Basic concepts. Git

What is JS?

JavaScript (JS) is a dynamic programming language primarily used for web development, providing:

- Versatility: Supports various programming paradigms and platforms.
- Cross-Platform Compatibility: Works across all modern web browsers.
- Vast Ecosystem: Offers a wide range of libraries and frameworks for efficient development.

JS powers web, mobile, and server-side applications, making it indispensable in modern software development.

The role of JS in web development

JavaScript is an important part of any modern Web application as it provides such opportunities:

1. **Responsiveness** for making the application “alive” even **without** page reloading.
2. **Cross-Platform development** for running one and the same code on various devices.
3. **Client-side data processing** for validating data in place so avoiding redundant HTTP-requests
4. **Relative simplicity** for speeding up the app development without quality losses (especially in case of framework usage)
5. **Strong standardization** for having the majority of frontend apps easier to support.
6. ***Full-stack development** is possible with using JS only, but won’t be covered in this course.

Setting up development environment

Another advantage of JavaScript is that in the basic case it actually doesn't need any special environment set up. For writing the first program it would be enough to:

1. Create a *.js file and fill in some code.
2. Create a basic *.html file and add <script> tag which would be referring to JS file
3. Open HTML file with browser

And that's it! You will see the outcome of your work immediately. In fact, it can be done even without using an IDE, but it is strongly recommended to start using one even for the simplest scripts to get familiar with the UI and basic features.

The most popular IDEs for frontend development and free Visual Studio Code and paid WebStrorm.

Running JS files in terminal

Studying JavaScript is not only about building the frontend applications with a UI. There are a lot of features and tricky underwater rocks to be tried out even without HTML interacting.

Here Node.js can be helpful. It is possible to run a JS file here and now without switching to browser which is relevant for a program we are writing just to take a look at console output.

Installing npm and node

This installation process is relatively easy to be performed. Just follow these steps:

1. run **npm install -g npm** to install the NPM (at this point we won't be using it so much but in general having a package manager is a crucial thing for a developer).
2. Follow the NodeVersionManager guide.
3. Run **nvm install latest** for getting the newest Node.js version
4. Run **nvm use latest** for enabling downloaded version

Besides of running simple programs in educational purposes node.js is strongly used for developing enterprise applications so it worth to understand how to get it to your machine.

Hello World!

TODOs:

1. Create a new JS file (by using any IDE or whatever)
2. Implement a classical Hello World program
3. Run the file using node

Variables

A variable is a named store for data. For most programming languages, this concept is **key** to writing programs. With their help, you can store and change any data during program execution.

Before the advent of the ECMAScript5 standard (hereinafter referred to as ES5), variables in JavaScript could be created **without** any keywords. We will not consider the features of such creation of variables, because this approach is completely outdated and in strict mode is already considered an error. ES5 introduced a new keyword, **var**, which is also obsolete, but can still be found in some “ancient” applications. It is also useful to understand var because the code of many modern frameworks is brought exactly to the ES5 standard.

Starting with the ECMAScript6 standard (hereinafter ES6) to this day, the **let** and **const** keywords are used to create variables.

Features of the var keyword

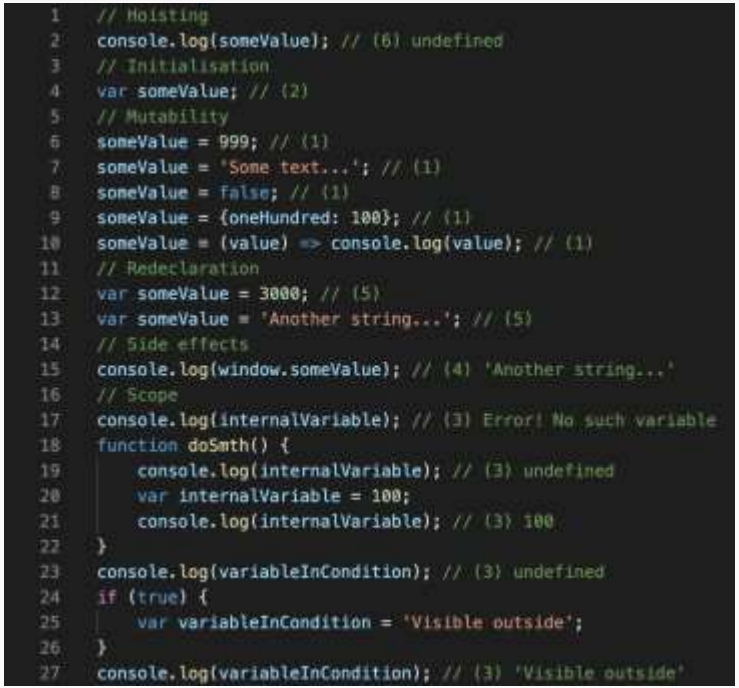
1. **Variability:** The value of a variable can change;

2. **Initialization:** When created, the variable may not be initialized;

3. **Scope:** Functional;

4. **Multiple creation:** Possible;

5. **Hoisting:** Yes, without restrictions.



```
// Hoisting
console.log(someValue); // (6) undefined
// Initialisation
var someValue; // (2)
// Mutability
someValue = 999; // (1)
someValue = 'Some text...'; // (1)
someValue = false; // (1)
someValue = {oneHundred: 100}; // (1)
someValue = (value) => console.log(value); // (1)
// Redeclaration
var someValue = 3000; // (5)
var someValue = 'Another string...'; // (5)
// Side effects
console.log(window.someValue); // (4) 'Another string...'
// Scope
console.log(internalVariable); // (3) Error! No such variable
function doSmth() {
    console.log(internalVariable); // (3) undefined
    var internalVariable = 100;
    console.log(internalVariable); // (3) 100
}
console.log(variableInCondition); // (3) undefined
if (true) {
    var variableInCondition = 'Visible outside';
}
console.log(variableInCondition); // (3) 'Visible outside'
```

Features of the let keyword

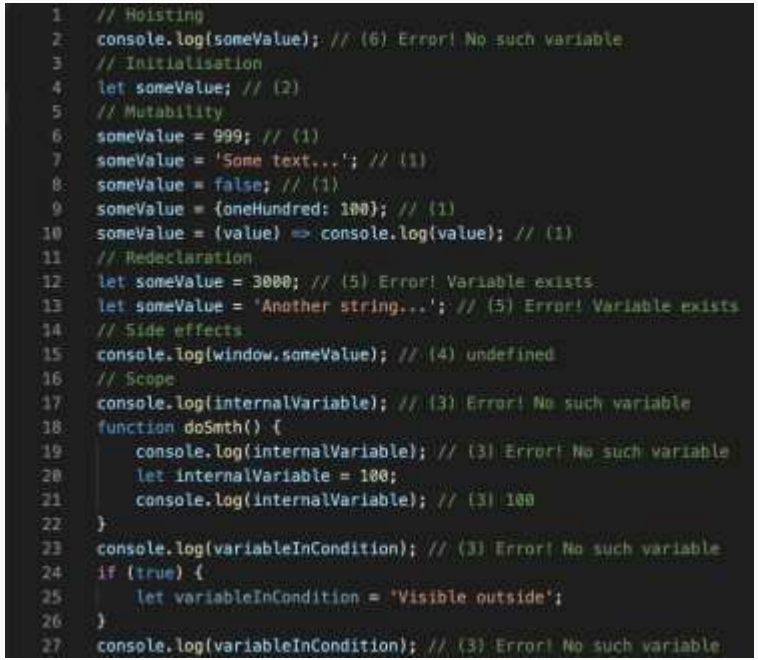
1. **Variability:** The value of a variable can change;

2. **Initialization:** When created, the variable may not be initialized;

3. **Scope:** Block;

4. **Multiple creation:** Impossible;

5. **Hoisting:** Yes, but with a dead zone.



```
// Hoisting
console.log(someValue); // (6) Error! No such variable
// Initialisation
let someValue; // (2)
// Mutability
someValue = 999; // (1)
someValue = 'Some text...'; // (1)
someValue = false; // (1)
someValue = {oneHundred: 100}; // (1)
someValue = (value) => console.log(value); // (1)
// Redeclaration
let someValue = 3000; // (5) Error! Variable exists
let someValue = 'Another string...'; // (5) Error! Variable exists
// Side effects
console.log(window.someValue); // (4) undefined
// Scope
console.log(internalVariable); // (3) Error! No such variable
function doSmth() {
    console.log(internalVariable); // (3) Error! No such variable
    let internalVariable = 100;
    console.log(internalVariable); // (3) 100
}
console.log(variableInCondition); // (3) Error! No such variable
if (true) {
    let variableInCondition = 'Visible outside';
}
console.log(variableInCondition); // (3) Error! No such variable
```

Features of the const keyword

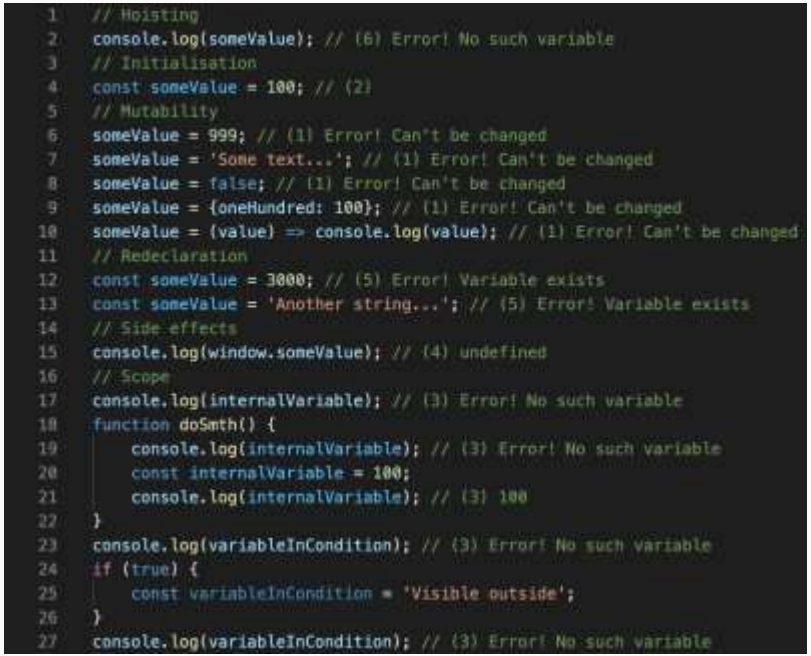
1. Variability: The value of a variable can't change;

2. Initialization: When created, the variable must be initialized;

3. Scope: Block;

4. Multiple creation: Impossible;

5. Hoisting: Yes, but with a dead zone.



```
// Hoisting
console.log(someValue); // (6) Error! No such variable
// Initialisation
const someValue = 100; // (2)
// Mutability
someValue = 999; // (1) Error! Can't be changed
someValue = 'Some text...'; // (1) Error! Can't be changed
someValue = false; // (1) Error! Can't be changed
someValue = {oneHundred: 100}; // (1) Error! Can't be changed
someValue = (value) => console.log(value); // (1) Error! Can't be changed
// Redeclaration
const someValue = 3000; // (5) Error! Variable exists
const someValue = 'Another string...'; // (5) Error! Variable exists
// Side effects
console.log(window.someValue); // (4) undefined
// Scope
console.log(internalVariable); // (3) Error! No such variable
function doSmth() {
    console.log(internalVariable); // (3) Error! No such variable
    const internalVariable = 100;
    console.log(internalVariable); // (3) 100
}
console.log(variableInCondition); // (3) Error! No such variable
if (true) {
    const variableInCondition = 'Visible outside';
}
console.log(variableInCondition); // (3) Error! No such variable
```

Data types

Whenever you specify a new value in a program, you choose the most appropriate representation for it. The choice may vary depending on what exactly you plan to do with it. Different representations for values are called **types** in programming terminology.

```
const name : string  = 'Max'
const age : number  = 24;
const isRegistered : boolean  = true;
const bigInteger : bigint  = 10n;
const symbol : symbol  = Symbol( description: 'id');
const empty : undefined  = undefined;
const almostEmpty : null  = null;
const user : {balance: number}  = { balance: 120 };
```

Number

In JavaScript, there is no distinction between integers and fractions at the data type level. The **number** data type includes both integer values (integer, int) and floating-point values (float, double).

```
const positiveNumber : number  = 100;
const negativeNumber : number  = -123.45;

const PI : number  = Math.PI;
const positiveInfinity : number  = Infinity;
const negativeInfinity : number  = - Infinity;
```

String

The JavaScript string, to some extent, also includes two data types from other C-like languages. The ***char*** type (character) and the ***string*** type itself. **Strings** in JavaScript are essentially just a sequence of characters. Strings can be denoted using either 'single quotes' or “double quotes”. There is no difference between these two options. However, there is another designation option: `backquotes`. They can include substitutions or in other words {evaluated expressions}`, and also preserve line breaks.

```
const helloWorld : string  = 'Hello World!'
const computedString : string  = `5 + 13 = ${ 5 + 13 }`;
const characterC : string  = "c";
const emptyString : string  = "";
```

Boolean

While numbers and strings can have countless different values, the **boolean** type has only two valid values. These values are **true** and **false**.

A Boolean type is, in most cases, the result of testing whether something is true.

```
const isJSComplicated : boolean  = false;
const twoLessThanFive : boolean  = 2 < 5;
```

Big integer values

Bigint is a data type that can hold integers of any size without loss of precision. Bigint values can be obtained by appending the character **n** to the end of a numeric literal, or by calling the built-in **BigInt()** function.

This data type can be very useful in situations where the numbers we have to work with have a chance of exceeding the maximum value of the regular number type. This value is contained in a constant inside JavaScript itself: **Number.MAX_SAFE_INTEGER**.

```
const zero : bigint  = 0n;
const tiny : bigint  = 10n;
const huge : bigint  = 99999999999999999999999999n;
```

Symbol

At first glance, it may seem that the **symbol** type is an analogue of the ***char*** type from other programming languages. Actually this is **not** true.

The **symbol** type in JavaScript is a *unique identifier*. This type was introduced to solve some problems of this programming language. However, you are free to use it for your own purposes too.

```
const justASymbol : symbol  = Symbol();
const stringSymbol : symbol  = Symbol( description: 'id')
const numberSymbol : symbol  = Symbol( description: 42)
```

undefined / null

undefined - absence of presence

null - presence of absence

```
let emptyVariable; // undefined
let nonInitializedVariable : null  = null; // null

const obj : {}  = {};
obj.value; // undefined

1 usage
function logHello() : void  {
        console.log('Hello');
}

console.log(logHello()); // undefined
```

Object

Unlike all previous data types, an object has a **reference** type. This means that all objects in JavaScript are handled **by reference**. **No** variable in JavaScript contains objects. They all just **refer** to them in memory.

Data in objects is stored as a “key: value” pair, where the key is a **string** (or **symbol**), the value can be **anything** (including another object).

```
const person : Person  = new Person( name: 'Alex');
const date : Date  = new Date();
const numbers : number[]  = [1, 2, 3, 4, 41, 42, 42];
const uniqueNumbers : Set<number>  = new Set(numbers);

1 usage
function logHello() : void  {
        console.log('Hello')
}
```

typeof

```
const oneHundred : number  = 100;
const hello : string  = 'Hello';
const mySymbol : symbol  = Symbol( description: 'id');
const isTruthy : boolean  = true;
const hugeNumber : bigint  = 42000n;
const user : {age: number}  = {  age: 23 };

1 usage
function logHello() : void  {
        console.log('Hello')
}
```

```
typeof oneHundred; // number
typeof(hello); // string
typeof mySymbol; // symbol
typeof isTruthy; // boolean
typeof undefined; // undefined
typeof hugeNumber; // bigint
typeof user; // object
typeof null; // object
typeof logHello; // function
```

Dynamic Typing

JavaScript is a **dynamically typed** programming language. This means that the same variable can contain values of **different types at different times**. JavaScript does not impose any restrictions on overwriting the value of a variable (unless the variable cannot be changed).

For comparison, in languages with ***static typing***, the type is ***rigidly bound*** to a variable and cannot be changed either during the operation of the application, or even during the compilation of the application.

```
let variable : string  = 'Hello!';
variable = 42;
variable = true;
variable = { message: 'Hello!' };
variable = null;
```

Type casting

In the process of developing applications, we are constantly faced with the need to convert one type to another. The most common situations are:

- Converting a number to a string;
- Converting a string to a number;
- Casting a value to a boolean type.

```
console.log(1 + '1'); // 11
```

Casting a number to a string

Explicit cast:

1. By calling the built-in function **String()**;

2. By calling the **.toString()** method on the number directly or on a variable containing the number.

```
String( value: 42);
4200..toString();
12..toString( radix: 2);

console.log(35 + '');
```

Implicit cast:

1. By “adding” an empty string to a number.

Casting a string to a number

Explicit cast:

1. By calling the built-in function **Number();**

2. By calling the built-in functions **parseInt()** and **parseFloat()**.

Implicit cast:

1. Using the unary operator '+';

2. During comparison;

3. When using arithmetic operators;

4. When using bitwise operators

```
Number( value: '100');
Number( value: 'a12'); // NaN
parseInt( string: '122.5'); // 122
parseFloat( string: '32.25');
parseFloat( string: '32.25abc'); // 32.25
+'12';
5 - '4';
2 > '3';
```

Boolean casting

Explicit cast:

1. By calling the built-in function **Boolean()**.

Implicit cast:

1. Using the double use of the unary operator '!' (not);

2. In the conditional statement if;

3. When using logical operators.

```
console.log(Boolean(153.22)); // true
console.log(Boolean(null)); // false
console.log(Boolean([])); // true
console.log(Boolean('Some text...')); // true
console.log(Boolean('')); // false
```

```
console.log(!!0); // false
console.log(!!{}); // true
if ('Some text...') {
    console.log('String is not empty!'); // will be logged
}
if (undefined) {
    console.log('Never...'); // won't be logged
}
console.log(2 || null); // 2
console.log(0 && 'foo'); // 0
```

Comparison

In the process of developing application logic, we very often have to check some values for equality (or inequality) to something. JavaScript has the following options:

• Comparison with type cast: == (equal), != (not equal);

• Comparison without type cast: === (equal), !== (not equal).

The key difference in these two approaches lies in the names themselves. When comparing with type casting (==), the operands are first cast to the same data type (number) and only then are compared. In the case of strict comparison (===), this does not happen, and if we try to compare values of different types, we always get **false**.

```
1  console.log("" == 0); // true
2  console.log('6' == 6); // true
3  console.log(12 === '12'); // false
4  console.log(false == ''); // true
```

Arithmetic operators

Arithmetic operators are actually working just as in math - they are simple adding (**+**), subtraction (**-**), multiplication (*****) and division(**/**) operations. The significant points is the type casting features (partially mentioned before):

1. Binary **+** when facing a string will end up with a concatenation even if the string contains a number.
2. Unary **+** is capable of string to number casting.
3. All the others will do the string to number casting implicitly and end up with NaN in case of failure.

Logical operators

Once again are just representations of AND (**&&**) and OR (||) boolean math operations. In significant difference from the majority of programming languages, in JS these operators return **not** the boolean result the **actual value** have been compared. The logic is quite simple: **&&** returns **first false** value or the **last true** one; || returns **first true** value or the **last false** one.

```
const a : null | string  = "apple" && "banana" && null && "cherry"; // null
const b : string  = "apple" && "banana" && 1 && "cherry"; // cherry
const c : string  = "" || false || "hello"; // hello
const d : number  = "" || false || 0; // 0
```

If statement

Conditional statement allows programer to add more variability to a program. The logic is equal to the meaning of keywords used.

The important thing is that in JS these statements are capable of doing implicit type casting.

```
if (condition) {
    // do something
} else if (anotherCondition) {
    // or something else
} else {
    // or event this
}
```

Switch statement

A long sequence of if-else statement is considered to be a bad practise. So here comes another construction - switch-case.

It is capable of comparing a variable value with provided options and execute a corresponding logic.

It’s important to use **break;** statement where needed, otherwise all subsequent blocks would be executed.

```
const fruit = getFruit();

switch (fruit) {
    case "apple":
        console.log("It's an apple");
    case "peach":
        console.log("It's an apple");
        break;
    default:
        console.log("It's an unknown fruit")
}
```

Switch statement

```
const number : number  = 42;

switch (true) {
    case number === 43:
        console.log("1");
        break;
    case 4 < 6:
        console.log("2");
        break;
    case someSpecialCheck(number):
        console.log("2");
        break;
}
```

What is Git

Git is a version control system that makes it easy to organize the workflow for a development team. This is achieved by the fact that with the help of Git you can: save different project states and easily move between them; make new branches from the current state of the project in order to develop new features “in isolation”; “pull up” new changes from other branches and quite easily resolve conflicts that arise. With all this abundance of useful features, Git itself remains a fairly small and fast utility. The high speed is due to the fact that Git is installed and runs locally on your PC.

Git stores the project's change history as a set of “frozen” states, each of which you can return to later. At each stage of saving a new project state in Git, the system remembers what each file looks like at that moment and stores a link to this state. The end result is a long story of changes, each of which imposes some new piece of code on your project.

Basic features of working with a local repository

- **git init** — initializing a new repository;
- **git add** — adding changes step by step;
- **git commit** — registration of a certain list of changes with a description and assignment of a unique ID to these changes;
- **git status** — tracking the current state of the repository;
- **git config** — writing and reading Git configuration;
- **git branch** — display current branch, create new branches, delete branches;
- **git checkout** — switching to other branches;
- **git merge** — merging branches, i.e. merging changes from two branches into one.

Git init

This command “turns” a directory on your PC into an empty Git repository. This is the first step towards creating a repository. After executing this command, you can begin adding and registering (creating commits) changes.

```
ramanaliaksanau@Ramans-MacBook-Pro ~ % cd Documents/Work/Git-repo-example
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git init
Initialized empty Git repository in /Users/ramanaliaksanau/Documents/Work/Git-repo-example/.git/
```

Git add

Using this command, you can add modified (added) files to some kind of intermediate Git data source. This is necessary for further registration (creating commits based on them) of these changes. There are several ways to add files: adding each file individually, adding a directory of files, or adding all changes at once.

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git add ./index.html
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git add css
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git add .
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example %
```

Git commit

This command allows you to write/register changes made to local repository files. In this case, a new set of changes (hereinafter referred to as a commit) receives its own unique identifier by which it can be found.

It is good practice to include a comprehensive description in the new commit message. This description should provide an explanation for the changes made. In the future, good commit messages greatly simplify searching through the project's change history.

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git commit -m "Create initial project structure"
[master (root-commit) 8254583] Create initial project structure
 4 files changed, 23 insertions(+)
 create mode 100644 css/normalize.css
 create mode 100644 css/style.css
 create mode 100644 index.html
 create mode 100644 js/main.js
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example %
```

Git status

This command tells you the current state of the repository.

Calling the git status command will display the branch you are currently on. In addition, if you have files at the adding stage, i.e. not yet committed, information about this will also be displayed. If there are no changes ready for registration, this command will display the message “nothing to commit, working tree clean”.

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   css/normalize.css
        new file:   css/style.css
        new file:   index.html
        new file:   js/main.js
```

Git config

Git allows us to set many different settings, and this is done using the git config command. The most important configuration items are user.name and user.email. These values represent information about the author of commits on the local device.

The git config command has a **--global** flag that allows us to set the configuration value for all repositories at once. Without this flag, settings will only affect the current repository.

There are a large number of settings that can be configured using the git config command.

```
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git config --global user.name "Roman Alexanov"
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git config --global user.email alexanov.roman@gmail.com
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git config user.name "Raman Aliaksanau"
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git config user.name
Raman Aliaksanau
```

Git branch

This command is needed to determine the current branch, as well as to create and delete branches.

```
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch
* master
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch feature-1
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch
  feature-1
* master
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch -a
  feature-1
* master
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch -d feature-1
Deleted branch feature-1 (was ceae1e9).
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch -a
* master
```

Git checkout

This command is needed to change the working (current) branch. Use git checkout to switch to a new branch.

You can also create new branches using this command. In this case, after calling **git checkout -b new-branch-name**, a new branch will be created and immediately become the current one.

```
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch
 * master
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git checkout test
 error: pathspec 'test' did not match any file(s) known to git
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git checkout -b test
 Switched to a new branch 'test'
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch
   master
 * test
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git checkout master
 Switched to branch 'master'
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch
 * master
   test
```

Git merge

This command merges changes from two branches. As a result of calling git merge, a **new commit** is created containing the changes from the current and target branches.

A common situation: the latest changes from the **dev** (or develop) branch are merged with the changes on your **feature** branch. This is done to access new features from the dev branch or to resolve conflicts.

Basic features of working with a remote repository

- **git remote** — creating a connection between local and remote repositories;
- **git clone** — creating a local copy of an existing remote repository;
- **git fetch** — getting data about changes in a branch;
- **git pull** — getting the latest version of the repository;
- **git push** — sending local changes as a list of commits to a remote repository.

Git remote

This command is needed to connect a local repository to a remote one. In this case, the remote repository can have some name so that it does not have to remember or store its full URL.

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git remote add test https://github.com/      /repo-example.git
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git remote -v
origin  https://github.com/      /repo-example.git (fetch)
origin  https://github.com/      /repo-example.git (push)
test    https://github.com/      /repo-example.git (fetch)
test    https://github.com/      /repo-example.git (push)
```

Git clone

This command is needed to create a local copy of an existing remote repository. After calling the git clone command, git will create a new directory with a name equal to the name of the remote repository and all its contents (files, branches, change history).

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git clone https://github.com/[redacted]/andersen-lesson-5.git
Cloning into 'andersen-lesson-5'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example %
```

Git fetch

This command is needed to obtain the latest data from a remote repository. These could be new branches, commits, etc. However, it is important to understand that this command simply receives this data, but does not do anything with it.

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 733 bytes | 366.00 KiB/s, done.
From https://github.com/R0Mkka/repo-example
   9c12ac4..97e4f07  master     -> origin/master
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example %
```

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git merge
Updating 9c12ac4..97e4f07
Fast-forward
 file-from-git.js | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 file-from-git.js
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example %
```

Git pull

This command essentially combines the git fetch and git merge calls. Those. first, the new state of the branch is obtained in the remote repository, and then the new data is automatically added to the local branch. Moreover, if your branch is already up to date, then when you call git pull, the message “Already up to date.” will be displayed.

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 663 bytes | 221.00 KiB/s, done.
From https://github.com/[redacted]/repo-example
   0417e30..12681c4  pull-example -> origin/pull-example
Updating 0417e30..12681c4
Fast-forward
 aga.js | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 aga.js
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git pull
Already up to date.
```

Git push

Writes local commits to a remote repository. Can take two parameters: the name (or URL) of the remote repository and the name of the branch from which we want to commit changes. Also, in some situations, this command can be called without parameters.

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git add .
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git commit -m "Add huge.js file"
[push-example 48abe93] Add huge.js file
 1 file changed, 2700 insertions(+)
 create mode 100644 huge.js
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git push test push-example
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 479 bytes | 479.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/[redacted]/repo-example.git
   12681c4..48abe93  push-example -> push-example
```

Additional features

- **git stash** — saving the current state of the repository and clearing the directory of all changes;
- **git cherry-pick** — inserting certain commit(s) into your branch;
- **git rebase** — moving multiple commits to a new base commit.

Q/A

H/A

Try to confuse others with an unusual “console log” using non-obvious JS tricks =)

Important: the complexity of an expression should not lie in its length.

```
console.log(`ba${2 / undefined}a`)
```

H/A

Write a function that transforms a number into a binary form.

Optionally:
*output the number by any method
*receive a number from user by any input method.