Fields

Class fields are variables that hold information. Fields can be attached to 2 entities:

1. Fields on the class instance
2. Fields on the class itself

The fields also have 2 levels of accessibility:

1. Public: the field is accessible anywhere
2. Private: the field is accessible only within the body of the class

Public instance fields

Let's look again at the previous code snippet:

```
class User {
  constructor(name) {
    this.name = name;
  }
}
const user = new User("Jon Snow");
console.log(user.name)
```

```
// Output
Jon Snow
```

`name` is a ***public field*** because you can access it outside of the `User` class body.

When the fields are created implicitly inside the constructor, like in the previous scenario, it could be difficult to grasp the fields list. You have to decipher them from the constructor's code.

A better approach is to explicitly declare the class fields. No matter what constructor does, the instance always has the same set of fields.

The class fields proposal lets you define the fields inside the body of the class. Plus, you can indicate the initial value right away:

```
class User {
  name;
  constructor(name) {
    this.name = name;
  }
}

const user = new User("Jon Snow");
console.log(user.name);
```

```
// Output
Jon Snow
```

name; inside the body of the class declares a public field **name**.

There's no restriction on access or update of the public fields. You can read and assign values to public fields inside the constructor, methods, and outside of the class.

Private instance fields

Encapsulation is an important concept that lets you hide the internal details of a class.A good way to hide internal data of an object is to use the private fields. These are the fields that can be read and change only within the class they belong to. The outside world of the class cannot change private fields directly.

> The private fields are accessible only within the body of the class.

Prefix the field name with the special symbol # to make it private, e.g. #myField. The prefix # must be kept every time you work with the field: declare it, read it, or modify it.

Let's make sure that the field #name can be set once at the instance initialization:

```
class User {
  #name;
  constructor(name) {
    this.#name = name;
  }

  getName() {
    return this.#name;
  }
}

const user = new User('Jon Snow');
console.log(user.getName());
console.log(user.#name);
```

```
// Output
Jon Snow
SyntaxError is thrown
```

name is a private field. You can access and modify #name within the body of the User. The method getName() can access the private field #name.

But if you try to access the private field #name outside of User class body, a syntax error is thrown: **SyntaxError:** `Private field '#name' must be declared in an enclosing class`.

Public static fields

You can also define fields on the class itself: ***the static fields***. These are helpful to define class constants or store information specific to the class.

To create static fields in a JavaScript class, use the special keyword `static` followed by the field name: `static myStaticField`.

Let's add a new field `type` that indicates the user type: admin or regular. The static fields `TYPE_ADMIN` and `TYPE_REGULAR` are handy constants to differentiate the user types:

```
class User {
  static TYPE_ADMIN = 'admin';
  static TYPE_REGULAR = 'regular';
  name;
  type;

  constructor(name, type) {
    this.name = name;
    this.type = type;
  }
}

const admin = new User('Site Admin', User.TYPE_ADMIN);
console.log(admin.type);
console.log(User.TYPE_ADMIN);
```

```
// Output
admin
admin
```

`static TYPE_ADMIN` and `static TYPE_REGULAR` define static variables inside the `User` class. To access the static fields, you have to use the class followed by the field name: `User.TYPE_ADMIN` and `User.TYPE_REGULAR`.

Private static fields

Sometimes even the static fields are an implementation detail that you'd like to hide. In this regard, you can make static fields private.

To make the static field private, prefix the field name with `#` special symbol: `static #myPrivateStaticField`.

Let's say you'd like to limit the number of instances of the `User` class. To hide the details about instances limits, you can create the private static fields:

```
class User {
  static #MAX_INSTANCES = 2;
  static #instances = 0;
  name;

  constructor(name) {
    User.#instances++;
    if (User.#instances > User.#MAX_INSTANCES) {
      throw new Error('Unable to create User instance');
    }
    this.name = name;
  }
}

const user1 = new User("Jon Snow");
console.log(user1.name)
const user2 = new User("Arya Stark");
console.log(user2.name)
const user3 = new User("Sansa Stark"); // throws Error due the pre-defined condition in constructor
```

```
// Output
Jon Snow
Arya Stark
Error: Unable to create User instance
```

The static field `User.#MAX_INSTANCES` sets the maximum number of allowed instances, while `User.#instances` static field counts the actual number of instances.

These private static fields are accessible only within the `User` class. Nothing from the external world can interfere with the limits mechanism: that's the benefit of encapsulation.