Decisions and Loops

IIKG1002/IDG1011 – Front-end web development

Johanna Johansen
johanna.johansen@ntnu.no

Decisions

Custom name plaque

Enter name:

Ex: Ole Nordmann

SHOW COST

Custom name plaque

Enter name:

Please enter your name below …

Ex: Ole Nordmann

SHOW COST

Custom name plaque

Enter name:

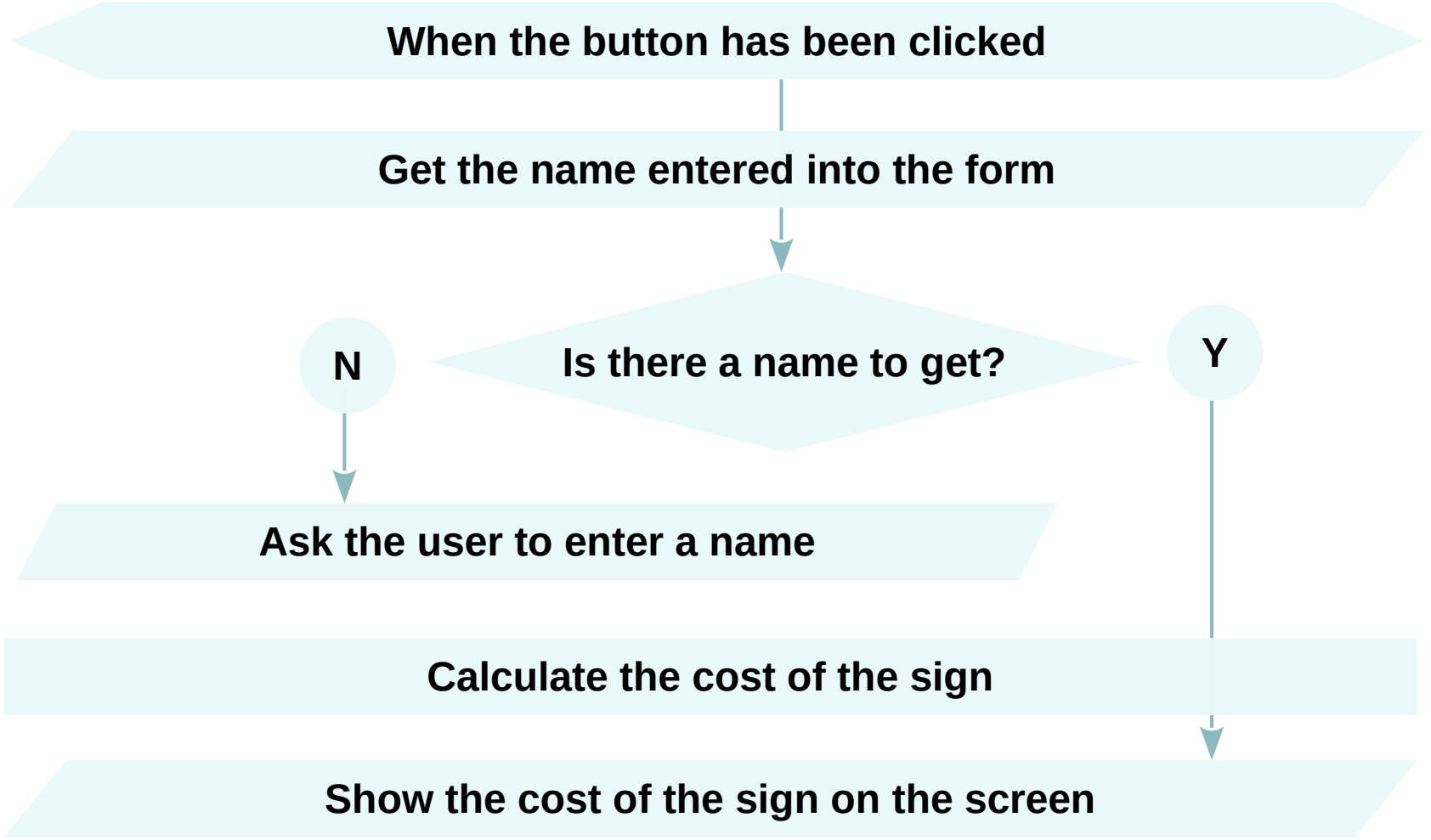
Johanna Johansen

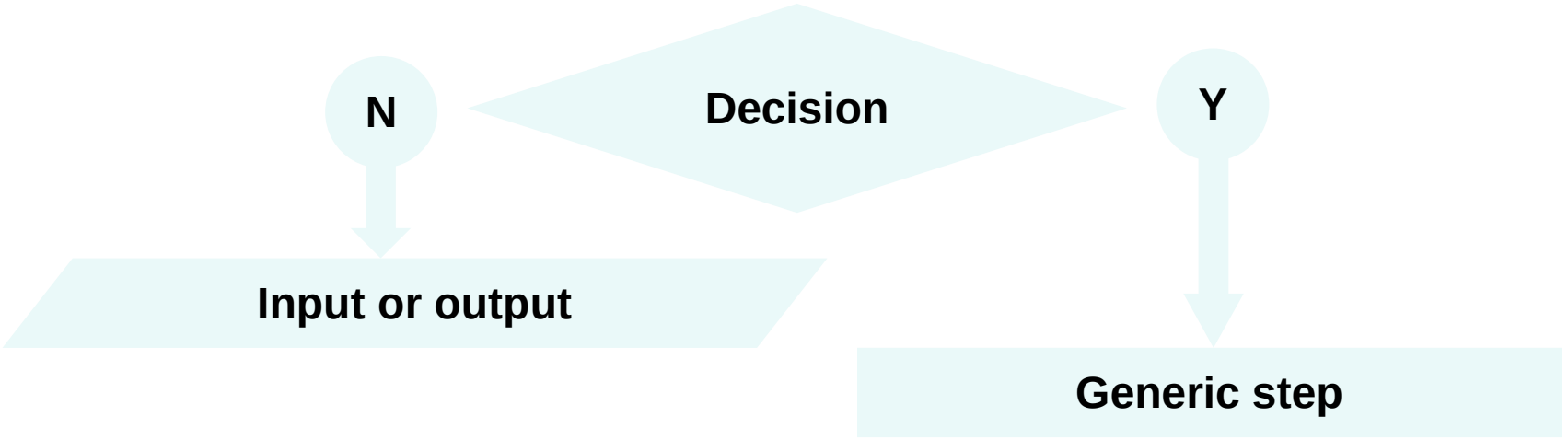
SHOW COST

Custom name plaque

Johanna Johansen

Price: 225,-


When the button has been clicked
Get the name entered into the form
Is there a name to get?
N
Y
Ask the user to enter a name
Calculate the cost of the sign
Show the cost of the sign on the screen


N
Decision
Y
Input or output
Generic step

Decision making

- scripts behave differently depending upon how the user interacts with the web page and/or the browser window itself
- there are several places in the script where decisions are made
  - determine which lines of code should be run next
  - the code can take more than one path
  - the browser runs different code in different situations
- flowcharts can help you plan for these occasions

Condition

- For determining which path to take, we set a **condition**
  - check if one value is equal to another, greater than another, or less than another
  - if the condition returns **true**, we take one path, if **false** we take another path
    - the condition results in one value, either true or false
    - the condition is an expression, i.e., results in one value
  - checking or testing a conditions is referred to as **evaluating the condition**

Condition

- Anatomy of a condition
  - in any condition there is usually one **operator** and two **operands**
    - the operands are placed on each side of the **comparison operator**
    - the operands can be values, variables, or expressions

```
3 == 4
score >= pass
(score1 + score2) > (pass1 + pass2)
```

Comparison operators

- `==` / `!=`
  - is equal to / is not equal to
  - the values can be numbers, strings, or Booleans
  - compares to values to see if they are / they are not the same

```
'Hello' == 'Goodbye' // evaluates to false

'Hello' == 'Hello'  // evaluates to true
```

```
'Hello' != 'Goodbye' // evaluates to true

'Hello' != 'Hello' // evaluates to false
```

Comparison operators

- `===` / `!==`
  - **strict** equal to / is not equal to
  - checks that both *data type* and *value* are / are not the same

```
'3' === 3 // evaluates to false
'3' === '3'  // evaluates to true
```

```
'3' !== 3 // evaluates to true
'3' !== '3' // evaluates to false
```

Comparison operators

- `>` / `<`
  - greater than / less than
  - checks if the number on the **left** is greater than / less than the number on the right

```
4 > 3 // evaluates to true
3 > 4 // evaluates to false
```

```
4 < 3 // evaluates to false
3 < 4 // evaluates to true
```

Comparison operators

- `>=` / `<=`
  - greater than or equal to / less than or equal to
  - checks if the number on the **left** is greater than or equal / less than or equal to the number on the right

```
4 >= 3 // evaluates to true
3 >= 4 // evaluates to false
3 >= 3 // evaluates to true
```

```
4 <= 3 // evaluates to false
3 <= 4 // evaluates to true
3 <= 3 // evaluates to true
```

Example

```
// evaluating two variables using a comparison operator to return true or false
let pass = 50; // Pass mark
let score = 90;  // Holds the users score

// Check if the user has obtained a passing mark
let hasPassed = score >= pass;  // evaluates to true; 90 is greater than 50

// write the message into a page
const el = document.getElementById('answer');
el.textContent = `Level passed: ${hasPassed}`; // Prints into the page: “Level passed: true”
```

Example taken from p. 155, Duckett syllabus book.

- The operands can be **expressions**, not just one value or one variable

```
/** The user takes a test in two rounds.
  * The result from adding the score of the user from both rounds are compared to the
  result from adding the height scores for the test,
  * to see if the respective user has exceeded the initial highest scores.
  */
 let score1 = 90;     // the score of the user from the round one of the test
 let score2 = 95;     // the score of the user from the second round of the test
 let highestScore1 = 75; // the highest score so far for the first round of the test
 let highestScore2 = 95; // the highest score so far for the second round of the test

 // comparing two expressions
 let comparison = (score1 + score2) > (highestScore1 + highestScore2);

 // Write the result of the comparison to the page
 const el = document.getElementById('answer');
 el.textContent = `New high score: ${comparison}`;
```

Logical operators

- used to test multiple conditions
  - each condition returns either true or false
- AND (&&), OR (||), NOT (!)

```
/* The student is evaluated in two disciplines and needs to get a passing
score for both in order to go up to the second year */

(score1 >= pass1) && (score2 >= pass2)
```

  - `score1 >= pass1` is one expression / condition / operand, can evaluate to either true or false
  - `score2 >= pass2` is one expression / condition / operand, can evaluate to either true or false
  - joined together with the **&&**, they form an expression that can evaluate to either true or false

Logical operators

- `&&`
  - logical AND
  - if **all** operands evaluate to true, the expression returns true
    - `true && true` evaluates to true
  - if **just one** of these returns false, the expression returns false
    - `true && false` evaluates to false
    - `false && true` evaluates to false
    - `false && false` evaluates to false

```
(2 < 5) && (3 >=2);
```

- 2 < 5 evaluates to true
- 3 >= 2 evaluates to true
- true && true evaluates to true

Logical operators

- `||`
  - logical OR
  - tests at least one condition
  - if **either** expression evaluates to true, the expression returns true
    - `true || true` evaluates to true
    - `true || false` evaluates to true
    - `false || true` evaluates to true
  - if **both** return false, the expression will be false
    - `false || false` evaluates to false

```
(2 < 5) || (2 < 1);
```

- 2 < 5 evaluates to true
- 2 < 1 evaluates to false
- true && false evaluates to true

Logical operators

- Short-circuit evaluations
  - logical operators are evaluated left to right
  - the first condition/expression can provide enough information
    - it will not be necessary to evaluate the second condition
    - they return the first value that stopped the processing

Logical operators

- Short-circuit evaluations
  - …

```
(2 < 5) || (2 < 1);
```

  - the first condition is **true**
    - the whole expression will evaluate to true, no matter what the second condition evaluates to
    - `true || anything` is short-circuit evaluated to *true*

```
(5 < 2) && (2 < 1);
```

  - the first condition is **false**
    - the whole expression will evaluate to false, no matter what the second condition evaluates to
    - `false && anything` *is short-circuit evaluated to false*

Logical operators

- Short-circuit evaluations
  - …
  - Works-around to make sure that all the options are checked
    - false is put first in OR operations, and true in AND operations
  - **Good practice:** Place the options requiring the most processing power last
  - Another thing to watch for is that values can be treated as *truthy* or *falsy* , although they are not Booleans
    - e.g, any non-empty string is evaluated as *truthy*

Logical operators

- `!`
  - logical NOT
  - takes one Boolean value and inverts it
    - `!true` evaluates to false
    - `!false` evaluates to true

```
!(2 < 5) || (2 < 1);
```

- `!`(2 < 5) evaluates to false
- 2 < 1 evaluates to false
- false && false evaluates to false

```
!((2 < 5) || (2 < 1));
```

- (2 < 5) || (2 < 1) evaluates to true
- adding `!` to the result makes it false

Exercise

- Do **one** of the exercises from the pp. 158 -159 (either “Using logical AND” or “Using logical OR & logical NOT”)
  - read the description and write the JS code **by hand** in the text editor / codepen
    - do not copy paste the code
  - Make changes such as these to understand better how the logical operators work
    - change the value of the scores to see what results you are getting
    - add a third variable `score3` and `pass3` to the two both exercises, and evaluate them in the expression, i.e., `(score1 >= pass1) && (score2 >= pass2) && (score3 >= pass3)`
- For those that need extra challenge
  - do both exercises and
  - make other changes to the code, e.g., change the value of `passBoth` to `false` and write a message where you tell the user that s/he fail one of the tests

if / if...else statements

- we evaluate a condition and based on the result we decide which block of code to run
- this can be done with an *if statement*

```
if (score >= 50) { // we use parentheses for the condition
  congratulate(); // calls this function only if the
  condition is evaluated to true
}
```

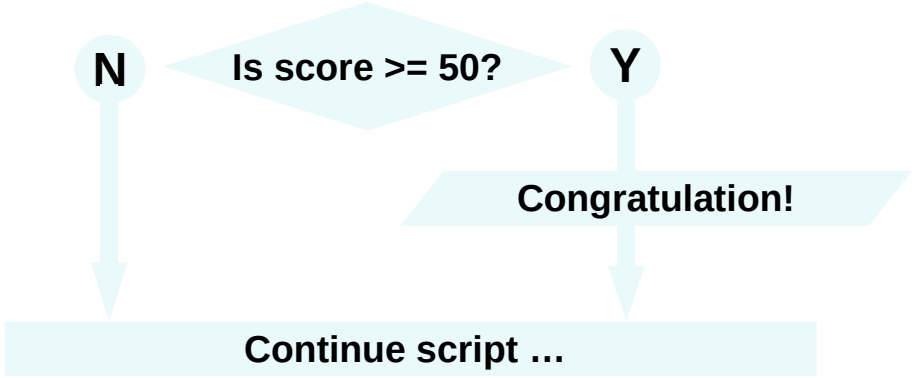
if / if...else statements

- or a *if...else* statements

```
if (condition) {
  code to run if condition is true
} else {
  run some other code if the condition
is false
}
```

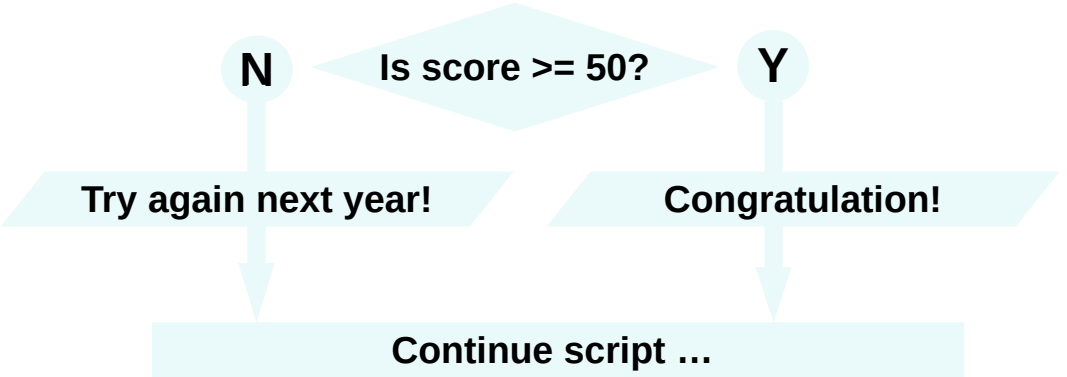
- **pseudocode**
  - refers to code-like syntax
  - used to indicate to humans how some code syntax works,
  - or illustrate the design of an item of code architecture
  - it won't work if you try to run it as code

```
if (score >= 50) {

   congratulate(); // code executed
   if value is true

} else {

   encourage(); // calls this
   function if the condition is
   evaluated to false

}
```



An *if statement* only runs a set of statements if the condition is true.

An *if...else statement* runs one set of statements if the condition is true and another if the condition is false.



Exercise

For this task you are given two variables:

- *machineActive* — contains an indicator of whether the answer machine is switched on or not (true/false)
- *response* — begins uninitialized, but is later used to store a response that will be printed to the output panel.

You need to create an *if...else* structure that checks whether the machine is switched on and puts a message into the response variable if it isn't, telling the user to switch the machine on.

See the code for this exercise here:
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Test_your_skills:_Conditionals#conditionals_2

Edit the code directly on the page of the exercise.

else if statement

- used when we have more than two choices

```
if (choice === 'sunny' && temperature < 30) {

  para.textContent = `It is ${temperature} degrees outside — nice and sunny. Let\'s go out to the beach, or the park, and get an ice cream.`;

} else if (choice === 'sunny' && temperature >= 30) {

  para.textContent = `It is ${temperature} degrees outside — REALLY HOT! If you want to go outside, make sure to put some sunscreen on.`;

} else { // the code inside it will be run if none of the conditions above are true

 para.textContent = '';

}
```

Example from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/conditionals#logical_operators_and_or_and_not

Switch statement

- we have **only one** condition that needs to be evaluated against several choices
- the expression to be evaluated is called a **switch value**
- each *case* indicates a possible value for this expression
- the code runs if the switch value matches that value
- if none of the cases match the code, the *default* is executed
- when the match is found, the *break* statement stops the rest of the switch statement running
- with the *else if* statements, all are checked even if the match has been found

```
switch (expression) {
  case choice1:
    run this code
    break;

  case choice2:
    run this code instead
    break;

  // include as many cases as you like

  default:
    actually, just run this code
}
```

```
let choice = 'snowing';
const para = document.querySelector('p');

switch (choice) {
    case 'sunny':
      para.textContent = 'It is nice and sunny outside today. Wear shorts! Go to the beach, or the park, and get an ice cream.';
      break;
    case 'rainy':
      para.textContent = 'Rain is falling outside; take a rain coat and an umbrella, and don\'t stay out for too long.';
      break;
    case 'snowing':
      para.textContent = 'The snow is coming down — it is freezing! Best to stay in with a cup of hot chocolate, or go build a snowman.';
      break;
    default:
      para.textContent = '';
  }
```

Example from:
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/conditionals#a_switch_example

Conditional (ternary) operator

- the only JavaScript operator that takes three operands:
  - a condition followed by a question mark (?),
  - then an expression to execute if the condition is *truthy*
  - followed by a colon (:),
  - and finally the expression to execute if the condition is *falsy*.

```
( condition ) ? run this code : run this code instead
```

- used as alternative for *if...else* statements

Truthy & falsy values

- every value in JavaScript can be treated as if it were true or false
  - *falsy* values are treated as if they are false
  - *falsy* values can also be treated as the number 0
  - *truthy* values are treated as if they are true
  - *truthy* values can also be treated as the number 1

Truthy & falsy values

- *Falsy* values
  - false, 0, '', null (the absence of any value), undefined (a variable with no value assigned to it) , NaN (not a number)
- *Truthy* values
  - true, 1, any string with content, number calculations, '0' (zero written as a string), 'false' (when written as a string), the presence of an array or object

FALSY	TRUTHY
if (false)	if (true)
if (null)	if ({})
if (undefined)	if ([])
if (0)	if (42)
if (-0)	if ('0')
if (NaN)	if ('false')
if ('')	if (new Date())
	if (-42)
	if (3.14)
	if (-3.14)

Truthy & falsy values

- If the first object is **truthy**, the logical AND operator returns the second operand

```
true && 'dog'  // returns 'dog'

[] && 'dog'     // returns 'dog'
```

- If the first object is **falsy**, it returns that object (short-circuit value)

```
false && 'dog'   // returns false

0 && 'dog'       // returns 0
```

Unary operator

- JavaScript has both **binary** and **unary** operators
- We have looked at binary operators until now
  - a binary operator requires two operands, one before the operator and one after the operator
  - `operand1 operator operand2` (e.g., `3 + 4;`)
- Conditional (ternary) operator → takes three operands, i.e.,

```
condition ? val1 : val2
```

Unary operator

- A **unary** operator uses just one operand in a condition
  - `operator operand`
  - example of use: to check for the presence of one element
    - this is possible because the presence of an object or array is considered *truthy*

```
if (document.getElementById('header')) {
   // Found: do something
} else {
    // Not found: do something else
}
```

- other *unary* operators that we have met before
  - `delete object.property;`
  - used to delete the property of an object

Type coercion

- When JavaScript coverts data types behind the scenes to complete an operation
  - instead of giving us an error, when we use wrong data types in the wrong places
    - e.g., `('1' > 0)` → it converts the string to a number, so that it can evaluate the expression
  - behind the scenes means hidden and is easy to make mistakes
  - instead of relying on JavaScript to do conversions for you, you should rather do it yourself to avoid unexpected results
  - **Good practice:** when checking if two values are equal use the *strict equal operators* (===, !==), that also check for data type equality
- Because the data type of a value can change the JavaScript is said to be using **weak typing**

Type coercion

- **Remember** → JavaScript is *dynamically typed*
  - Other languages require that you specify what type each variable will be (strong typing)
- the data type for a value in JavaScript can change
- data types
  - string (text), number (number), Boolean (true or false), null (empty value), undefined (variable without a value)
  - finding the type of a value with `typeof` operator
    - NOTE: typeof is also a **unary** operator

```
typeof 42;                // expected output: "number"
typeof 'blubber';         // expected output: "string"
typeof true;              // expected output: "boolean"
typeof undeclaredVariable; // expected output: "undefined"
```

Loops

- Loops are useful for doing repetitive tasks
- How do they work
  - they check a condition
  - if the condition returns *true*, a code block will run
  - then the condition is checked again, and if it is still *true*, the code block will run again
    - often the code is slightly different each round, or the same code will be run but with different variables
  - this scenario is repeated until the condition returns *false*

- There are four common types of loops
  - the ***for*** loop is the standard/ most common type of loop
  - ***for ...of*** to loop through a collection, such an array
  - ***while*** does the same thing as the **for** loop
  - ***do...while*** which is very similar to the *for* and *while* loops; the main difference is that it will always run the statements inside the curly braces at least once

for loop

- the condition for a *for* loop is a ***counter*** that tells how many times the loop should run

pseudocode:

```
for (initializer; condition; final-expression) {
    // code to execute during loop
}
```

machine-readable code:

```
/* a loop that runs 10 times and writes to the page the value of the variable i */
for (let i = 0; i < 10; i++) { // we have a condition that counts to ten
    document.write(i);  //  it will print 0123456789 to the page
}
```

for loop

- the ***for*** keyword for followed by parentheses where we have
  - the ***initializer (let i = 0)*** is a variable set to an initial number;
    - this number is incremented with every round to count the number of times the loop has run
    - it is also referred to as the ***counter variable***
  - the ***condition (i < 10)*** defines when the loop should stop looping
    - it is an expression where a comparison operator is used to see if the condition is met or not
  - the ***final-expression (i++ / i--)*** is used to update (increment / decrement) the counter variable
    - it adds/subtracts one to/from the counter every time the loop has run the statements in the curly braces
    - it increments/decrements until the condition is no longer true
    - it uses the ++ / -- arithmetic operators
- in the curly braces we have the block of code that will run each time the loop iterates

Exercise

- Do the exercise: “Active learning: Launch countdown”

  https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Looping_code#active_learning_launch_countdown

  - use a *for loop* (instead of *while* that is given in the suggested solution)
  - use *if...else* and *else...if* to print different texts for 10 and 0
  - if you want to copy over the code to work with it in *codepen*, you will have to add `<div class="output"></div>` in the HTML editor

for...of loop

- for when working with a collection of items and want to do something with every item
- When to choose *for...of* loop
  - if we are iterating through an array
  - we do not need access to the index position of each item
- In these cases you are encouraged to use *for...of* because
  - it is easier to read
  - there is less to go wrong

```
for (const item of array) {
  // code to run
}
```

for...of loop

```
let para = document.querySelector('p');
let contacts = ['Alfhild:2232322', 'Aslaug:3453456', 'Benedikte:7654322',
'Bergliot:9998769', 'Dagrun:9384975', 'Else:9399975', 'Grethe:7656622',
'Gro:2232388' ];
let contactList = '';

for (const contact of contacts) {
   contactList += '<li>' + contact + '</li>';
}

para.innerHTML = '<ul>' + contactList + '</ul>';
```

for...of loop

- `for (const contact of contacts)`
  - given the collection `contacts`, get the first item in the collection
  - assign it to the variable `contact` and then run the code between the curly brackets `{}`
  - get the next item, and repeat until we have reached the end of the collection
- **Aside note**: the *addition assignment operator* (+=) both adds and assigns the value of the right operand to a variable (the left operand)

for loop

```
let para = document.querySelector('p');

let contacts = ['Alfhild:2232322', 'Aslaug:3453456', 'Benedikte:7654322',
'Bergliot:9998769', 'Dagrun:9384975', 'Else:9399975', 'Grethe:7656622',
'Gro:2232388' ];

let contactList = '';


for (let i = 0; i < contacts.length; i++) {

   contactList += '<li>' + contacts[i] + '</li>';

}


para.innerHTML = '<ul>' + contactList + '</ul>';
```

for vs. *for...of*

- we are starting `i` at `0`, and stopping when `i` reaches the `length` of the array
- inside the loop we're using `i` to access each item in the array in turn

- in early versions of JavaScript, *for...of* didn't exist
- a for *loop* was the standard way to iterate through an array

- However, it offers more chances to introduce bugs into your code
  - you might start `i` at `1`, forgetting that the first array index is zero, not 1
  - you might stop at `i <= cats.length`, forgetting that the last array index is at `length - 1`
- It is usually best to use *for...of* if you can

for vs. *for...of*

- we need to use the index of the elements, e.g., to style differently every other element in the array

```
let para = document.querySelector('p');
let contactList = '';
let contacts = ['Alfhild:2232322',
'Aslaug:3453456', 'Benedikte:7654322',
'Bergliot:9998769', 'Dagrun:9384975',
'Else:9399975', 'Grethe:7656622',
'Gro:2232388' ];

for (let i = 0; i < contacts.length; i++) {
  if (i % 2 === 0) {
    contactList += '<li>&bigstar; ' +
    contacts[i] + '</li>';
  } else {
    contactList += '<li>' + contacts[i] +
    '</li>';
  }
}

para.innerHTML = '<ul>' + contactList +
'</ul>';
```

while loop

```
initializer
while (condition) {
  // code to run
  final-expression
}
```

- works in a very similar way to the *for* loop
- the syntax is different
  - that the *initializer* variable is set before the loop
  - the *final-expression* is included inside the loop after the code to run,
  - the initializer and *final-expression* are not being included inside the parentheses as in the case of *for* loop
  - similarly to the *for* loop *condition* is included inside the parentheses, which is preceded by the *while* keyword rather than *for*

```
let para = document.querySelector('p');

let contacts = ['Alfhild:2232322', 'Aslaug:3453456',
'Benedikte:7654322', 'Bergliot:9998769'];

let contactList = '';

let i = 0;


while (i < contacts.length) {
   contactList += '<li>' + contacts[i] + '</li>';
    i++;
}


para.innerHTML = '<ul>' + contactList + '</ul>';
```

for vs. while loop

- the *for* loop has a slightly shorter and more comprehensive form
  - all the statements that are related to the “state” of the loop are grouped together after *for*

do...while

- very similar with the *while* loop
- the code inside a *do...while* loop is always executed **at least once**
  - because the condition comes after the code inside the loop
  - we always run that code once, then check to see if we need to run it again
  - in *while* and *for* loops, the check comes first, so the code might never be executed

```
initializer
do {
  // code to run
  final-expression
} while (condition)
```

do...while

- in practice it is somewhat uncommon to be certain that you want a loop to execute at least once
- the *do...while* loop must always be terminated with a semicolon (;)
  - we do not need this for the *while* loop because the body of the loop is enclosed in curly braces

```
// the code forces you to enter a name

let para = document.querySelector('p');

let yourName;

do {

  yourName = prompt("Who are you?");

} while (!yourName);

para.innerHTML = 'Nice to meet you ' +
yourName  + '!';
```

Jumps

- Jump statements that cause JavaScript interpreter to jump to a new location in the source code
- Exiting loops with ***break***
  - we have seen it used with the *switch statement*
  - we can use it to exit a loop before all the iterations have been completed
- ***continue*** makes the interpreter skip the rest of the body of the loop and jump back to the top of the loop to begin a new iteration
  - instead of breaking out of the loop entirely (as *break* does), it skips to the next iteration of the loop
- Example of use case:
    - we wanted to search through an array of contacts and telephone numbers
    - we search with the name of the person
    - we return just the number related to that person we wanted to find (***break***)
    - we want to return that person and all the remaining in the list (***continue***)

```
let para = document.querySelector('p');
let contacts = ['Alfhild:2232322', 'Aslaug:3453456', 'Benedikte:7654322',
'Bergliot:9998769', 'Dagrun:9384975', 'Else:9399975', 'Grethe:7656622', 'Gro:2232388' ];

let arrayLength = contacts.length;
let searchName = 'Else';

for (i = 0; i < arrayLength; i++) {
  //para.textContent = contacts[i].split(':');
  let splitContact = contacts[i].split(':');

if (splitContact[0] === searchName) {
      para.innerHTML = splitContact[0] + '\'s number is ' + splitContact[1] + '.';
      continue;  // change with break; remove both break and continue
    }

   para.innerHTML += '<br \>' + splitContact[0] + '\'s number is ' + splitContact[1] + '.';
}
```

Performance issues

- **infinite loops**
  - if your condition never returns *false*
  - you must make sure that the *initializer* is incremented /decremented, so the condition eventually becomes *false*
    - especially easy to forget in the case of the *while* or *do..while* loops
  - if not, the loop will go on forever, and either the browser will force it to stop, or it will crash
- define any variable that does not need to change within the loop outside of it
  - variables declared inside are recalculated every time the loop is ran
  - unnecessary use of resources