AP CS A (Java) Training

Unit 2: Selection and Iteration

Updated Curriculum (2025-2026)

2.1 Algorithms with Selection and Repetition

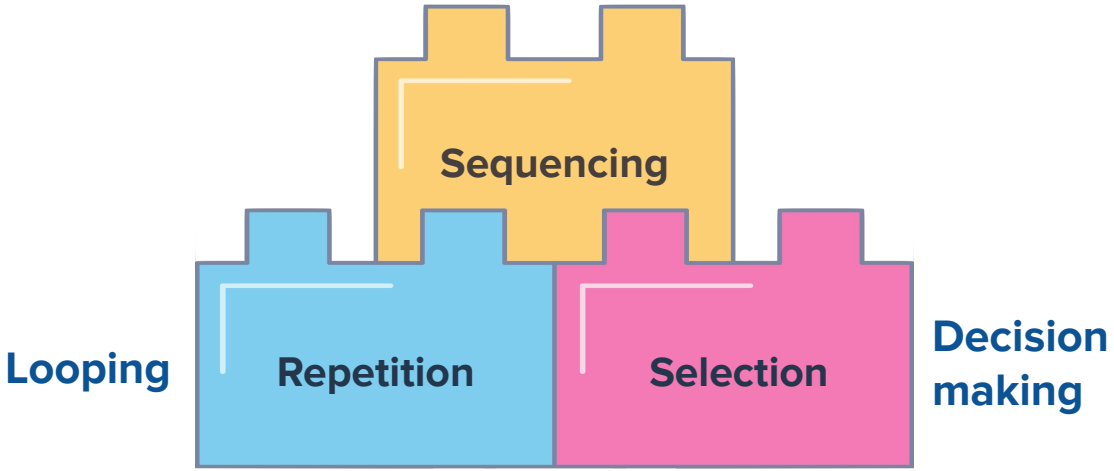
Learning Objectives:

- Represent patterns and algorithms that involve selection and repetition found in everyday life using written language or diagrams.

Build Blocks of Algorithms

Algorithm is a step-by-step process to follow when completing a task or solving a problem. Algorithms can tackle real world tasks, like making a peanut butter & jelly sandwich, or they can make up parts of computer programs.

There are 3 main building blocks for algorithms: sequencing, repetition, and selection. In this lesson, we'll focus on how algorithms use selection, which involves decision making and repetition, which involves looping.



Sequencing

Sequencing is the order in which instructions are arranged and processed in order to achieve a desired outcome.

For example, part of the process of getting dressed would involve putting on your pants, putting on your socks, and putting on your shoes. If the order was different …for example putting on your shoes before your socks, you will have a hard time getting dressed.

Repetition

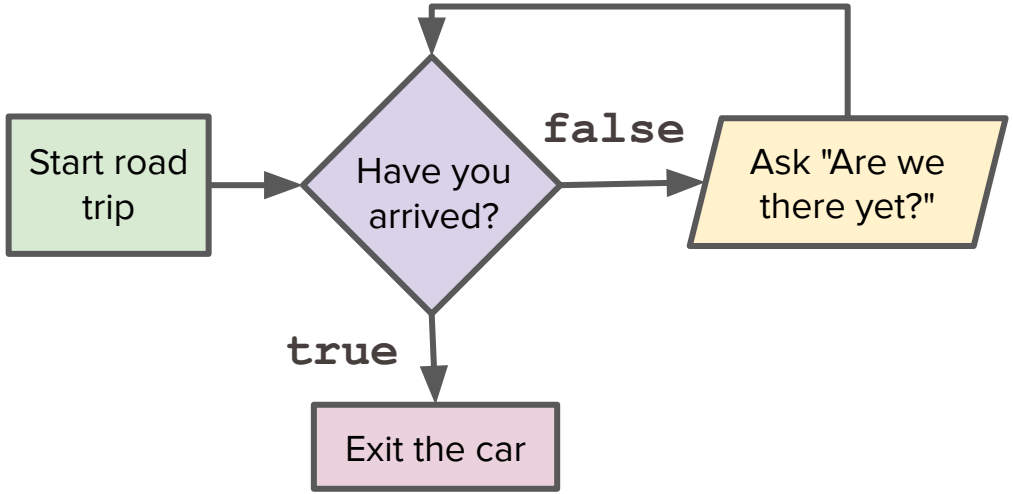
Repetition is when a process repeats itself until a desired outcome is reached.

For example: you are heading out on a road trip with no idea as to when you will get there, so what do you do? You ask your parents if you are there yet. You keep asking until you arrive, at which point you are there and can exit the car.

Start road trip

Repeat until you arrive:
- Ask parents "Are we there yet?"

You have arrived. Exit the car.



Selection

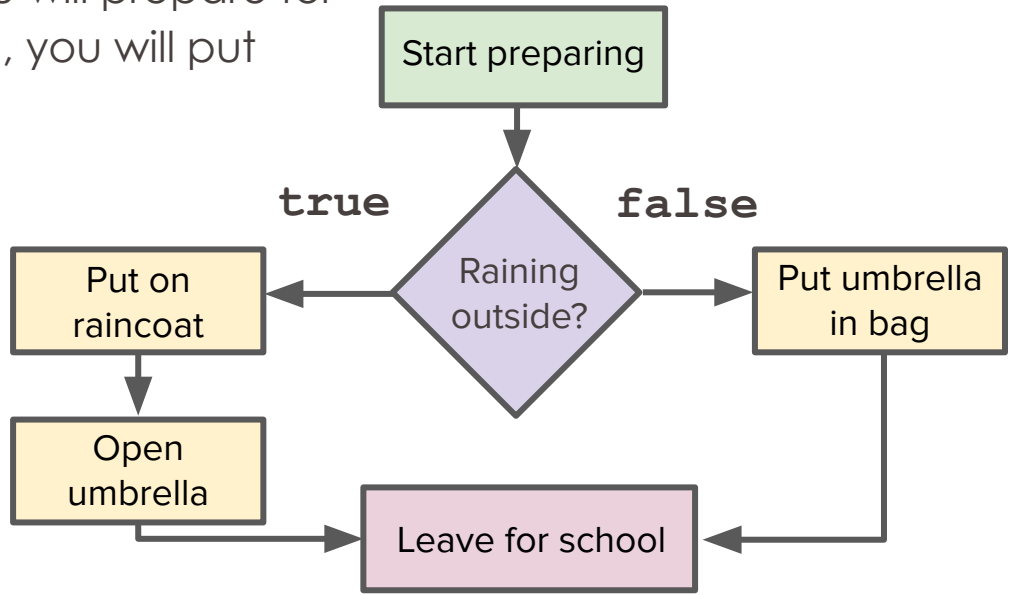
Selection occurs when a decision on how to proceed needs to be made.

For example: it's time to leave for school, but before you do, you check to see if it is raining. If it is raining outside, you will prepare for the rain. Otherwise, if it's not raining, you will put your umbrella in your bag.

If it is raining outside:
- Put on raincoat
- Open umbrella

Otherwise:
- Put umbrella in bag

Leave for school



The order in which sequencing, selection, and repetition are used contributes to the outcome of the algorithm. Let's look at an example algorithm that uses all three:

1. Prepare your workspace.
2. Review assignments.
3. Prioritize tasks:
    - If a task is due tomorrow, move it to the top of the list.
    - Otherwise, prioritize by difficulty or estimated time required.
4. Repeat until all assignments are completed:
    - Read the instructions carefully.
    - Work on the task until it's complete.
    - Submit:
        - If the assignment is digital, upload to the required platform.
        - Otherwise, place the completed work in your backpack.
5. Wrap up:
    - Preview upcoming assignments.
6. Relax!

Sequence: The order of the steps.

1. Prepare your workspace.
2. Review assignments.
3. Prioritize tasks:
    - If a task is due tomorrow, move it to the top of the list.
    - Otherwise, prioritize by difficulty or estimated time required.
4. Repeat until all assignments are completed:
    - Read the instructions carefully.
    - Work on the task until it's complete.
    - Submit:
        - If the assignment is digital, upload to the required platform.
        - Otherwise, place the completed work in your backpack.
5. Wrap up:
    - Preview upcoming assignments.
6. Relax!

Selection: Decide if a task is prioritized and how to submit each assignment.

1. Prepare your workspace.
2. Review assignments.
3. Prioritize tasks:
   - If a task is due tomorrow, move it to the top of the list.
   - Otherwise, prioritize by difficulty or estimated time required.
4. Repeat until all assignments are completed:
   - Read the instructions carefully.
   - Work on the task until it's complete.
   - Submit:
     - If the assignment is digital, upload to the required platform.
     - Otherwise, place the completed work in your backpack.
5. Wrap up:
   - Preview upcoming assignments.
6. Relax!

Repetition: Repeat a set of actions until all of the assignments are complete.

1. Prepare your workspace.
2. Review assignments.
3. Prioritize tasks:
    - If a task is due tomorrow, move it to the top of the list.
    - Otherwise, prioritize by difficulty or estimated time required.
4. Repeat until all assignments are completed:
    - Read the instructions carefully.
    - Work on the task until it's complete.
    - Submit:
        - If the assignment is digital, upload to the required platform.
        - Otherwise, place the completed work in your backpack.
5. Wrap up:
    - Preview upcoming assignments.
6. Relax!

Summary

- The building blocks of algorithms include sequencing, selection, and repetition.
- **Algorithms** can contain selection, through decision making, and repetition, via looping.
- **Selection** occurs when a choice of how the execution of an algorithm will proceed is based on a true or false decision.
- **Repetition** is when a process repeats itself until a desired outcome is reached.
- The order in which sequencing, selection, and repetition are used contributes to the outcome of the algorithm.

2.2 Boolean Expressions

Learning Objectives:

- Develop code to create Boolean expressions with relational operators and determine the result of these expressions.

Type boolean

- **`boolean`**: A logical type whose values are **`true`** and **`false`**.
  - It is legal to:
    - create a **`boolean`** variable
    - pass a **`boolean`** value as a parameter
    - return a **`boolean`** value from methods
    - call a method that returns a **`boolean`** and use it as a test
  - Example:

```
boolean loggedIn = false;
boolean gameOver = true;
```

- A **boolean expression** is a statement or expression that can be evaluated as **`true`** or **`false`**.
- Why is type **`boolean`** useful?
  - Can capture a complex logical test result and use it later
  - Can write a method that does a complex test and returns it
  - Makes code more readable
  - Can pass around the result of a logical test (as parameter/return)

Relational Operators

Relational operators are used to compare the value of two expressions. This is a full list of all the arithmetic relational operators that can be used to evaluate boolean expressions in Java.

Note that the `==` tests equality, not the `=` . The `=` is used for the assignment operator!

Operator	Meaning	Example	Value
`==`	is equal to	`1 + 1 == 2`	`true`
`!=`	does not equal	`3.2 != 2.5`	`true`
`<`	is less than	`10 < 5`	`false`
`>`	is greater than	`10 > 5`	`true`
`<=`	is less than or equal to	`126 <= 100`	`false`
`>=`	is greater than or equal to	`5.0 >= 5.0`	`true`

VERY IMPORTANT: Although some programming languages allow using relational operators like `<` to compare strings, Java only uses these operators for primitive data types. So these operators do **NOT** work properly on `String` or any other object(reference) data type!!

Important Distinction!

- **Assignment Operator**
  is used to assign a variable a value, use `=`
  - `int x = 34;`
  - the integer variable **x** is <u>assigned</u> the value **34**

- **Equality Operator**
  is used to check the whether two numbers are equal, use `==`
  - `int x = 34;`
    `x == 45;`
  - Checks whether the value stored in **x** (**34**) is equal to **45**

Useful Boolean Expressions

```
// Test if a number is positive
boolean isPositive = number > 0

//Test if a number is negative
boolean isNegative = number < 0

//Test if a number is even by seeing if the remainder is 0 when divided by 2
boolean isEven = number % 2 == 0

//Test if a number is odd by seeing if there is a remainder when divided by 2
boolean isOdd1 = number % 2 == 1
boolean isOdd2 = number % 2 > 0

//Test if a number is a multiple of x (or divisible by x with no remainder)
boolean isMultipleOfX = number % x == 0
```

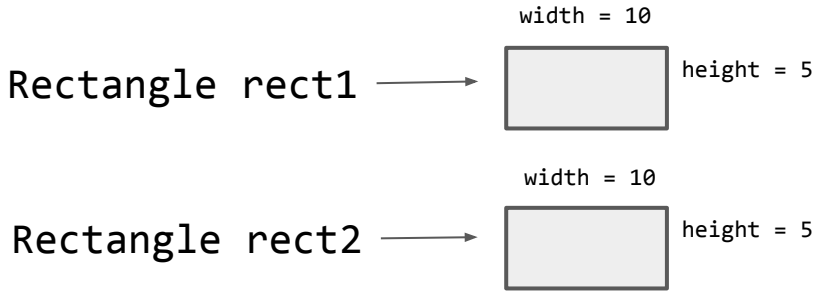
Reference Comparisons Different Object

Warning! Reference Comparisons can be Misleading!

```
Rectangle rect1 = new Rectangle(10,5);
Rectangle rect2 = new Rectangle(10,5);
boolean test1 = rect1 == rect2;
System.out.println(test1);
```

false

Even though both **rect1** and **rect2** have the same values, they are **NOT** the same object. **test1** evaluates to **false** because they reference different **Rectangle** objects



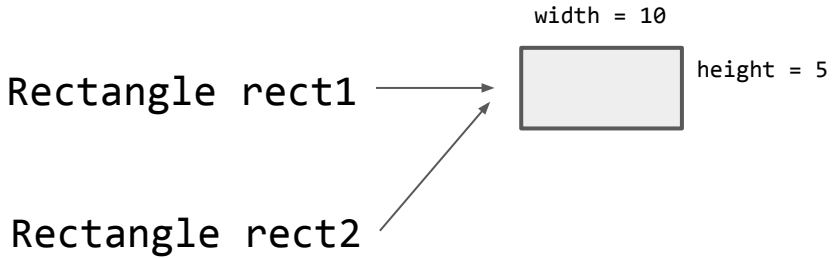
Reference Comparisons Same Object

Warning! Reference Comparisons can be Misleading!

```
Rectangle rect1 = new Rectangle(10,5);
Rectangle rect2 = rect1;
boolean test1 = rect1 == rect2;
System.out.println(test1);
```

→ `true`

Now that **rect1** has been assigned to **rect2**, **test1** will evaluate **true**. Both **rect1** and **rect2** point to the same **Rectangle** object.



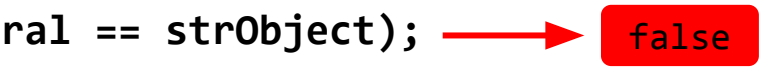
Comparing Strings

This is why we use **`.equals()`** for Strings instead of **`==`**

```
String literal = "Test!";

String strObject = new String("Test!");

System.out.println(literal == strObject);
```




```
System.out.println(literal.equals(strObject));
```

Summary

- Values or expressions can be compared using the relational operators `==` and `!=` to determine whether the values are the same. With primitive types, this compares the actual primitive values. With reference types, this compares the object references.
- Numeric values or expressions can be compared using the relational operators (`<`, `>`, `<=`, `>=`) to determine the relationship between the values.
- An expression involving relational operators evaluates to a `boolean` value of `true` or `false`.
- The remainder operator `%` can be used to test for divisibility by a number. For example, `num % 2 == 0` can be used to test if a number is even.

2.3 if Statements

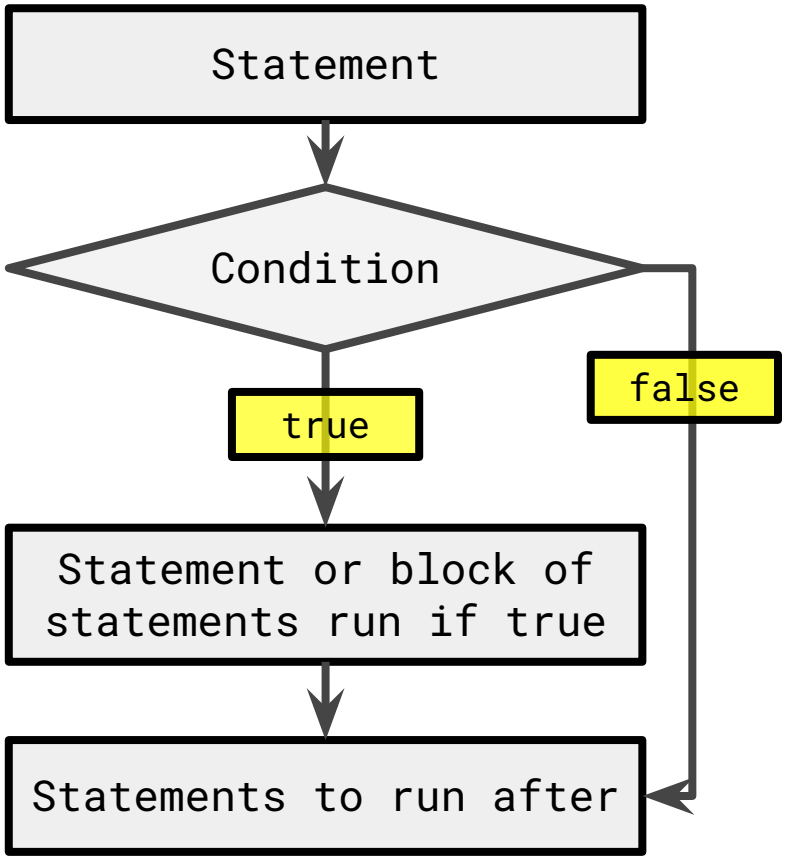
Learning Objectives:

- Develop code to represent branching logical processes by using selection statements and determine the result of these processes.

If statement Flowchart

- **Selection** (one of the 3 basic logic structures in algorithms) allows us to choose different outcomes based on the result of a decision/condition
- We can use a flowchart to represent algorithms that use selection. See the example to the right

Note that the diamond in the flowchart is referred to as a **decision diamond**.



How If-statements work

if-statements are the lines of code you need to change the flow while you're program is running. You can write code that makes a decision that determines if certain lines of code should be run.

There are two basic parts to an if-statement.

1. A **condition** to be evaluated (A Boolean expression that evaluates to true or false)
2. **Code** that should run if the expression was true {enclosed in curly braces}

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Very Important Note: The if-statement does NOT NEED to use { } when there is only one line of code for the conditional statement.

Writing if Statements

- for `if` statements, always make sure the `condition` is encapsulated in in open and closed parenthesis `()`
- `{ }` is not needed when there is only one line of code for the conditional statement

```
// An if statement
if (condition)
     statement;

//An if statement with curly braces {}
if (condition) {
     statement;
}

// An if statement with multiple statements -- must use {}
if (condition) {
     statement1;
     statement2;
          .
          .
          .
}
```

If-statement Execution

```
int age = 19;
if(age >= 18)
{
    System.out.println("You can vote!");
}
```

If the boolean expression is **true**, the code in between the curly brackets will execute!

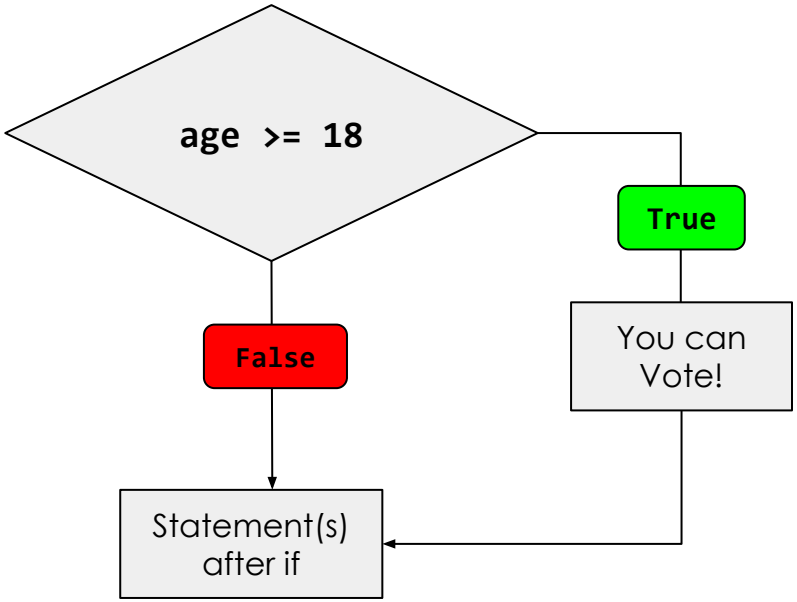
```
You can vote!
```

```
int age = 11;
if(age >= 18)
{
    System.out.println("You can vote!");
}
```

If the boolean expression is **false**, the code in between the curly brackets will NOT execute, and the program will execute everything after the if statement!

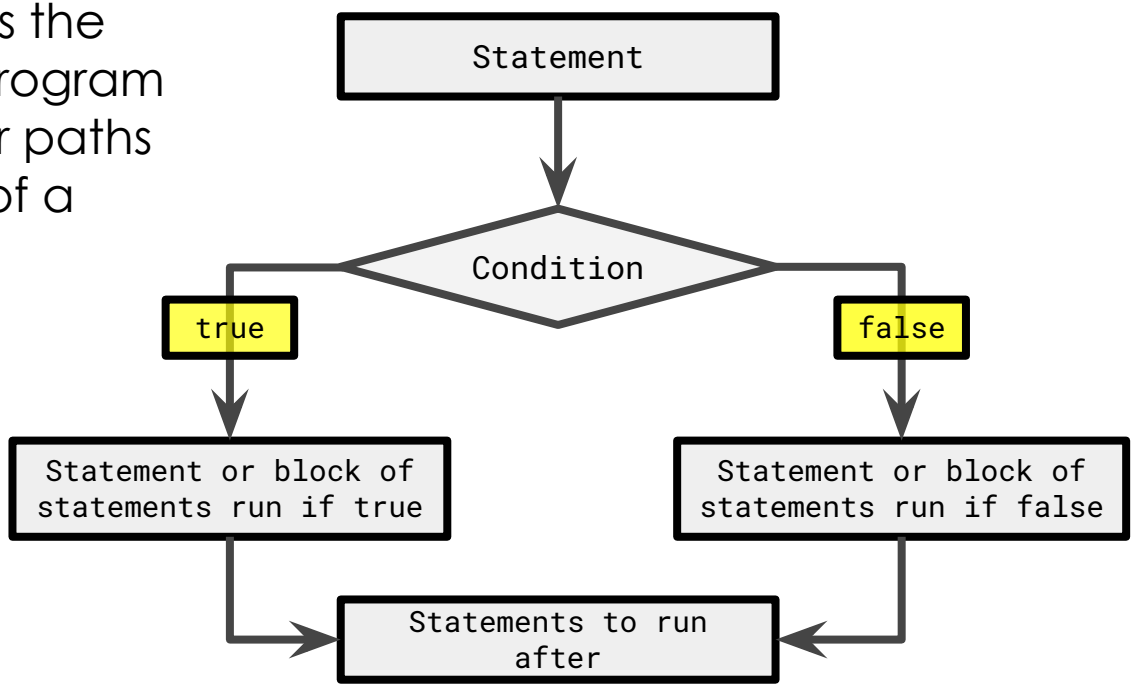
if Statement Execution

Taking the previous example, we can see how this looks in this flowchart form.



If/else statement Flowchart

An if-else statement allows the programmer to make a program have multiple branches or paths depending on the value of a boolean.



How If-Else Statements work

With an if-else statement you are giving an either-or command: either the lines of code inside the if will execute **or** the lines inside the else will execute. Those are the options. Inside the curly braces for the else clause you put lines of code that you want to run if the Boolean condition from the if statement is false.

Some important notes about the **`else`** clause:

- The **`else`** must come immediately after the closing curly brace of an if statement
- The **`else`** also has its own set of opening and closing curly braces to encapsulate lines of code that will run if the if condition is **`false`**

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Writing if-else Statements

{ } is still not needed when there is only one line of code for the conditional statement!!

```
// An if-else statement
if (condition)
     statement;
else
     statement;

//An if-else statement with curly braces {}
if (condition) {
     statement;
} else {
     statement;
}

// An if-else statement with multiple statements
if (condition) {
     statement1;
} else {
     statement1;
     statement2;
          .
          .
}
```

If-statement Execution

```
int age = 19;
if(age >= 18)
{
     System.out.println("You can vote!");
}
else
{
     System.out.println("Sorry, you can't vote!");
}
```

If the boolean expression is **true**, the code in between the **if** curly brackets will execute!

```
You can vote!
```

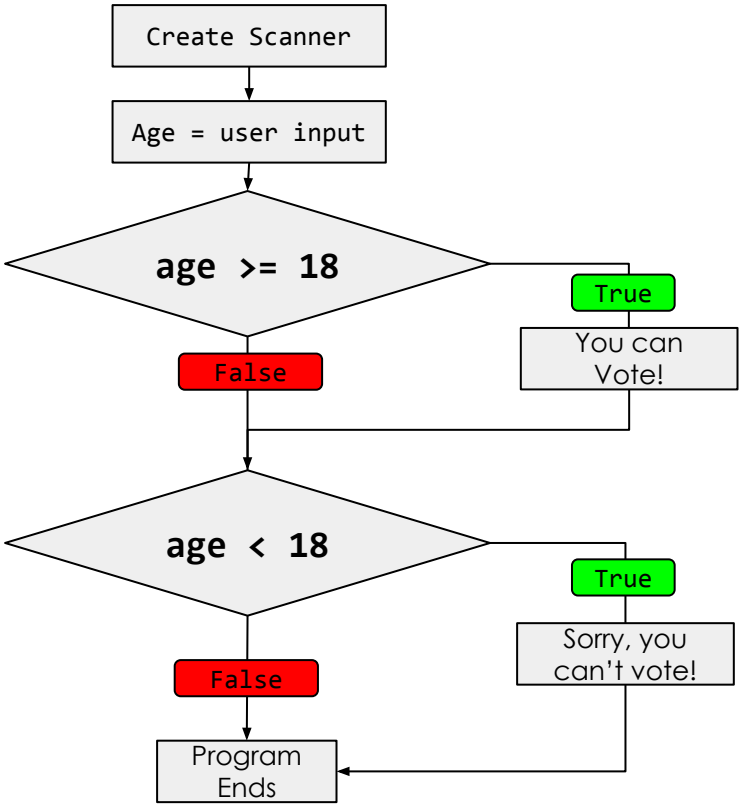
```
int age = 11;
if(age >= 18)
{
     System.out.println("You can vote!");
}
else
{
     System.out.println("Sorry, you can't vote!");
}
```

If the boolean expression is **false**, the code in the **else** curly brackets will execute!

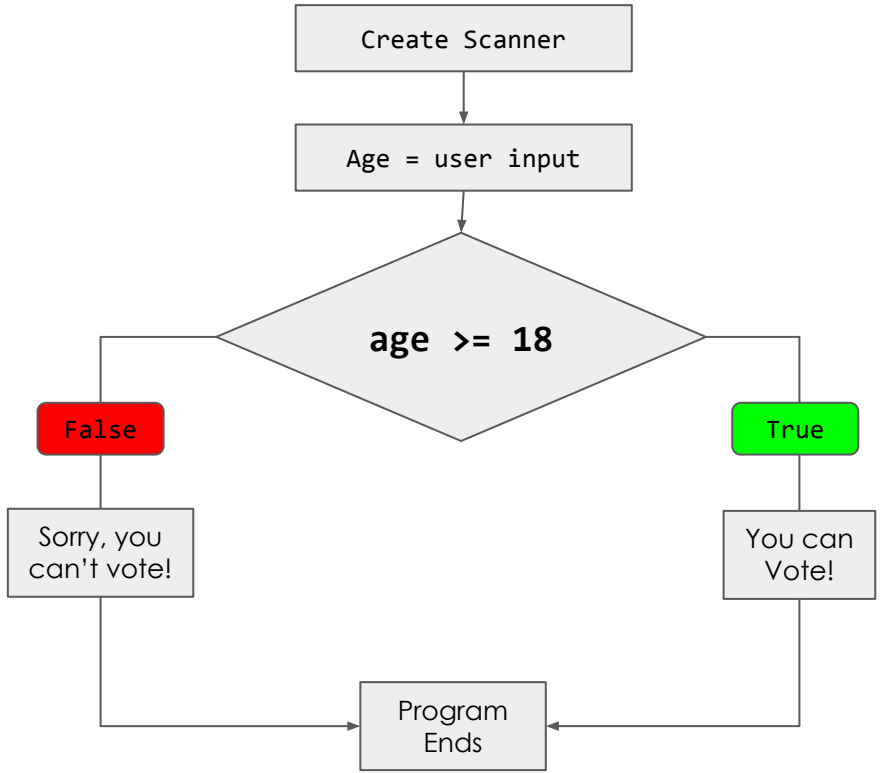
```
Sorry, you can't vote!
```

IF VS IF ELSE EXECUTION

By adding the else statement, we are removing additional execution steps that would otherwise be needed if we had two if statements back to back.



VS



Summary

- **Selection** statements change the sequential execution of statements.
- An if statement is a type of selection statement that affects the flow of control by executing different segments of code based on the value of a Boolean expression.
- A one-way selection (if statement) is used when there is a segment of code to execute under a certain condition. In this case, the body is executed only when the Boolean expression is true.
- **if statements** test a boolean expression and if it is true, go on to execute the following statement or block of statements surrounded by curly braces ({}) like below.

```
// A single if statement
if (boolean expression)
    Do statement;
// A block if statement
if (boolean expression)
{
   Do Statement1;
   Do Statement2;
   ...
   Do StatementN;
}
```

Summary continued...

- **Relational operators** (`==`, `!=`, `<`, `>`, `<=`, `>=`) are used in boolean expressions to compare values and arithmetic expressions.
- **If statements** can be followed by an associated `else` part to form a 2-way branch:

```
if (boolean expression)
{
    Do statement;
}
else
{
    Do other statement;
}
```

- A two-way selection (`if/else`) is used when there are two segments of code—one to be executed when the Boolean expression is `true` and another segment for when the Boolean expression is `false`. In this case, the body of the `if` is executed when the Boolean expression is `true`, and the body of the else is executed when the Boolean expression is `false`.

2.4 Nested if Statements

Learning Objectives:

- Develop code to represent nested branching logical processes and determine the result of these processes.

Misuse of if

- What's wrong with the following code?

```
int percent = 90;

if (percent >= 90) {
    System.out.println("You got an A!");
}
if (percent >= 80) {
    System.out.println("You got a B!");
}
if (percent >= 70) {
    System.out.println("You got a C!");
}
if (percent >= 60) {
    System.out.println("You got a D!");
}
if (percent < 60) {
    System.out.println("You got an F!");
}
...
```

These are all separate if statements, so they will each be checked separately one by one. So this will actually output:

```
You got an A!
You got a B!
You got a C!
You got a D!
```

if statements and if-else statements allows us to compare one condition, and choose a path between **`true`** and **`false`** which is great! But what if there are more than two possible scenarios for a particular condition?

`else if` statements allow us to incorporate additional conditions!!

How else-if Works

Not all conditions you want to check have only two possible outcomes. However a computer can only check one true/false condition at a time. A **multi-selection statement** is a statement that selects a single action from three or more conditional statements based on which Boolean expression is **true**.

- You add an **else-if** clause to an if statement when you have another condition you want to check.
- You can add as many **else-if**s as you want.
- Each condition in an **if-else-if** is checked in order from top to bottom and the final else clause is executed if all the previous conditions are **false**.

Syntax

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

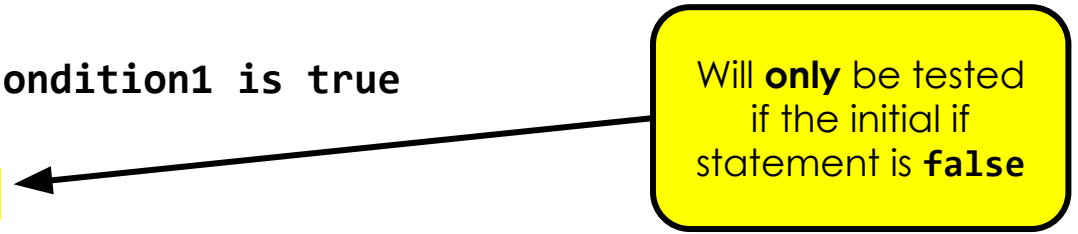
Writing else-if Statements

- MUST start a conditional with an **`if`**. **`else if`** and **`else`** are optional additions that can be used
- A conditional can only have one **`if`** and **`else`** statement, but can have an unlimited number of **`else if`** statements.
- Adding an **`if`** statement to an existing conditional statement would just create an additional conditional statement.

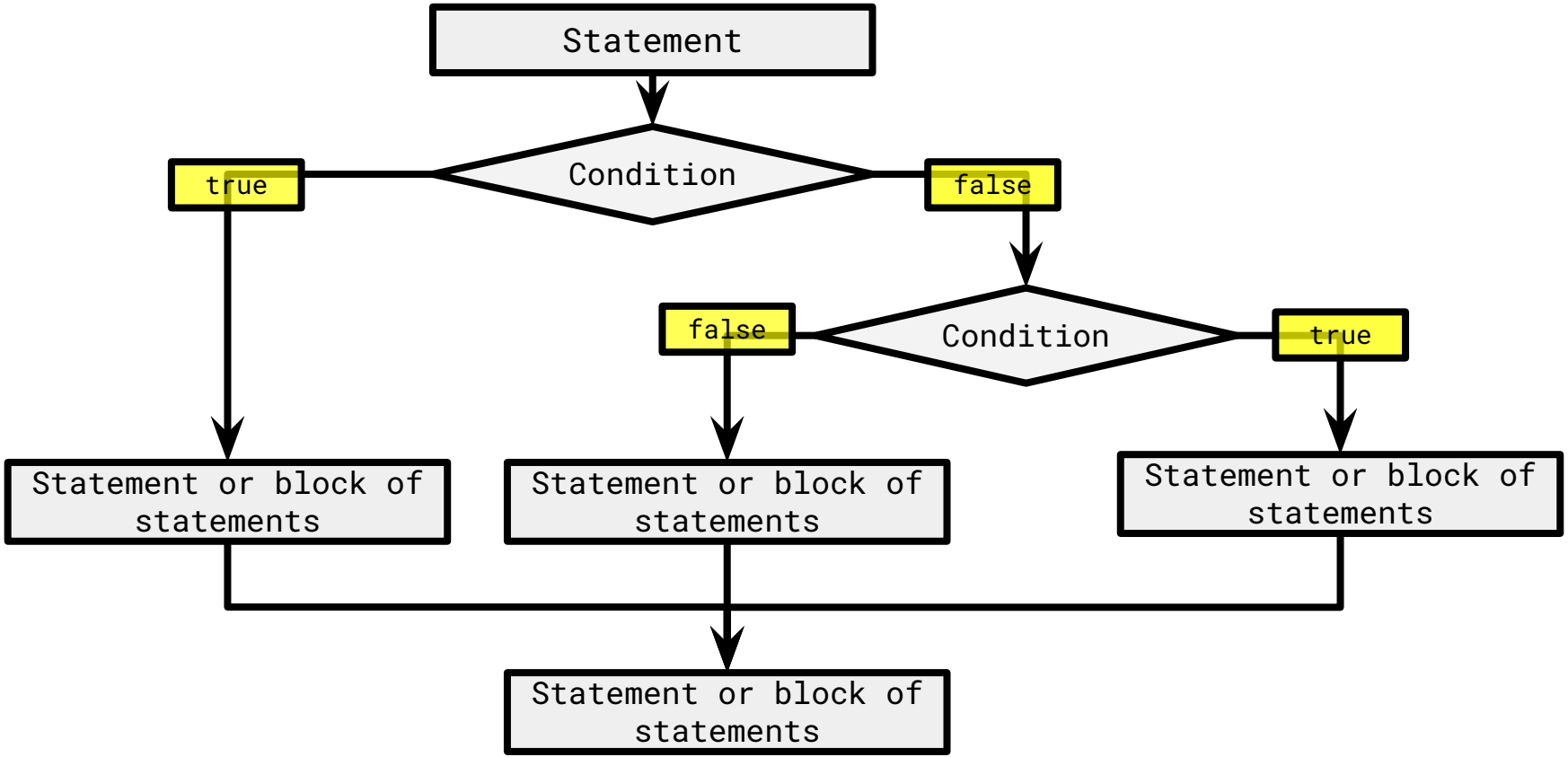
```
//A single if-else statement with curly braces {}
//NOTE: if only one statement, { } are not needed
if (condition)
{
     //statement or block of statements
}
else if (condition)
{
     //statement or block of statements
}
else
{
     //statement or block of statements;
}
```

Important Note about the conditions!!

```
if(condition1)
{
    //executed if condition1 is true
}
else if(condition2)
{
    // executed if the condition1 is false and condition2 is true
}
else
{
    //executed if all previous conditions are false
}
```



Else-if Statement Flowchart



If/else if/else Example

```
int x = 10;
if (x > 0) {
   System.out.println("Positive");
}
else if (x < 0) {
   System.out.println("Negative");
}
else {
   System.out.println("Zero");
}
```

Output:
Positive

If/else if/else Example

```
int x = 0;
if (x > 0) {
   System.out.println("Positive");
}
else if (x < 0) {
   System.out.println("Negative");
}
else {
   System.out.println("Zero");
}
```

Output:
Zero

Path executed from else if

- When an **else if** ends with **else**, exactly one path must be taken.

```
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else {
    statement(s);
}
```

- When an **else if** ends with **else if**, the code might not execute any path.

```
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else if (test) {
    statement(s);
}
```

if/else/if Example

```
int place = 2;
if (place == 1) {
   System.out.println("Gold medal!");
}
else if (place == 2) {
   System.out.println("Silver medal!");
}
else if (place == 3) {
   System.out.println("Bronze medal.");
}
```

```
Output:
Silver medal!
```

if/else/if Example

```
int place = 6;
if (place == 1) {
   System.out.println("Gold medal!");
}
else if (place == 2) {
   System.out.println("Silver medal!");
}
else if (place == 3) {
   System.out.println("Bronze medal.");
}
```

Output:
No output.

Conditional(selection) statements all use boolean expressions to decide whether to run certain pieces of code. There are 4 ways conditionals can be used:

1. **`if`** by itself
   - **if-statements** check if one boolean expression is true. If it is, the code in the if runs. Otherwise the selection block is skipped.

2. **`if`** followed by **`else`**
   - **if-else statements** add the functionality that if the condition is false it can still run some code.

3. **`if`** followed by **`else-if(s)`** and finally an **`else`**
   - **if-else if-else statements** can check more than one boolean expression. They will only run the code for the first boolean expression that evaluates to true. If none of the conditions are true, the final else will run.

4. **`if`** followed by **`else-if(s)`** and no **`else`**
   - **if-else if statements** also heck more than one boolean expression, but they do not use the final else, so if none of the conditions are true, nothing from the selection block will run.

```
//if-statement
if(faveNum == 2){
  System.out.println("two is blue");
}
```

```
//if-else statement
if(faceNum == 2){
  System.out.println("two is blue");
} else {
  System.out.println("you don't love two :(");
}
```

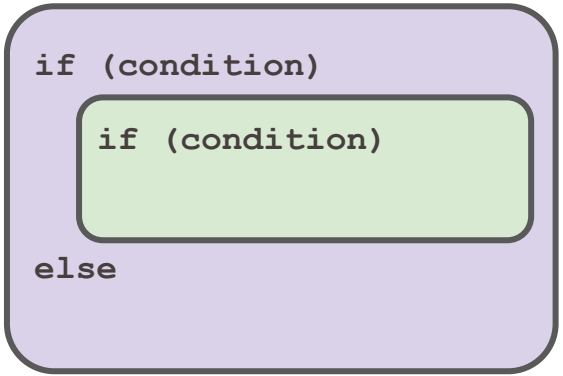
```
//if - else if(s) - else statement
if(faceNum == 2){
  System.out.println("two is blue");
} else if(faveNum == 3){
  System.out.println("three is free");
} else if(faveNum == 4){
  System.out.println("four is adore");
} else {
  System.out.println("you don't love two :(");
}
```

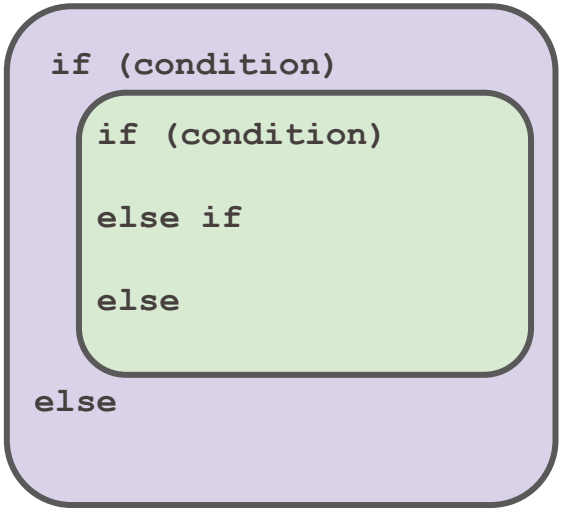
```
//if - else if(s) statement
if(faveNum == 2){
  System.out.println("two is blue");
} else if(faveNum == 3){
  System.out.println("three is free");
} else if(faveNum == 4){
  System.out.println("four is adore");
}
```

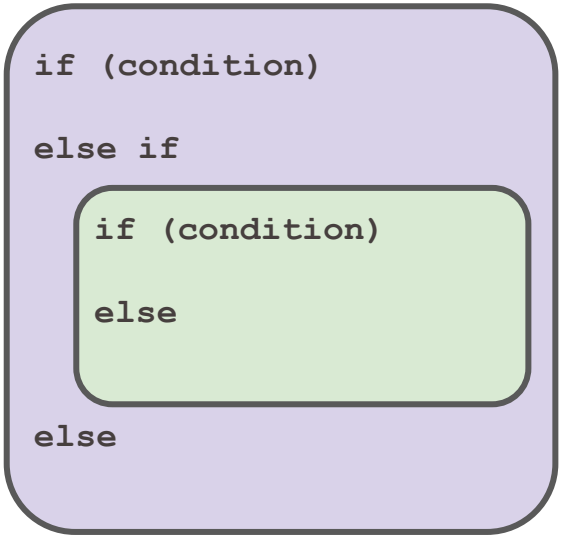
Nested `if` statements

You can use **nested if statements** if there are multiple conditions, and one of them depends on the other being true.

A nested **`if`** statement is an **`if`** statement that is placed within another **`if`** statement.

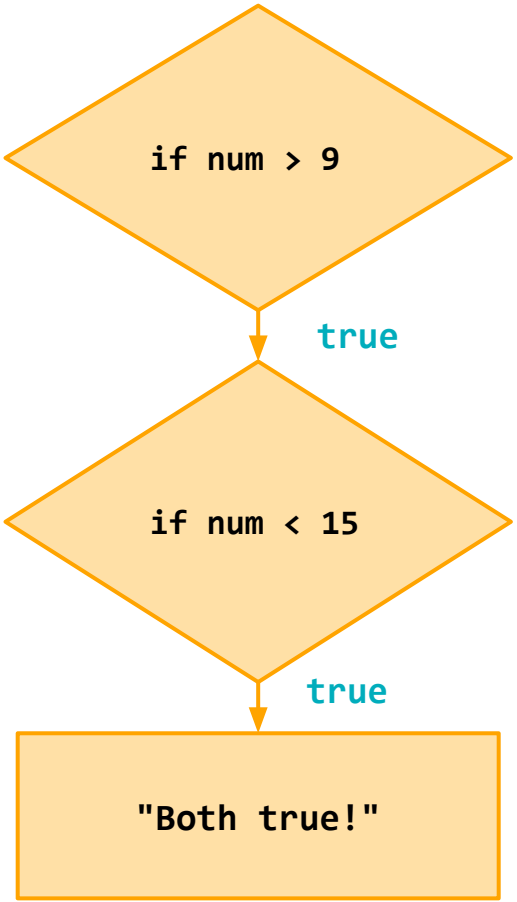






We can use nested **if** statements to allow the program to check conditions within other conditions. In these cases were really checking if multiple conditions are **true**, such as checking if someone has enough money for a movie ticket and an available movie time.

```
int num = 10;
if (num > 9) {
  if (num < 15) {
    System.out.println("Both true!");
  }
}
```



Example: A program that uses a nested if statement to check if a user's password meets the requirements: has at least 8 characters AND starts with the pound sign

```
System.out.println("Enter password: ");
String password = input.nextLine();
if (password.length() >= 8) {
    if (password.startsWith("#")) {
        System.out.println("Password accepted.");
    } else {
        System.out.println("Password must start with #.");
    }
} else {
    System.out.println("Password needs 8 or more characters.");
}
```

password:
CodeHSisKewl

```
Enter password: CodeHSisKewl
Passwords must start with #
```

Example: A program that uses a nested if statement to check if a user's password meets the requirements: has at least 8 characters AND starts with the pound sign

```
System.out.println("Enter password: ");
String password = input.nextLine();
if (password.length() >= 8) {
    if (password.startsWith("#")) {
        System.out.println("Password accepted.");
    } else {
        System.out.println("Password must start with #.");
    }
} else {
    System.out.println("Password needs 8 or more characters.");
}
```

password: #CodeHS

```
Enter password: #CodeHS
Passwords needs 8 or more characters.
```

The `if` of the inner nested `if` statement is only evaluated if the `if` of the outer `if` statement evaluates to `true`.

Summary

- **Nested if statements** consist of if, if-else, or if-else-if statements within `if`, `if-else`, or `if-else-if` statements.
- The **Boolean expression** of the inner nested if statement is evaluated only if the Boolean expression of the outer if statement evaluates to `true`.
- A **multi-way selection** (`if-else-if`) is used when there are a series of expressions with different segments of code for each condition. Multi-way selection is performed such that no more than one segment of code is executed based on the first expression that evaluates to `true`. If no expression evaluates to true and there is a trailing else statement, then the body of the else is executed.

```
// 3 way choice with else if
if (boolean expression)
{
   statement1;
}
else if (boolean expression)
{
   statement2;
}
else
{
   statement3;
}
```

2.5 Compound Boolean Expressions

Learning Objectives:

- Develop code to represent compound Boolean expressions and determine the result of these expressions.

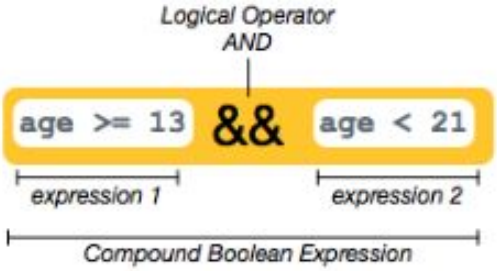
Evaluating logic expressions

Sometimes it is useful to use **nested if conditions**: if statements within if statements.

```
// if x is odd
if(x % 2 != 0){
    // if x is positive
    if(x > 0){
    ...
    }
}
```

Other times it makes more sense to use logical operators to check multiple conditions. We can combine the above nested if conditions using **logical operators**.

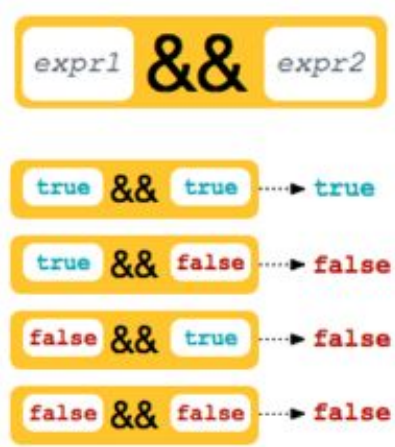
Logical operators && and || and !



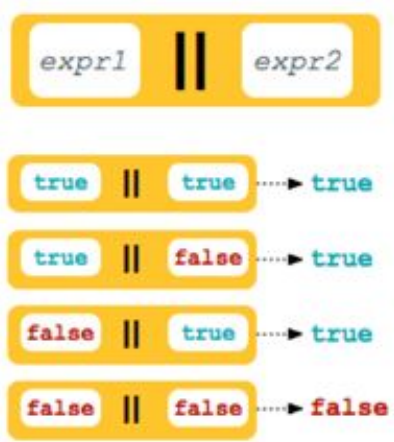
NOTE: the OR is made with two vertical "pipe" characters. The "pipe" is on the keyboard with same button as \ -- it's right next to the key with }]on it, just above the Return/enter key.

The logical operators -- also known as the Boolean Operators -- AND (&&), OR (||) and NOT (!) allow you to compare the results of more than one Boolean operation at a time.

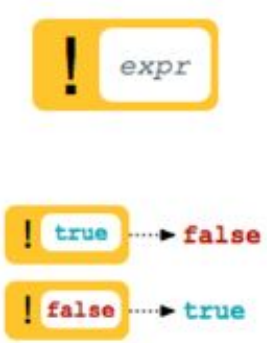
Logical AND



Logical OR



Logical NOT



Short Circuit Evaluation

Both **&&** and **||** use **short circuit evaluation**. That means that the second expression (on the right of the operator) isn't necessarily checked, if the result from the first expression is enough to tell if the compound boolean expression is **`true`** or **`false`**:

- When evaluating a logical **or** (**||**) and the first expression is **`true`**, then the second expression won't be executed, since only one needs to be **`true`** for the result to be **`true`**.

`true` || `anything` is `true`

if **first** is **true**, and it's an OR (**||**)then don't bother evaluating **second,** the whole expression is **true**!

- When evaluating a logical **and** (**&&**) and the first expression is **`false`**, then the second expression won't be executed. If the first expression is **`false`**, the result will be **`false`**, since both sides of the && need to be **`true`** for the result to be **`true`**.

`false` && `anything` is `false`

if **first** is **false**, and it's an AND (**&&**) then don't bother evaluating **second,** the whole expression is **false**!

Short Circuit Example

Dividing by 0 would crash the program!

BUT it doesn't in the following example due to short circuit evaluation since the second expression is not evaluated by the computer!

Errors:

MyProgram.java: Line 8: You may be dividing by zero.

```
int numSlices = 10;
int numPeople = 0;
```

false — not evaluated

```
if(numPeople != 0 && numSlices / numPeople > 0)
{
     System.out.println("There's enough pizza!");
}
else
{
     System.out.println("Not enough pizza.");
}
```

Output:

"Not enough pizza."

Order of Operations in Java

! is evaluated before &&
&& is evaluated before ||

Precedence	Operator	Description
First	( )	parenthesis
Second	++ -- ! (type)	unary operators, logical not, typecasting
Third	* / %	multiplication, division, modulus
Fourth	+ -	addition, subtraction, string concatenation
Fifth	< <= >= >	relational operators for greater/lesser
Sixth	== !=	relational operators for equality
Seventh	&&	logical and
Eighth	||	logical or
Ninth	= += -= *= /= %=	assignment operator

Summary

- Logical operators `!` (not), `&&` (and), and `||` (or) are used with Boolean values.
- `A && B` is `true` if both `A` and `B` are `true`.
- `A || B` is `true` if either `A` or `B` (or both) are `true`.
- `!A` is `true` if `A` is `false`.
- `!` has precedence (is executed before) `&&` which has precedence over `||`. Parentheses can be used to force the order of execution in a different way.
- An expression involving logical operators evaluates to a Boolean value.
- **Short-circuit evaluation** occurs when the result of a logical operation using `&&` or `||` can be determined by evaluating only the first Boolean expression. In this case, the second Boolean expression is not evaluated. (If the first expression is `true` in an `||` operation, the second expression is not evaluated since the result is `true`. If the first expression is `false` in an `&&` operation, the second expression is not evaluated since the result is `false`.)

2.6 Comparing Boolean Expressions

Learning Objectives:

- Compare equivalent Boolean expressions.
- Develop code to compare object references using Boolean expressions and determine the result of these expressions.

Truth Tables

A **truth table** is a table used to determine the truth values of a Boolean expression. We can evaluate boolean statements using truth tables.
Truth tables are a way of looking at all possible values of the variables, and determining the value of the whole statement.

To evaluate a boolean expression using truth tables:

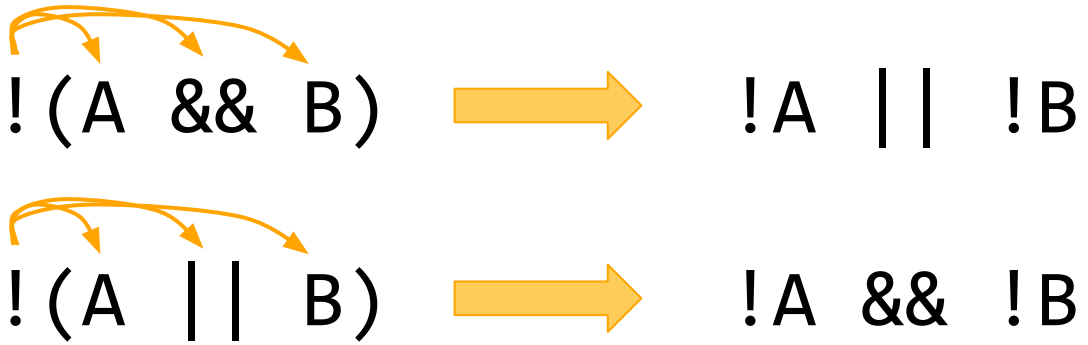
- Step 1: Write out all combinations of the terms
- Step 2: Evaluate the logic statement for each combination of terms

A	B	A && B
T	T	T
T	F	F
F	T	F
F	F	F

As our Boolean expressions become more complex, we may need to find ways to make them easier to understand and make our programs run more efficiently. **De Morgan's Laws** are a set of rules that describe how to simplify complex Boolean expressions.

De Morgan's Laws

- Move the NOT (**!**) inside
- AND (**&&**) becomes OR (**||**)
- OR (**||**) becomes AND (**&&**)



You can also simplify Boolean expressions that have relational operators like <, >, ==.

De Morgan's Laws

- Move the NOT (**!**) inside
- AND (**&&**) becomes OR (**||**)
- OR (**||**) becomes AND (**&&**)
- Flip the sign

```
!(x == y)   ⟹   (x != y)
!(x != y)   ⟹   (x == y)
 !(x < y)   ⟹   (x >= y)
 !(x > y)   ⟹   (x <= y)
!(x <= y)   ⟹   (x > y)
!(x >= y)   ⟹   (x < y)
```

De morgan's Example

Two software engineers have written these code segments to solve the same problem. Is the first code segment equivalent to the second code segment?

1

```
!(sara.getX() != 3 && sara.getY() != 5)
```

2

```
(sara.getX() == 3) || (sara.getY() == 5)
```

Yes! Using De Morgan's we can distribute the outside **!**, which flips the **!=** sign to **==** and **&&** becomes **||**

De Morgan's Practice Questions

Using De Morgan's Laws, what would be the equivalent of the expression below?

1. `!(x > 3 && y < 5)` → `x <= 3 || y >= 5`

2. `!(hasHeadlight || hasBikelight)` → `!hasHeadlight && !hasBikelight`

3. `!(sum < 5 || !(num > 0))` → `sum >= 5 && num > 0`

4. `!(!(num > 0) || temp <= 0) && num > 0` → `num > 0 && temp > 0 && num > 0`

5. `!(num > 0 || temp <= 0) && !(num > 0 && temp <= 0)`
   → `num <= 0 && temp > 0 && num <= 0 || temp > 0`

6. `!(!(num > 0 || temp <= 0) || !(num > 0) && temp <= 0)`
   → `num > 0 || temp <= 0 && num > 0 || temp > 0`

Properties to consider

When determining logical equivalence of expressions consider the following properties:

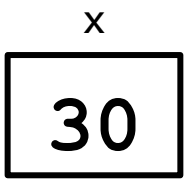
- Distributive Property: `A && (B || C) ≡ (A && B) || (A && C)`
- Associative Property: `A && (B && C) ≡ (A && B) && C`
- De Morgan's Laws: `!(A && B) ≡ !A || !B` and `!(A || B) ≡ !A && !B`
- Identity Properties: `A && true ≡ A` and `A || false ≡ A`

Difference Behind the Scenes

Primitives in code:

```
int x = 30;
```

In memory:

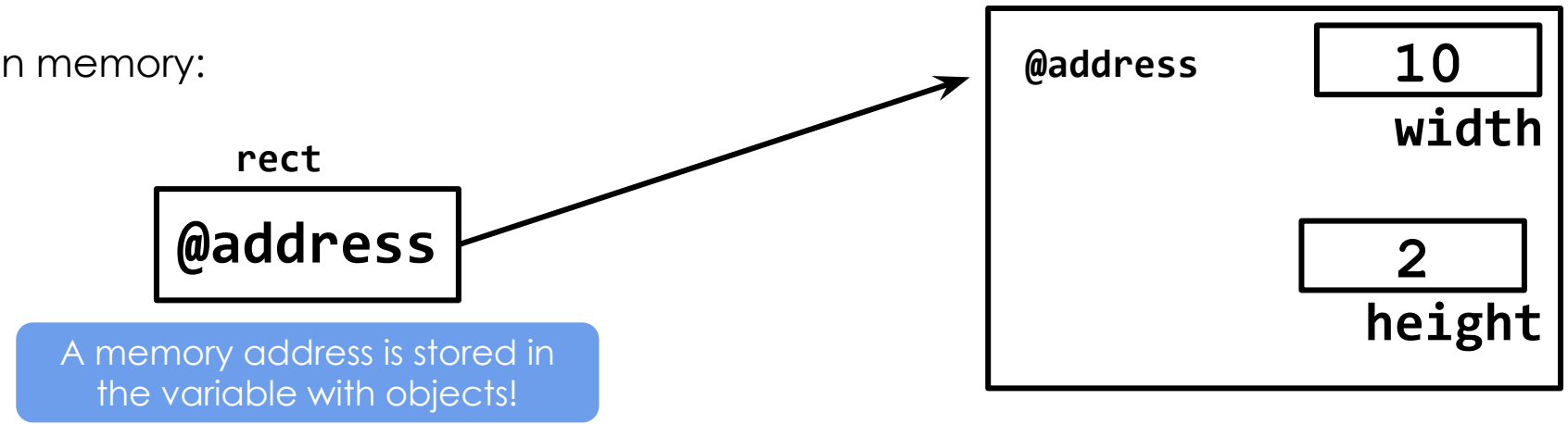


Differences Behind the Scenes

Objects in code:

```
Rectangle rect = new Rectangle(10, 2);
```

In memory:

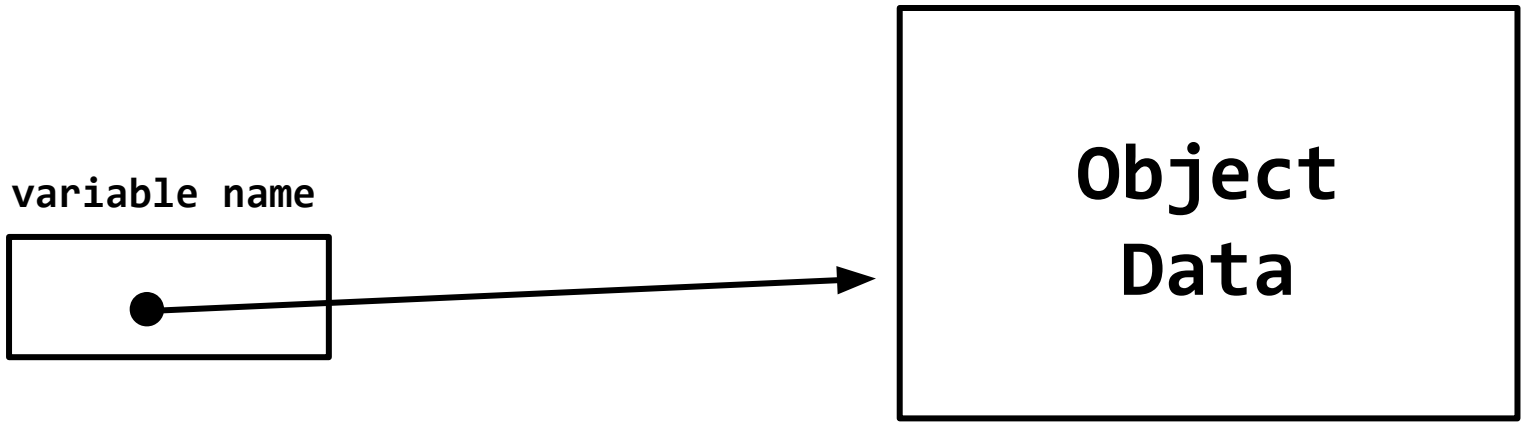


A memory address is stored in the variable with objects!

Object Storage

In memory, the variable simply stores a *location* or a *reference* to where the actual object data is located.

The variable *points* to the object data.

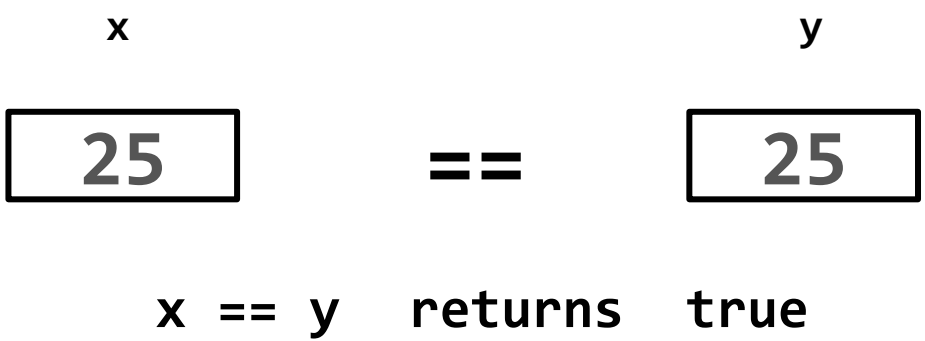


Comparing Primitives

With primitives, we can use `==` to compare

```
int x = 25;

int y = 25;
```

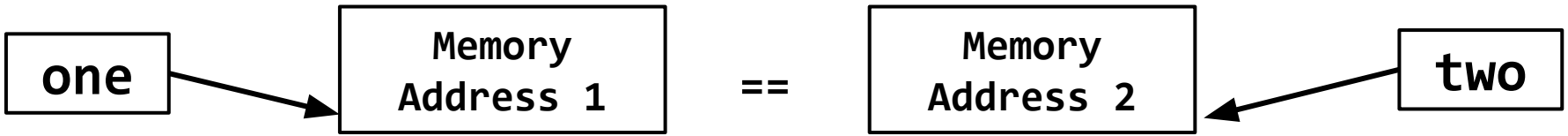


`x == y` returns `true`

Comparing Objects

With objects, == compares the *pointers* rather than the actual objects.

```
Rectangle one = new Rectangle(3,7);
Rectangle two = new Rectangle(3,7);
```



one == two returns false

Comparing Objects that are Aliases

Two objects are considered **aliases** when they both reference the same object. Comparing using **==** check whether two variables are aliases. Consider the Sprite class we discussed in Unit 2 used to represent a game character.

```
public class Aliases{
  public static void main(String[] args){
    Sprite player = new Sprite(30, 50);
    Sprite another = player;
    System.out.println(player == another); // true
  }
}
```

An **alias** is an object reference that refers to the same object as another variable. Here **player** and **another** are **aliases**.

Both object references **player** and **another** points to the same address hence the same object in memory.

Comparing Objects that are NOT Aliases

Two **different** objects can have the same attributes/data.

```
public class Aliases2{
  public static void main(String[] args){
    Sprite player = new Sprite(30, 50);
    Sprite another = new Sprite(30, 50);
    System.out.println(player == another); // false
    System.out.println(player != another); // true
  }
}
```

The references **player** and **another** above are two different Sprite objects(created individually using **new**) but both are located at the same coordinate.

EQUALS

We saw that for String objects, **==** is used to check if the two String references point to the same object whereas the equals method check if they have the same characters.

```
String a = "hi";
String b = new String("hi");
System.out.println(a == b); // false, different objects
System.out.println(a.equals(b)); // true
```

Later in Unit 5 when we write our own classes, it will be useful to implement the **equals** method for our class to check whether two different objects are equivalent(same data values).

For example, consider Point objects with attributes x and y representing points on the plane. Although the following two points are distinct programmatically, they are equivalent mathematically. The **equals** method will allow us to detect this. More on this later.

```
Point a = new Point(3,4);
Point b = new Point(3,4);
System.out.println(a == b); // false, different objects
System.out.println(a.equals(b)); // true
```

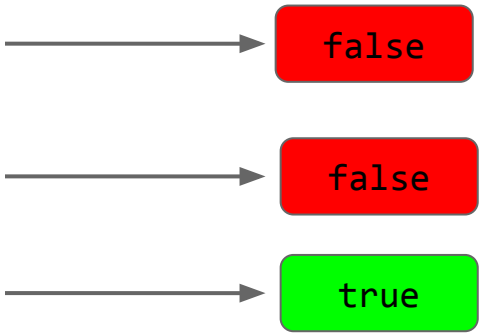
Reference Equality

```
String str1 = new String("dog");
String str2 = new String("dog");
String str3 = new String("cat");
String str4 = str1;

// Print out the results of various equality checks


System.out.println(str1 == str2);
System.out.println(str1 == str3);
System.out.println(str1 == str4);
```

str1 and **str4** are aliases!



Logical Equality

```
String str1 = new String("dog");
String str2 = new String("dog");
String str3 = new String("cat");
String str4 = str1;

// Print out the results of various equality checks using equals()
```

str1 and **str4** are aliases!

```
System.out.println(str1.equals(str2));
```
→ true

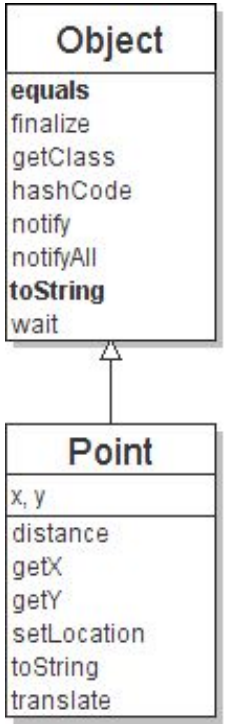
```
System.out.println(str1.equals(str3));
```
→ false

```
System.out.println(str1.equals(str4));
```
→ true

Recall: The Cosmic SuperClass Object

At the top of the hierarchy for every object is the **`Object`** class. This makes the **`Object`** class the superclass of all other classes in Java. The **`Object`** class is part of the built in **`java.lang`** package.

- Every class implicitly extends **`Object`**



There are about 11 methods in the **`Object`** class that can be inherited by any object. We are going to focus on 2 of these (both of which are included in the Java Quick Reference):

- **`public String toString()`**
  Returns a text representation of the object, often so that it can be printed. We have seen this in Unit 5.
- **`public boolean equals(Object other)`**
  Compare the object to any other for equality. Returns **`true`** if the objects have equal state.

Example: Rectangle Equals Method

Going back to our rectangle example, here is a possible equals method for a Rectangle class. You can see that two rectangles are considered equal if they have the same width and height.

```
// Checks if another Rectangle object
// has the same values for width and height
public boolean equals(Rectangle other)
{
    return other.getWidth() == width && other.getHeight() == height;
}
```

So now, when you use the equals method to compare rectangles one and two, what's actually being compared are the attributes width and height, not the memory addresses.

```
Rectangle rect1 = new Rectangle(10,5);
Rectangle rect2 = new Rectangle(10,5);
boolean equalRects = rect1.equals(rect2);
```

Null

We can also use **==** with objects to determine if they actually reference an object at all. We learned in previous lessons about the keyword **null**, which indicates that an object has yet to be initialized(**new** keyword). We can compare objects to the keyword **null** to determine if the object is actually a reference to anything.

```
object == null //true if no reference object
               //false if there is a reference object
```

Null Exception example

```
public class StringChecker {
   public static void main(String[] args) {
      String str1 = null; // str1 is not initialized (it's null)
      String str2 = "Computer Science"; // str2 is initialized with a string

      // Check if last character of str1 is "!"
      if (str1.indexOf("!") != -1) { // NullPointerException, str1 is null
         System.out.println("String ends with an exclamation!");
      } else {
         System.out.println("String DOES NOT end with an exclamation!");
      }

      // Check if last character of str2 is "!"
      if (str2.indexOf("!") != -1) { // NO EXCEPTION, but only because str2 is not null
         System.out.println("String ends with an exclamation!");
      } else {
         System.out.println("String DOES NOT end with an exclamation!");
      }
   }
}
```

Exception Fixed example

```
public class StringChecker {
   public static void main(String[] args) {
      String str1 = null; // str1 is not initialized (it's null)
      String str2 = "Computer Science"; // str2 is initialized with a string

      // Check if last character of str1 is "!"
      if (str1 != null && str1.indexOf("!") != -1) { //NO EXCEPTION
         System.out.println("String ends with an exclamation!");
      } else {
         System.out.println("String DOES NOT end with an exclamation!");
      }

      // Check if last character of str2 is "!"
      if (str2 != null && str2.indexOf("!") != -1) { // NO EXCEPTION
         System.out.println("String ends with an exclamation!");
      } else {
         System.out.println("String DOES NOT end with an exclamation!");
      }
   }
}
```

Summary

- Two Boolean expressions are **equivalent** if they evaluate to the same value in all cases. **Truth tables** can be used to prove Boolean expressions are equivalent.
- De Morgan's Laws can be applied to Boolean expressions to create equivalent ones:
  - `!(a && b)` is equivalent to `!a || !b`
  - `!(a || b)` is equivalent to `!a && !b`
- A negated expression with a relational operator can be simplified by flipping the relational operator to its opposite sign.
  - `!(c == d)` is equivalent to `c != d`
  - `!(c != d)` is equivalent to `c == d`
  - `!(c < d)` is equivalent to `c >= d`
  - `!(c > d)` is equivalent to `c <= d`
  - `!(c <= d)` is equivalent to `c > d`
  - `!(c >= d)` is equivalent to `c < d`
- Two different variables can hold references to the same object. Object references can be compared using `==` and `!=`. (Two object references are considered **aliases** when they both reference the same object.)

Summary continued...

- A reference value can be compared with `null`, using `==` or `!=`, to determine if the reference actually references an object.
- A `NullPointerException` is thrown when you try to perform an operation (such as calling a method or accessing a field) on an object that hasn't been initialized and is `null`
- Classes often define their own `equals` method, which can be used to specify the criteria for equivalency for two objects of the class. The equivalency of two objects is most often determined using attributes from the two objects.

2.7 while Loops

Learning Objectives:

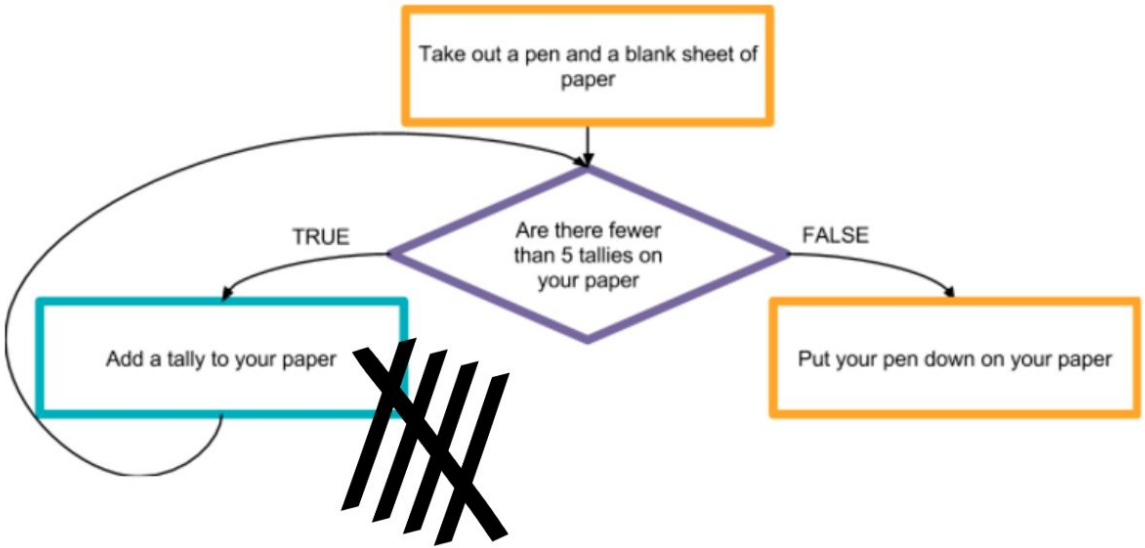
- Identify when an iterative process is required to achieve a desired result.
- Develop code to represent iterative processes using while loops and determine the result of these processes.

Loops

A **loop** in programming, also called **iteration** or **repetition**, is a way to repeat one or more statements. If you didn't have loops to allow you to repeat code, your programs would get very long very quickly!

Using a sequence of code, selection (ifs), and repetition (loops), the **control structures** in programming, you can construct an algorithm to solve almost any programming problem.

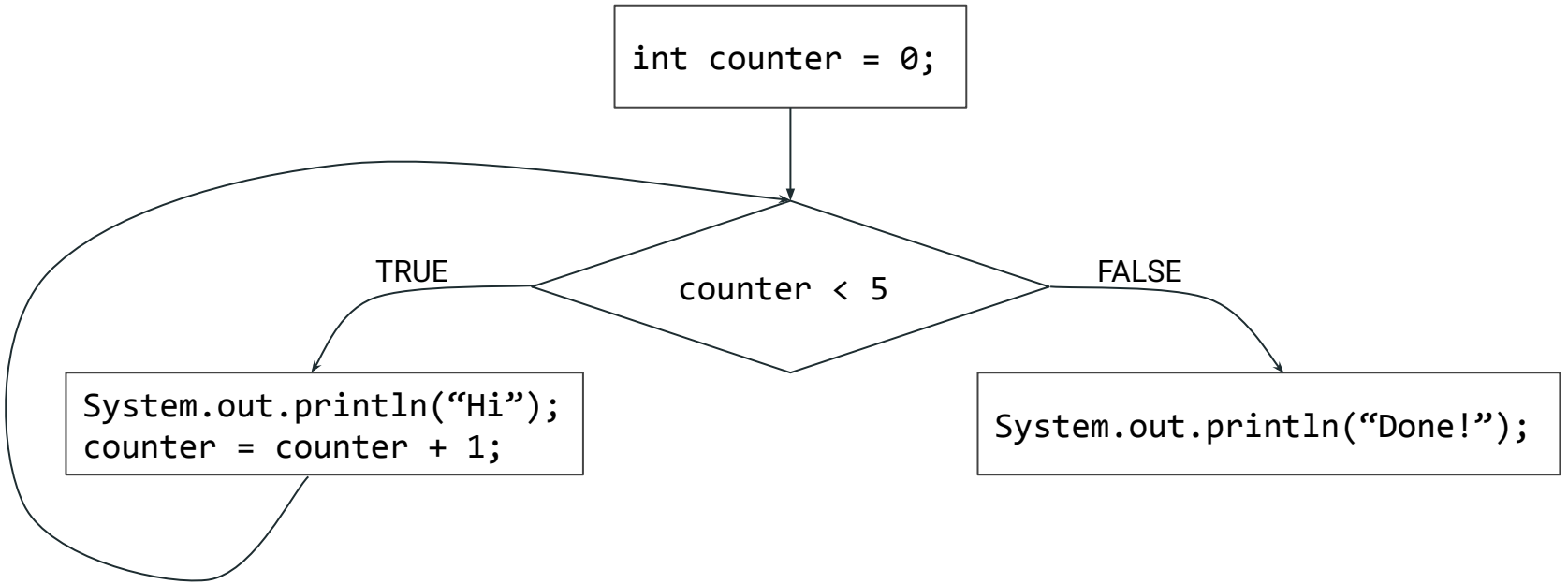
Iteration Flowchart



Notice:

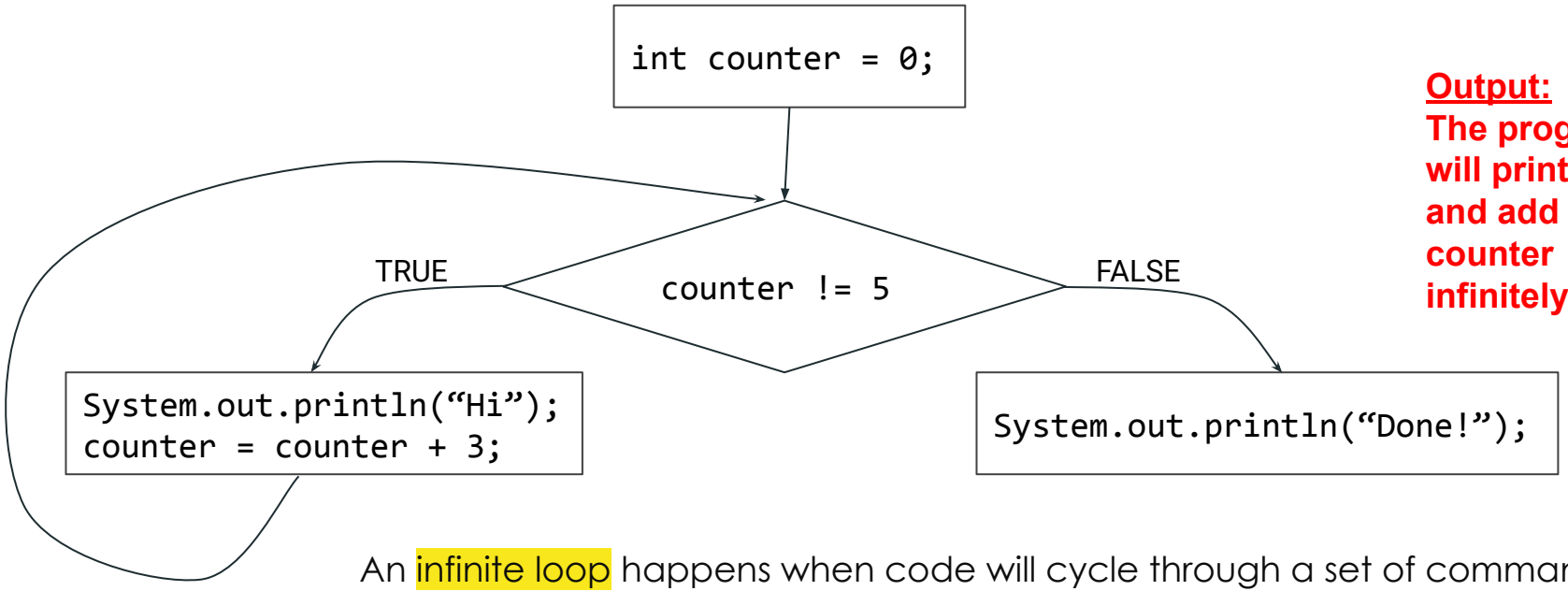
- This conditional statement loops back on itself. As a result, the conditional might be evaluated multiple times.
- As a result, the same command is running multiple times.

The below flowchart contains a “loop” that runs “while” a condition is true. How many times will the loop run? What is the output that would be generated from the program?



Output:
Hi!
Hi!
Hi!
Hi!
Hi!
Done!

The below flowchart contains a "loop" that runs "while" a condition is true. How many times will the loop run? What is the output that would be generated from the program?



Output:
The program will print "Hi!" and add 3 to the counter infinitely!

An infinite loop happens when code will cycle through a set of commands forever, since the loop condition is always true. Infinite loops are almost always undesirable.

The While Loop in Java Explained

Syntax

```
while (condition) {
  <instructions>
}
```

- While the **<condition>** is **true**, the computer keeps repeating the **<instructions>**.
- When the test condition is **false**, we exit the loop and continue with the statements that are after the body of the **while** loop.
- If the condition is **false** the first time you check it, the body of the loop will not execute.
- Conceptually, a **while** loop is very similar to an **if** conditional, except that a **while** is continually executed until it's no longer true and an **if** is only executed once.

Example

```
int i = 1;
while (i < 10) {

System.out.println(i);
  i++;
}
```

That code will print out the numbers 1 to 9, and then once **i** becomes 10 and is thus no longer less than 10, the computer will stop printing out the value of **i**.

The previous example is simple but it shows the basic common structure of while loops. Our code often starts off by initializing a variable (or more than one) before the while, then references that variable (or a related one) in the condition, and then modifies that variable in some way in the instructions.

RULE 1: Declare and initialize your loop's control variable

RULE 2: Test your loop control variable with the condition you want to verify is still true

RULE 3: Update your variable. Note: failing to update the variable inside the loop is probably the most common mistake!

```
//Rule 1
int counter = 0;

//Rule 2
while (counter <=10)
{
     statement1;
     statement2;
          .
          .
     counter++; //Rule 3
}
```

Example 1

What is the value of count after the loop?

```
public class Main{
    public static void main(String[] args)
    {
      // 1. initialize the loop variable
      int count = 1;
      // 2. test the loop variable
      while (count <= 5){
         System.out.println(count);
         // 3. change/update the loop variable
         count++;
      }
    }
}
```

Output:

1
2
3
4
5

Answer: 6

Example 2

What is the value of count after the loop?

```
public class Main{
    public static void main(String[] args)
    {
      // 1. initialize the loop variable
      int count = 1;
      // 2. test the loop variable
      while (count <= 5){
         // 3. change/update the loop variable
         count++;
         System.out.println(count);
      }
    }
}
```

Output:

2
3
4
5
6

Answer: 6

Writing While Loops

NOTE: Curly braces mark the body of methods, loops, and conditional blocks. They are not necessary if the body or the block consists of only one statement.

- for `while` statements, always make sure the `condition` is encapsulated in in open and closed parenthesis `()`
- without curly `{ }` braces to denote a while loop body, by default the body only contains one statement.

```
// A while statement
while (condition)
     statement;

//A while statement with curly braces {}
while (condition) {
     statement;
}

// A while statement with multiple statements -- must use {}
while (condition) {
     statement1;
     statement2;
          .
          .
          .
}
```

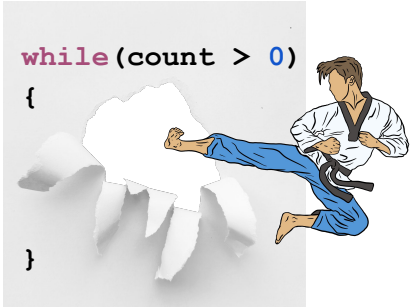
Break Statements

```
int counter = 0;
while(true)
{
    if(counter == 5)
    {
        break;
    }
    counter++;
}
```

while(true) will cause the loop to run forever, because the condition is always **true**.

The **break** statement allows us to exit the **while** loop and continue executing the program.

- Writing **while(true)** will make a program run infinitely.
- We can terminate while loops by adding a **break** statement.
- Break statements allow you to halt the execution and break out of the while loop.
- This will then execute statements that follow once the while loop terminates.



Break vs Return

We can also halt the execution of a loop by using the `return` keyword

```
public static void passwordCheck(String s)
{
  Scanner input = new Scanner(System.in);
  while(true)
  {
      if(s.equals("strongPassword"))
      {
        break;
      }
      System.out.println("Weak password, try again");
      s = input.nextLine();
  }
  System.out.println("Next line of code");
}
```

This line of code **will execute** because the break statement ends the while loop and continues to the next line of code following the while loop

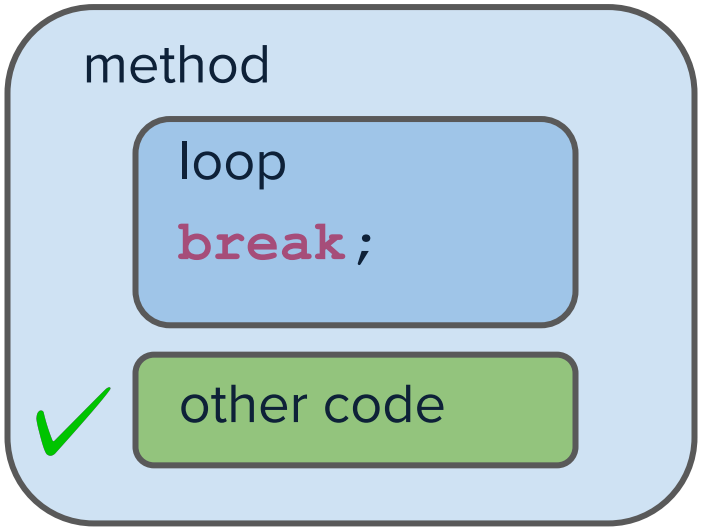
```
public static void passwordCheck(String s)
{
  Scanner input = new Scanner(System.in);
  while(true)
  {
      if(s.equals("strongPassword"))
      {
        return;
      }
      System.out.println("Weak password, try again");
      s = input.nextLine();
  }
  System.out.println("Next line of code");
}
```

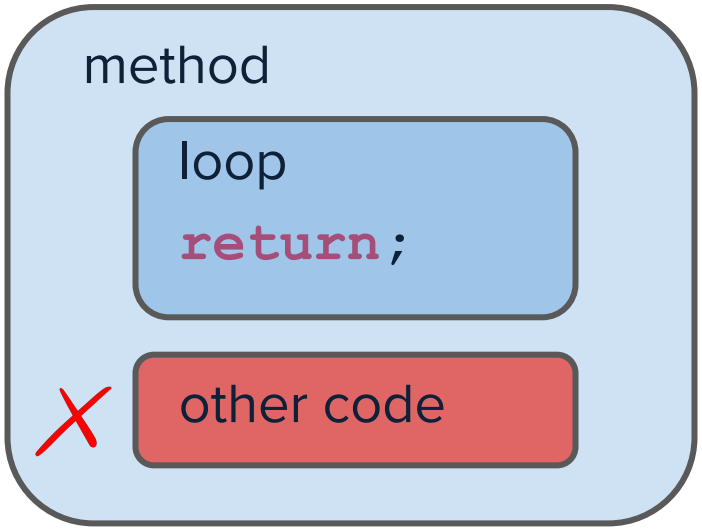
This line of code **will not** execute because the **`return`** statement exits the method or constructor regardless of what code follows the **`return`** statement or the **`while`** loop

Break vs. Return

The **break** statement will exit only the loop, other code after the loop will still execute.

the **return** statement will immediately exit the entire method, it will NOT run the other code in the method. Also if its a non-void method, it can also return a value as well.





Whether or not the programmer uses **break** or **return** statements will depend on the situation and the desired behavior.

Summary

- **Iteration statements** (loops) change the flow of control by repeating a segment of code zero or more times as long as the Boolean expression controlling the loop evaluates to true. Iteration is a form of repetition.
- Loops often have a **loop control variable** that is used in the boolean condition of the loop. Remember the 3 steps of writing a loop:
  - (Step 1) Initialize the loop variable
  - (Step 2) Test the loop variable
  - (Step 3) Update the loop variable
- A `while` loop is a type of iterative statement. In while loops, the Boolean expression is evaluated before each iteration of the loop body, including the first. When the expression evaluates to true, the loop body is executed. This continues until the Boolean expression evaluates to false, whereupon the iteration terminates.
- The loop body of an iterative statement will not execute if the Boolean expression initially evaluates to `false`.

Summary (continued...)

- An **infinite loop** occurs when the Boolean expression in an iterative statement always evaluates to `true`.
- **Off by one** errors occur when the iteration statement loops one time too many or one time too few.
- **Input-controlled loops** often use a sentinel value that is input by the user like "bye" or -1 as the condition for the loop to stop. Input-controlled loops are not on the AP CSA exam, but are very useful to accept data from the user.
- `break` is a keyword used to break out of a loop and executes statements that immediately follow the loop.
- `return` is a keyword used in methods to return a value back to the initial program that called the method.

2.8 for Loops

Learning Objectives:

- Develop code to represent iterative processes using for loops and determine the result of these processes.

While Loop Recap: 3 Parts

```
int steps = 0;

while(steps < 4){
    System.out.println("hi");
    steps++;
};
```

- A counting variable set to an initial value
- A Boolean expression which checks the condition of that variable
- A statement which increases or decreases the variable that is being checked.
    - Note: if this piece is missing, you may create an **infinite loop** that never stops running, or crashes your browser!

A for loop combines these three parts into one statement

```
int steps = 0;

while(steps < 4){
    System.out.println("hi");
    steps++;
}
```

Any variable name can be used here. It's most common to use **i**.

```
for (int i=0; i<4; i++){
    System.out.println("hi");
}
```

A loop is **iteration:** a repetitive portion of an algorithm which repeats a specified number of times or until a given condition is met.

The For Loop in Java Explained

Syntax

```
for (statement 1; statement 2; statement 3) {
  <instructions>
}
```

the is referred to as the **loop header**

Statement 1 is executed (one time) before the execution of the code block. It is often referred to as the initialization, because it initializes the counter variable to a starting value. This counter variable is used throughout the loop and keeps track of the current repetition, and is typically named i.

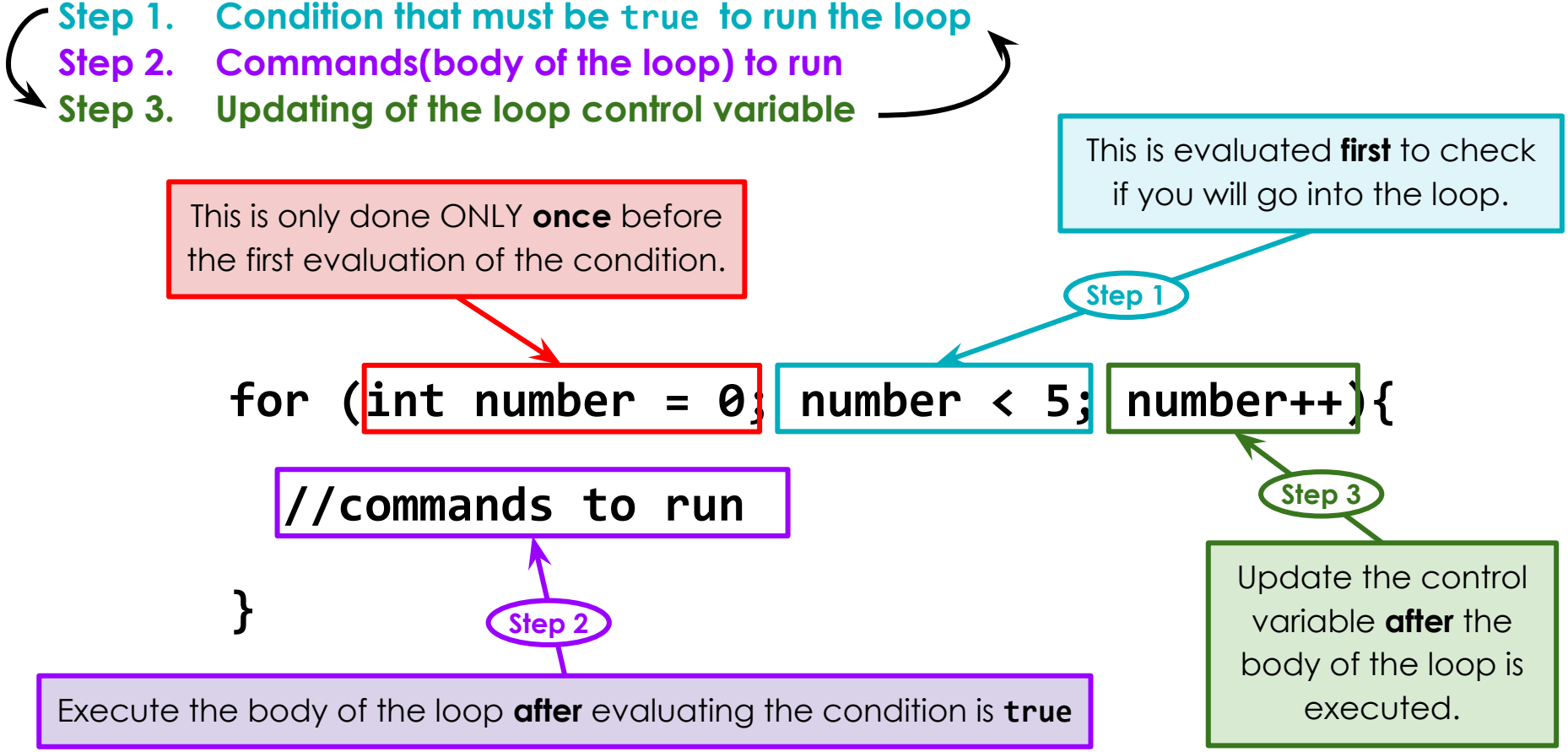
Statement 2 defines the condition for executing the code block. The condition part tells the computer whether to keep repeating or not. The computer evaluates the condition each time, and if the expression is true, it executes the inside instructions. If not, it exits the loop entirely.

Statement 3 is executed (every time) **after the code block** has been executed. It is often an increment/decrement which modifies the counter variable after each repetition.

A `for` loop consists of three parts:

- Declaring and initializing a loop control variable

Step 1. Condition that must be `true` to run the loop
Step 2. Commands(body of the loop) to run
Step 3. Updating of the loop control variable



Loop Tracing a For Loop Example

```
int sum = 0;

for(int i = 0; i <= 20; i += 5){
    sum += i;
}

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

output: **50**

sum	i
~~0~~	~~0~~ ✓
~~0~~	~~5~~ ✓
~~5~~	~~10~~ ✓
~~15~~	~~15~~ ✓
~~30~~	~~20~~ ✓
50	25 ✗

Off by one Loop Example

When a for loop iterates one too few or one too many times, it's referred to as an **off by one** error.

When we run the code below, it only prints 1-9, this is because loop incorrectly doesn't include the value 10 in the boolean expression. This is an example of an **Off by One Error**, because it doesn't account for the additional number.

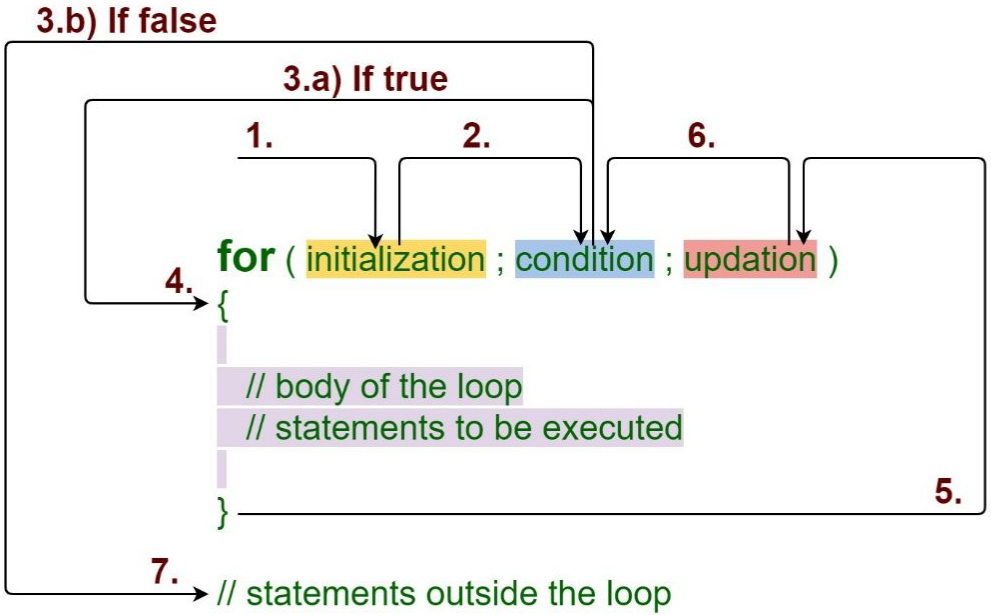
This program is meant to count from 1 - 10

```
for(int i = 1; i < 10; i++)
{
     System.out.println(i);
}
```

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

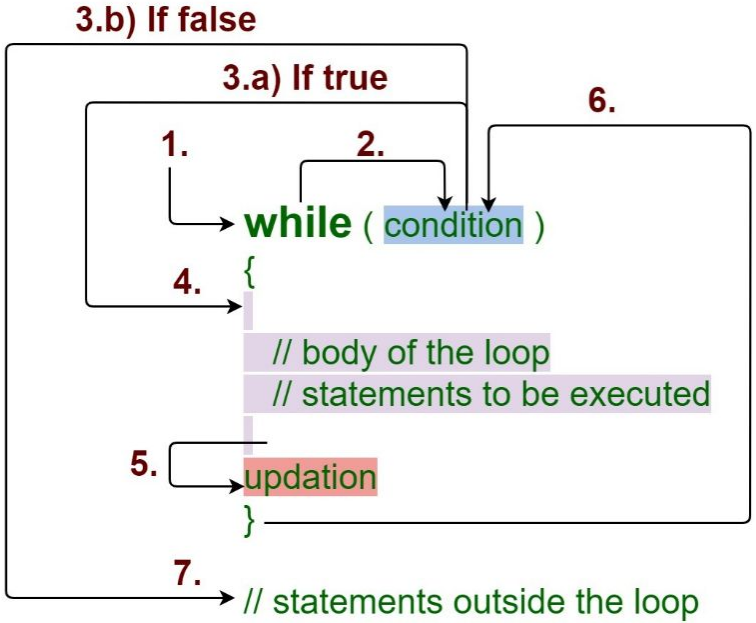
For Loop vs While Loop

For Loop



The for loop should be used if you know the amount of times you want something to execute, the for loop can repeat instructions a specific number of times.

While Loop



The while loop should be used if you want something to execute as long as a certain condition is true.

Categories of loops

indefinite(undetermined) loop: One where the number of times its body repeats is not known in advance.

- Prompt the user until they type a non-negative number.
- Print random numbers until a prime number is printed.
- Repeat until the user types "q" to quit.

The **`while` loop** is usually used for indefinite loops.

definite(predetermined) loop: Executes a known number of times.

- Print "hello" 10 times.
- Find all the prime numbers up to an integer *n*.
- Print each odd number between 5 and 127.

In this lecture, we will see that a **`for` loop** is often used to implement a definite loop.

For vs While

Write a loop to compute the sum: **1 + 2 + 3 + ... + 99 + 100**

```
int sum = 0;
int number = 1;
while(number <= 100){
    sum += number;
    number++;
}
```

```
int sum = 0;
for(int i = 1; i <= 100; i++){
    sum += i;
}
```

Both `for` and `while` loops can be used to solve this problem.

For vs. While

Although **for** and **while** loop can generally be interchangeable. It is best practice to use a **for** loop as a definite loop where the beginning and termination of the loop is well defined.

```
for(int i = 1; i <= 100; i++){
    System.out.println(i);
}
```

This **for** loop executes 100 times. It is a definite loop. It is usually better to use a **for** loop as a definite loop.

For vs. While

If the termination condition of a loop is less well defined, use a **while** loop. For example, suppose you require a positive integer input from the user. The following will loop until a positive number is entered.

```
Scanner inp = new Scanner(System.in);
System.out.print("Enter a positive number:");
int x = inp.nextInt();

while(x <= 0){
     System.out.print("Enter a positive number:");
     x = inp.nextInt();
}
System.out.print("Thank you!");
```

Sample Output:

```
Enter a positive number: -4
Enter a positive number: -5
Enter a positive number: -10
Enter a positive number: 6
Thank you!
```

This **while** loop executes an unknown number of times since you don't know when the user will enter a positive number. It is a indefinite loop. It is better to use a **while** loop here.

Summary

- A for loop is a type of iterative statement. There are three parts in a for loop header: the initialization (of the loop control variable or counter), the Boolean expression (testing the loop variable), and the update (to change the loop variable).
- In a for loop, the initialization statement is only executed once before the first Boolean expression evaluation. The variable being initialized is referred to as a **loop control variable**.
- The for loop Boolean expression is evaluated immediately after the loop control variable is initialized and then followed by each execution of the increment (or update) statement until it is false.
- In each iteration of the for loop, the update is executed after the entire loop body is executed and before the Boolean expression is evaluated again.
- A for loop can be rewritten into an equivalent while loop (and vice versa).
- **Off by One Error** happens when a for loop iteration is off by one too many or one too few.

2.9 Implementing Selection and Iteration Algorithms

Learning Objectives:

- Develop code for standard and original algorithms (without data structures) and determine the result of these algorithms.

Basic Loop Algorithms

Important basic algorithms that use loops:

1) Compute a sum of a series(list of numbers)
2) Determine the frequency with which a specific criterion is met(for example, divisibility)
3) Identify the individual digits in an integer

We will do an example of each of the above.

Compute a sum

Write a loop to compute the sum: 1 + 2 + 3 + … + 99 + 100

```
int sum = 0;
int number = 1;
while(number <= 100){
    sum += number;
    number++;
}
```

cumulative sum: A variable that keeps a sum in progress and is updated repeatedly until summing is finished.

- The **`sum`** in the above code computes a cumulative sum.

Determine Frequency

Write a loop to determine how many numbers from 1327 to 4542 that are multiples of 3 but not multiples of 5.

```
int count = 0;
int num = 1327;
while(num <= 4542){
    if(num % 3 == 0 && num % 5 != 0){
        count++;
    }
    num++;
}
```

Extracting Digits

Code that can extract digits from a number is useful (last four digits of a social security number, whether digits form a valid credit card number)

The trick is to repeatedly modulo 10 and integer divide by 10. For example:

```
int num = 1347;
int ones = num % 10; // extract the last digit: 7
num /= 10; // remove the last digit, num = 134
int tens = num % 10;   // extract the last digit: 4
num /=10;  // remove the last digit again, num = 13
int hundreds = num % 10;  // extract the last digit: 3
num /= 10;   // num = 1
int thousands = num % 10; // 1
```

Extracting Digits

For numbers of arbitrary lengths, we can use a while loop to implement the previous algorithm!

```
Scanner console = new Scanner(System.in);
System.out.print("Enter number: ");
int number = console.nextInt();
while(number != 0){
    // extract last digit
    System.out.println(number % 10);
    number /= 10; // remove last digit
}
```

Output:

```
Enter a number: 2348
8
4
3
2
```

2.10 Implementing String Algorithms

Learning Objectives:

- Develop code for standard and original algorithms that involve strings and determine the result of these algorithms.

String Traversal

0	1	2	3	4	5	6	7	8
P	r	i	n	t		M	e	!

String traversing is the process of going through a String one character at a time, often using loops! The **substring()** method is particularly useful when attempting to traverse Strings. We can traverse Strings using loops, here's an example:

```
//Prints each character in a String on a new line
String print = "Print Me!";

for(int i = 0; i < print.length(); i++)
{
    System.out.println(print.substring(i, i+1));
}
```

Isolates each character of the String **print**

Initialize the variable **i** to **0** since the first index is **0**

print.length() = 9, but **Strings** start at index **0**. We put **i < print.length()** so **i** only goes up to the last index!

We want to increase **i** by **1** to access each character individually

IndexOutofBounds

0	1	2	3	4	5	6	7	8
P	r	i	n	t		M	e	!

If we were to make **`i <= print.length()`**, we would get an **`IndexOutofBounds Error`**, because we are attempting to access a **`String`** index that does not exist.

```
//Prints each character in a String on a new line
String print = "Print Me!";

for(int i = 0; i <= print.length(); i++)
{
    System.out.println(print.substring(i, i+1));
}
```

when **`i = 9`**, then the condition **`i <= print.length()`** is **`true`** since the **`length`** is **`9`**, so **`9 <= 9`** is **`true`**. But, then when trying to **`substring(9, 10)`** will give an error since there's no such index **`9`** that we can return a **`substring`** for.

The fix: when traversing make sure you use either:

- **`i < string.length()`**
  or
- **`i <= string.length() - 1`**

String Algorithms: For Loops

Loops allow us to do **String** traversals. Suppose we want to know the number of spaces in a **String**.

```
public static int countSpaces(String str){
    int count = 0;
    for(int i = 0; i < str.length(); i++){
        if(str.substring(i, i + 1).equals(" "))
            count++;
    }
    return count;
}
```

String Algorithms: For Loops

Assume that a given **String** is a sequence of words separated by spaces, return the number of words. Use **countSpaces**.

```
public static int numOfWords(String str){
    return countSpaces(str) + 1;
}
```

Reversing a String

Create a method that reverses the order of a String:

```
public static String reverse(String string)
{
   String newString = "";
   for(int i = string.length() - 1; i >= 0; i--)
   {
      String character = string.substring(i, i+1);
      newString += character;
   }
   return newString;
}
```

Remove Spaces (While)

Given a string, return the string with all of its spaces removed.

We will do this first with a **`while` loop**.

One way to do this is to use a **`while`** loop with **`indexOf`**.

```
public static String removeSpaces1(String str){
    while(str.indexOf(" ") != -1){
       int indexSpace = str.indexOf(" ");
       String first = str.substring(0, indexSpace);
       String second = str.substring(indexSpace + 1);
       str = first + second; // first space removed
    }
    return str;
}
```

Remove Spaces (For)

Given a string, return the string with all of its spaces removed.

We now implement this with a **for loop**. Start with an empty string. Then examine every letter and if it is NOT a space, add it to the string. This builds up the String one letter at a time but avoids the spaces. For some students, this may be easier to understand and write then the previous while loop version.

```
public static String removeSpaces1(String str){
    String ans = "";
    for(int i = 0; i < str.length(); i++){
        String letter = str.substring(i, i+1);
        if(!letter.equals(" "))
            ans += letter;
    }
    return ans;
}
```

Palindrome

Given a string, return whether the string is a palindrome(reads the same forward as backwards).

```
public static boolean isPalindrome(String str){
    int len = str.length();
    for(int i = 0; i < len; i++){
        String current = str.substring(i, i + 1);
        String opposite = str.substring(len – 1 - i, len - i);
        if(!current.equals(opposite))
            return false;
    }
    return true;
}
```

Can we make the code slightly more efficient?

Improved Palindrome

We can go just to the middle of the String.

```
public static boolean isPalindrome(String str){
    int len = str.length();
    for(int i = 0; i < len/2; i++){
        String current = str.substring(i, i + 1);
        String opposite = str.substring(len – 1 - i, len - i);
        if(!current.equals(opposite))
            return false;
    }
    return true;
}
```

Alternate Palindrome

Assuming the method **reverse()** is defined.

```
public static boolean isPalindrome(String str){
    String reverse = reverse(str);
    if(str.equals(reverse))
        return true;
    return false;
}
```

charAt Example

You can also use **charAt(index)** to access individual characters in a **String**. The following code prints each character on a separate line.

Code:

```
String name = "CodeHS";
for (int i = 0; i < name.length(); i++)
{
    System.out.println(name.charAt(i));
}
```

Output:

```
C
o
d
e
H
S
```

Summary

- Loops can be used to traverse or process a string.
- There are standard algorithms that utilize String traversals to:
  - Find if one or more substrings has a particular property
  - Determine the number of substrings that meet specific criteria
  - Create a new string with the characters reversed

2.11 Nested Iteration

Learning Objectives:

- Develop code to represent nested iterative processes and determine the result of these processes.

Nested Loops Example 1

Putting a loop inside another loop is called **nesting.**

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 1

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 1

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 1

number*line = 1

```
1
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 2

The inner loop needs to finish executing BEFORE the outer loop moves to the next iteration!

```
1
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 2

```
1
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 2

number*line = 2

```
1   2
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 3

```
1   2
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 3

```
1   2
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 3

number*line = 3

```
1   2   3
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 4

```
1   2   3
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 4

```
1   2   3
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 4

number*line = 4

```
1   2   3   4
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 5

```
1   2   3   4
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 5

```
1   2   3   4
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 5

number*line = 5

```
1   2   3   4   5
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 6

```
1   2   3   4   5
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

number = 6

```
1   2   3   4   5
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 1

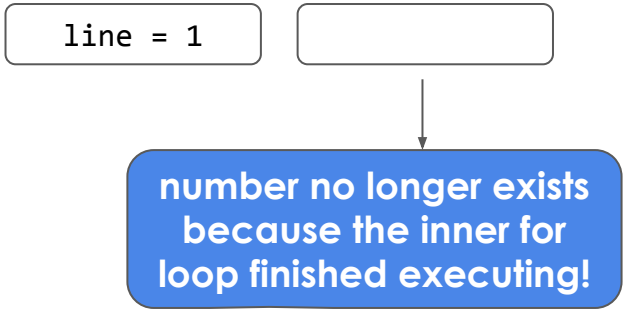
number =

Now we exit the for loop and move to the code that directly follows the inner loop. In this case, we are adding a println() statement so that the next line of code printed to the console is on a new line.

```
1   2   3   4   5
```

Nested Loops Example 1

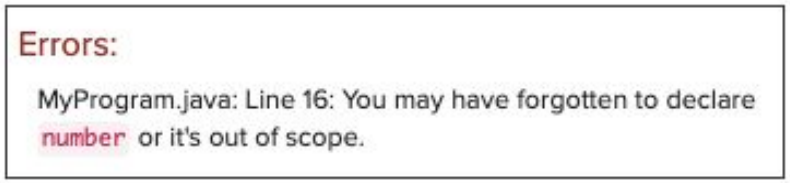
```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

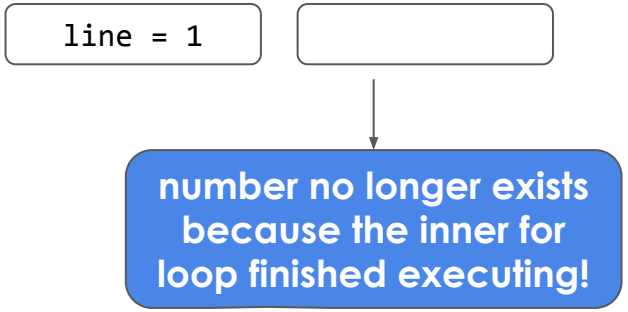



```
1   2   3   4   5
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
    number++;
}
```






```
1   2   3   4   5
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

Now the outer loop finally increments, increasing the value of line

```
1   2   3   4   5
```

Nested Loops Example 1

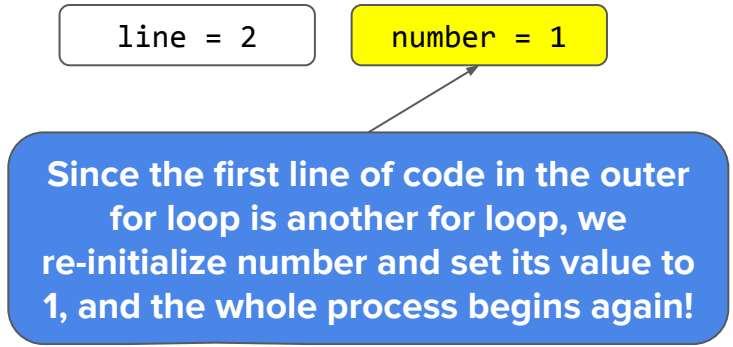
```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

```
1   2   3   4   5
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```




```
1   2   3   4   5
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

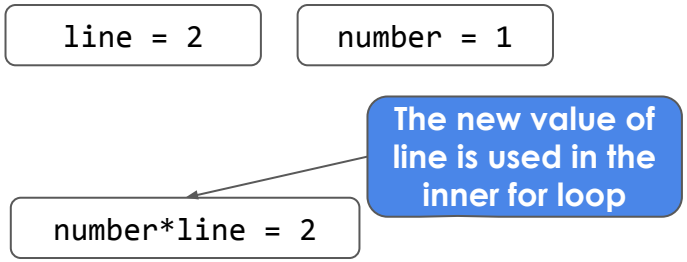
line = 2

number = 1

```
1   2   3   4   5
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```




```
1   2   3   4   5
2
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 2

```
1  2  3  4  5
2
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 2

```
1  2  3  4  5
2
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 2

number*line = 4

```
1  2  3  4  5
2  4
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 3

```
1  2  3  4  5
2  4
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 3

```
1  2  3  4  5
2  4
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 3

number*line = 6

```
1  2  3  4  5
2  4  6
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 4

```
1  2  3  4  5
2  4  6
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 4

```
1  2  3  4  5
2  4  6
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 4

number*line = 8

```
1  2  3  4  5
2  4  6  8
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 5

```
1  2  3  4  5
2  4  6  8
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 5

```
1  2  3  4  5
2  4  6  8
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 5

number*line = 10

```
1  2  3  4  5
2  4  6  8  10
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

number = 6

```
1  2  3  4  5
2  4  6  8  10
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2

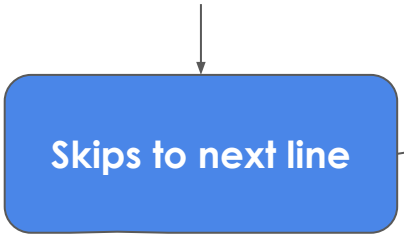
number = 6

```
1  2  3  4  5
2  4  6  8  10
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 2




```
1  2  3  4  5
2  4  6  8  10
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.print(number*line + " ");
    }
    System.out.println();
}
```

line = 3

Now that the outer for loop has completed, it returns to the increment, and increases the value of line by 1. This process repeats each time until the value of line is greater than or equal to 6.

```
1  2  3  4  5
2  4  6  8  10
3  6  9  12 15
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 4

Now that the outer for loop has completed, it returns to the increment, and increases the value of line by 1. This process repeats each time until the value of line is greater than or equal to 6.

```
1  2  3  4  5
2  4  6  8  10
3  6  9  12 15
4  8  12 16 20
```

Nested Loops Example 1

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

line = 5

Now that the outer for loop has completed, it returns to the increment, and increases the value of line by 1. This process repeats each time until the value of line is greater than or equal to 6.

```
1  2  3  4  5
2  4  6  8  10
3  6  9  12 15
4  8  12 16 20
5  10 15 20 25
```

Nested Loops Example 1

Note: the entire inner loop runs for each iteration of the outer loop.

```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```

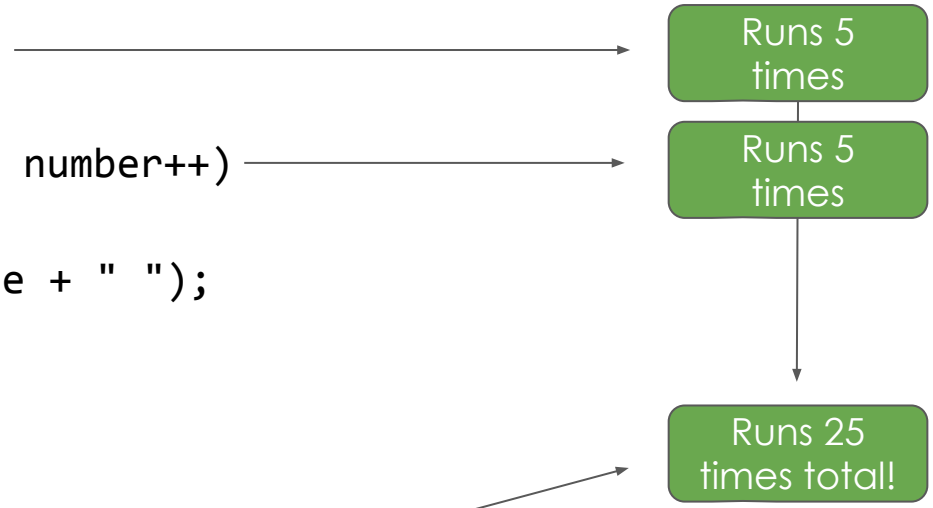
line = 6

Program stops!

```
1   2   3   4   5
2   4   6   8   10
3   6   9   12  15
4   8   12  16  20
5   10  15  20  25
```

Nested Loops Example 1

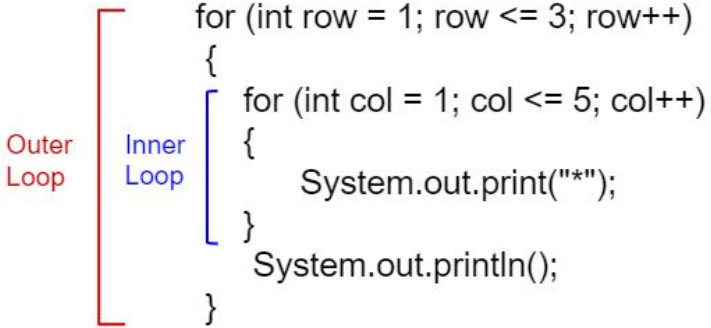
```
for(int line = 1; line < 6; line++)
{
    for(int number = 1; number < 6; number++)
    {
        System.out.print(number*line + " ");
    }
    System.out.println();
}
```



The number of times a nested loop will run is calculated by the number of iterations in the inner loop * iterations of the outer loop. This can be a helpful calculation when trying to figure out what to set your for loop increment and boolean conditions to.

Nested Loops

- Having a loop inside a loop is what is referred to as **nested loops**
  - Important facts:
    - the **inner loops** runs as many times as the outer loop runs
    - the **inner loop** must finish for the next iteration in the outer loop to occur
  - Let's look at more examples…



Variable Scope

Variable scope refers to where in a program a variable can be accessed or used. A variable declared inside a method or block { } only exists and can be used within that method or block.

```
int outsideLoop;
for(int outer = 0; outer < 5; outer++)
{
    //code
    for(int inner = 0; inner < 5; inner++)
    {
        //code
    }
}
//code
```

As stated earlier, it's very important to pay attention to the placement of variables in your program, especially as we continue to add control structures that have different levels of variable access.

Variable Scope

Because the variable **outsideLoop** is outside of both loops and is executed prior to the rest of the code, **outsideLoop** can be used in both for loops, and in any code that follows both for loops.

```
int outsideLoop;
for(int outer = 0; outer < 5; outer++)
{
    //code
    for(int inner = 0; inner < 5; inner++)
    {
        //code
    }
}
//code
```

Variable Scope

The variable **outer** is initialized in the outer for loop, and can only be used within the brackets of the **outer** for loop. Because the inner for loop is inside of the outer for loop, the variable **outer** can be used in the inner for loop.

```
int outsideLoop;
for(int outer= 0; outer<5; outer++)
{
    //code
    for(int inner= 0; inner<5; inner++)
    {
        //code
    }
}
//code
```

Variable Scope

The variable **inner** can be used only between the brackets that enclose the **inner** for loop. If the variable **inner** is used outside of the inner for loop, an error will occur.

```
int outsideLoop;
for(int outer= 0; outer<5; outer++)
{
    //code
    for(int inner= 0; inner<5; inner++)
    {
        //code
    }
}
//code
```

Here are two examples of nesting using while loops, and a while/for loop combination. This is the same times table problem that we just solved, just written using different control structures.

Nested While Loops

```
int line = 1;
while(line < 6)
{
     int number = 1;
     while(number < 6)
     {
          System.out.print(number*line + " ");
          number++;
     }
     System.out.println()
     line++;
}
```

Nested While/For Loops

```
int line = 1;
while(line < 6)
{
     for(int number = 1; number < 6; number++)
     {
          System.out.print(number*line + " ");
     }
     System.out.println()
     line++;
}
```

Summary

- Nested iteration statements are iteration statements that appear in the body of another iteration statement.
- When a loop is nested inside another loop, the inner loop must complete all its iterations before the outer loop can continue.

2.12 Informal Run-Time Analysis

Learning Objectives:

- Calculate statement execution counts and informal run-time comparison of iterative statements.

Measuring Algorithm Performance

The absolute running time of an algorithm can't be predicted since this depends on the:

- Programming language
- Computer
- Other programs running at the same time
- The quality of the operating system
- and many other factors.

A basic approach that we can take, is the **Statement Execution Count**(the number of times a statement is executed by the program). This suggests that we can count the number of instructions that algorithm has to execute. Once we know how many times a piece of code executes, we can have a relative way to compare 2 or more algorithms for efficiency. Algorithms with **lower** execution counts are likely to be more efficient.

Example Algorithm

Execution Count

13

```
public void computeSum ()
{
    int sum = 0;
    int n = 10;

    for (int i = 0; i < n; i++)
    {
        sum += i;
    }

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

First instruction executes once!

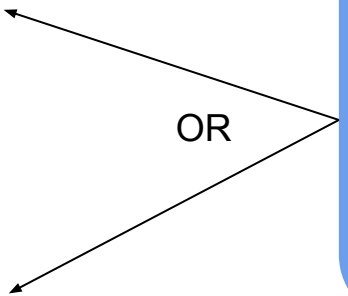
Second instruction executes once!

Third instruction executes n times!(in this case 10)

Fourth instruction executes once!

If you have…`if` Statements

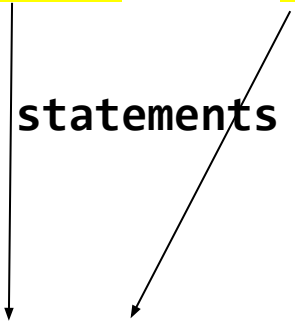
```
if (condition)
{
    sequence of statements
}
else
{
    sequence of statements
}
```



For conditional statements, either sequence 1 will execute, or sequence 2 will execute. So the count will include one of the 2 statement counts.

If you have…**for** loops

```
for (int i = initial; i < n; i++)
{
    sequence of statements
}
```



initial	n	Execution count
0	10	10 - 0 = 10
4	8	8 - 4 = 4

If the increment of a for loop is ++, the execution count is (n - initial)

If you have...Nested Loops

```
for (int i = initial; i < n; i++)
{
    for (j = init2; j < m; j++)
    {
        sequence of statements
    }
}
```

For nested loops, the execution count is (n - initial) * (m - init2)

initial	n	init2	m	Execution count
0	10	0	5	(10 - 0) * (5 - 0) = 50
4	8	1	4	(8 - 4) * (4 - 1) = 12

Some Examples

What's the total number of **x++** operations?

```
int x = 0;
for(int i = 0; i < 10; i++){
    x++;
}
for(int j = 1; j <= 15; j++){
    x++;
}
```

Answer: **10 + 15** = **25**

Some Examples

What's the total number of **x++** operations?

```
int x = 0;
for(int i = 0; i < 10; i++){
    for(int j = 0; j < 15; j++)
        x++;
}
```

Answer: **10 * 15** = **150**

Summary

- A **statement execution count** indicates the number of times a statement is executed by the program. Statement execution counts are often calculated informally through tracing and analysis of the iterative statements.
- A trace table can be used to keep track of the variables and their values throughout each iteration of the loop.
- The number of times a loop executes can be calculated by `largestVal - smallestVal + 1` where these are the largest and smallest values of the loop counter variable possible in the body of the loop.
- The number of times a nested for-loop runs is the number of times the outer loop runs times the number of times the inner loop runs.
- In non-rectangular loops, the number of times the inner loop runs can be calculated with the sum of natural numbers formula `n(n+1)/2` where n is the number of times the outer loop runs or the maximum number of times the inner loop runs.