The **`parseInt()`** function parses a string argument and returns an integer of the specified radix(the base in mathematical numeral systems).

Syntax

```
parseInt(string [, radix])
```

Parameters

`string`
The value to parse. If this argument is not a string, then it is converted to one using the `ToString` abstract operation. Leading whitespace in this argument is ignored.

`radix` *Optional*
An integer between `2` and `36` that represents the *radix* (the base in mathematical numeral systems) of the *`string`*. Be careful—this does ***not*** default to `10`!

> The description below explains in more detail what happens when *`radix`* is not provided.

Return value

An integer parsed from the given *`string`*.

Or `NaN` when

- the *`radix`* is smaller than `2` or bigger than `36`, or
- the first non-whitespace character cannot be converted to a number.

Description

The `parseInt` function converts its first argument to a string, parses that string, then returns an integer or `NaN`.

If not `NaN`, the return value will be the integer that is the first argument taken as a number in the specified *`radix`*. (For example, a *`radix`* of `10` converts from a decimal number, `8` converts from octal, `16` from hexadecimal, and so on.)

For radices above `10`, letters of the English alphabet indicate numerals greater than `9`. For example, for hexadecimal numbers (base `16`), `A` through `F` are used.

If `parseInt` encounters a character that is not a numeral in the specified `radix`, it ignores it and all succeeding characters and returns the integer value parsed up to that point. `parseInt` truncates numbers to integer values. Leading and trailing spaces are allowed.

Because some numbers use the `e` character in their string representation (e.g. **`6.022e23`** for 6.022×10^{23}), using `parseInt` to truncate numbers will produce unexpected results when used on very large or very small numbers. `parseInt` should *not* be used as a substitute for `Math.floor()`.

`parseInt` understands exactly two signs: `+` for positive, and `-` for negative (since ECMAScript 1). It is done as an initial step in the parsing after whitespace is removed. If no signs are found, the algorithm moves to the following step; otherwise, it removes the sign and runs the number-parsing on the rest of the string.

If *`radix`* is `undefined`, `0`, or unspecified, JavaScript assumes the following:

1. If the input `string` begins with `"0x"` or `"0X"` (a zero, followed by lowercase or uppercase X), *`radix`* is assumed to be `16` and the rest of the string is parsed as a hexidecimal number.
2. If the input `string` begins with `"0"` (a zero), *`radix`* is assumed to be `8` (octal) or `10` (decimal). Exactly which radix is chosen is implementation-dependent. ECMAScript 5 clarifies that `10` (decimal) *should* be used, but not all browsers support this yet. For this reason, **always specify a *`radix`* when using `parseInt`.**
3. If the input `string` begins with any other value, the radix is `10` (decimal).

If the first character cannot be converted to a number, `parseInt` returns `NaN`.

For arithmetic purposes, the `NaN` value is not a number in any radix. You can call the `isNaN` function to determine if the result of `parseInt` is `NaN`. If `NaN` is passed on to arithmetic operations, the operation result will also be `NaN`.

To convert a number to its string literal in a particular radix, use *`thatNumber`*`.toString(`*`radix`*`)`.

> **`BigInt` Warning:** `parseInt` converts a `BigInt` to a `Number` and loses precision in the process. This is because trailing non-numeric values, including `"n"`, are discarded.

Octal interpretations with no radix

Although discouraged by ECMAScript 3 and forbidden by ECMAScript 5, many implementations interpret a numeric string beginning with a leading `0` as octal. The following may have an octal result, or it may have a decimal result. **Always specify a *`radix`* to avoid this unreliable behavior.**

```
parseInt('0e0')  // 0
parseInt('08')   // 0, because '8' is not an octal digit.
```

The ECMAScript 5 specification of the function `parseInt` no longer allows implementations to treat Strings beginning with a `0` character as octal values.

ECMAScript 5 states:

> The `parseInt` function produces an integer value dictated by interpretation of the contents of the string argument according to the specified radix. Leading whitespace in string is ignored. If radix is `undefined` or `0`, it is assumed to be `10` except when the number begins with the character pairs `0x` or `0X`, in which case a radix of `16` is assumed.

This differs from ECMAScript 3, which merely *discouraged* (but allowed) octal interpretation.

Many implementations have not adopted this behavior as of 2013. And, because older browsers must be supported, **always specify a radix**.

A stricter parse function

It is sometimes useful to have a stricter way to parse integers.

Regular expressions can help:

```
function filterInt(value) {
  if (/^[-+]?(\d+|Infinity)$/.test(value)) {
    return Number(value)
  } else {
    return NaN
  }
}
```

```
console.log(filterInt('421'))               // 421
console.log(filterInt('-421'))              // -421
console.log(filterInt('+421'))              // 421
console.log(filterInt('Infinity'))          // Infinity
console.log(filterInt('421e+0'))            // NaN
console.log(filterInt('421hop'))            // NaN
console.log(filterInt('hop1.61803398875'))  // NaN
console.log(filterInt('1.61803398875'))     // NaN
```

Examples

Using parseInt

The following examples all return `15`:

```
parseInt('0xF', 16)
parseInt('F', 16)
parseInt('17', 8)
parseInt(021, 8)
parseInt('015', 10)    // but `parseInt(015, 8)` will return 13
parseInt(15.99, 10)
parseInt('15,123', 10)
parseInt('FXX123', 16)
parseInt('1111', 2)
parseInt('15 * 3', 10)
parseInt('15e2', 10)
parseInt('15px', 10)
parseInt('12', 13)
```

The following examples all return `NaN`:

```
parseInt('Hello', 8)  // Not a number at all
parseInt('546', 2)    // Digits other than 0 or 1 are invalid for binar
```

The following examples all return `-15`:

```
parseInt('-F', 16)
parseInt('-0F', 16)
parseInt('-0XF', 16)
parseInt(-15.1, 10)
parseInt('-17', 8)
parseInt('-15', 10)
parseInt('-1111', 2)
parseInt('-15e1', 10)
parseInt('-12', 13)
```

The following examples all return `4`.

```
parseInt(4.7, 10)
parseInt(4.7 * 1e22, 10)        // Very large number becomes 4
parseInt(0.00000000000434, 10)  // Very small number becomes 4
```

If the number is greater than 1e+21 (including) or less than 1e-7 (including), it will return `1.` (when using radix 10).

```
parseInt(0.0000001,10);
parseInt(0.000000123,10);
parseInt(1e-7,10);
parseInt(100000000000000000000,10);
parseInt(123000000000000000000000,10);
parseInt(1e+21,10);
```

The following example returns `224`:

```
parseInt('0e0', 16)
```

`BigInt` values lose precision:

```
parseInt('900719925474099267n')
// 900719925474099300
```

`parseInt` doesn't work with numeric separators:

```
parseInt('123_456')
// 123
```