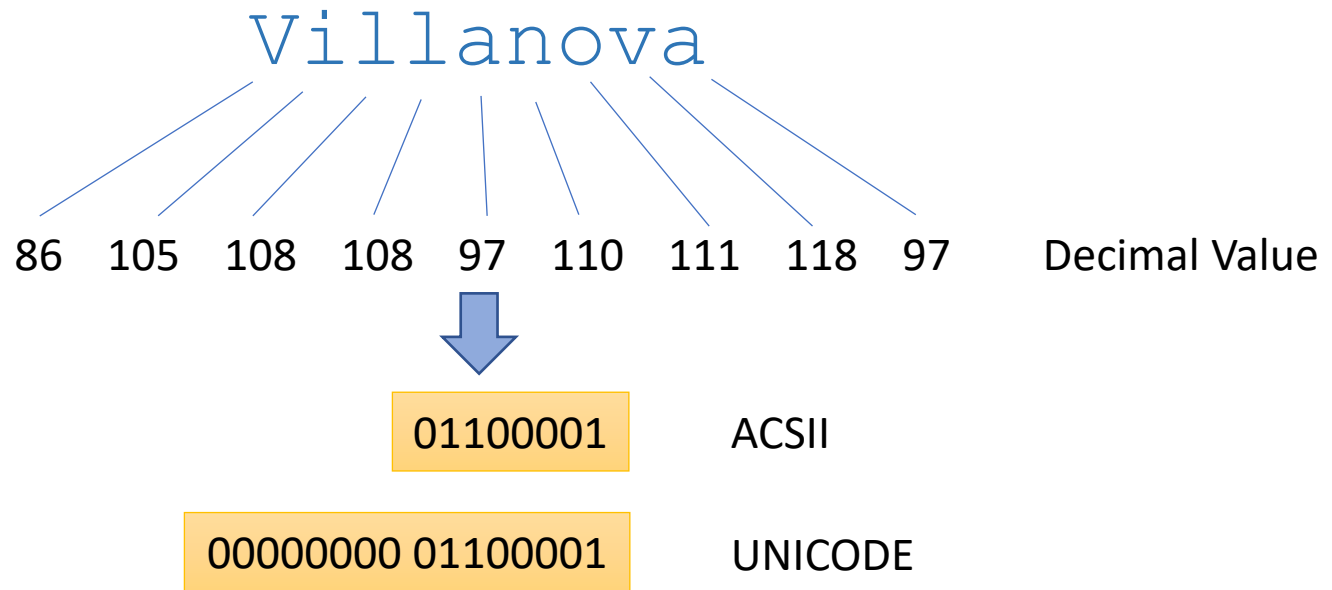
"Zoo" comes *before* "able" because upper case 'Z' comes before lower case 'a'

ASCII part of The Unicode Character Set

Value	Char	Value	Char	Value	Char	Value	Char	Value	Char
32	space	51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	"	53	5	72	H	91	[110	n
35	#	54	6	73	I	92	\	111	o
36	\$	55	7	74	J	93]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	'	58	:	77	M	96	`	115	s
40	(59	;	78	N	97	a	116	t
41)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

Characters in Java

Characters, including spaces, digits, and punctuation are represented by numeric values.



ASCII uses eight bits per character, allowing for 256 unique characters

Unicode extends ASCII to sixteen bits per character, allowing for 65,536 unique characters.

Characters in Java

A **char** variable stores a single character

Character literals are delimited by single quotes:

```
'a' 'X' '7' '$' ',' '\n'
```

```
char grade = 'A';  
char terminator = ';', separator = ' ', newline = '\n';  
String oneLetter = "A"; // this is a string not char
```

A **String literal** can hold multiple characters.

Break and Continue Statements



The **break statement** is also used to break out of a **switch statement**.

The **break statement** and the **continue statement** affect the flow of execution in a loop.

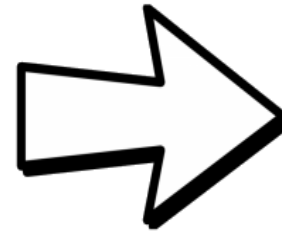
The **break statement** is also used to break out of a **switch statement**.



Break Statement

```
int num = 0;
while (num < 10)
{
    num++;
    System.out.println(num);

    if (num == 5)
        break;
}
System.out.println("Now here.");
```



```
1
2
3
4
5
Now here.
```

As soon as the value of `num` equals 5, the `break` statement is executed... and the loop is immediately **exited**.

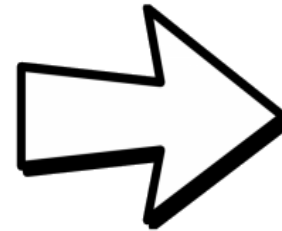
Next, the `println` statement after the loop is executed.

Continue Statement

```
int num = 0;
while (num < 6)
{
    num++;

    if (num == 3)
        continue;

    System.out.println(num);
}
System.out.println("Now here.");
```



```
1
2
4
5
6
Now here.
```

When the value of `num` equals 3, the `continue statement` is executed... the rest of the current iteration of the loop is `skipped`.... so 3 is not printed.

Finally, the `println statement` after the loop is executed.