Your generator will interact a lot with the end user. By default Yeoman runs on a terminal, but it also supports custom user interfaces that different tools can provide. For example, nothing prevents a Yeoman generator from being run inside of a graphical tool like an editor or a standalone app.

To allow for this flexibility, Yeoman provides a set of user interface element abstractions. It is your responsibility as an author to only use those abstractions when interacting with your end user. Using other ways will probably prevent your generator from running correctly in different Yeoman tools.

For example, it is important to never use `console.log()` or `process.stdout.write()` to output content. Using them would hide the output from users not using a terminal. Instead, always rely on the UI generic `this.log()` method, where `this` is the context of your current generator.

User interactions

Prompts

Prompts are the main way a generator interacts with a user. The prompt module is provided by Inquirer.js (https://github.com/SBoudrias/Inquirer.js) and you should refer to its API (https://github.com/SBoudrias/Inquirer.js) for a list of available prompt options.

The `prompt` method is asynchronous and returns a promise. You'll need to return the promise from your task in order to wait for its completion before running the next one. (learn more about asynchronous task (/authoring/running-context.html))

```
module.exports = class extends Generator {
  async prompting() {
    const answers = await this.prompt([
      {
        type: "input",
        name: "name",
        message: "Your project name",
        default: this.appname // Default to current folder name
      },
      {
        type: "confirm",
        name: "cool",
        message: "Would you like to enable the Cool feature?"
      }
    ]);

    this.log("app name", answers.name);
    this.log("cool feature", answers.cool);
  }
};
```

Note here that we use the `prompting` queue (/authoring/running-context.html) to ask for feedback from the user.

Using user answers at a later stage

A very common scenario is to use the the user answers at a later stage, e.g. in `writing` queue (/authoring/file-system.html). This can be easily achieved by adding them to `this` context:

```
module.exports = class extends Generator {
  async prompting() {
    this.answers = await this.prompt([
      {
        type: "confirm",
        name: "cool",
        message: "Would you like to enable the Cool feature?"
      }
    ]);
  }

  writing() {
    this.log("cool feature", this.answers.cool); // user answer `cool` used
  }
};
```

Remembering user preferences

A user may give the same input to certain questions every time they run your generator. For these questions, you probably want to remember what the user answered previously and use that answer as the new `default`.

Yeoman extends the Inquirer.js API by adding a `store` property to question objects. This property allows you to specify that the user provided answer should be used as the default answer in the future. This can be done as follows:

```
this.prompt({
  type: "input",
  name: "username",
  message: "What's your GitHub username",
  store: true
});
```

Note: Providing a default value will prevent the user from returning any empty answers.

If you're only looking to store data without being directly tied to the prompt, make sure to checkout the Yeoman storage documentation (/authoring/storage.html).

Arguments

Arguments are passed directly from the command line:

```
yo webapp my-project
```

In this example, `my-project` would be the first argument.

To notify the system that we expect an argument, we use the `this.argument()` method. This method accepts a `name` (String) and an optional hash of options.

The `name` argument will then be available as: `this.options[name]`.

The options hash accepts multiple key-value pairs:

- `desc` Description for the argument
- `required` Boolean whether it is required
- `type` String, Number, Array (can also be a custom function receiving the raw string value and parsing it)
- `default` Default value for this argument

This method must be called inside the `constructor` method. Otherwise Yeoman won't be able to output the relevant help information when a user calls your generator with the help option: e.g. `yo webapp --help`.

Here is an example:

```
module.exports = class extends Generator {
  // note: arguments and options should be defined in the constructor.
  constructor(args, opts) {
    super(args, opts);

    // This makes `appname` a required argument.
    this.argument("appname", { type: String, required: true });

    // And you can then access it later; e.g.
    this.log(this.options.appname);
  }
};
```

Argument of type `Array` will contain all remaining arguments passed to the generator.

Options

Options look a lot like arguments, but they are written as command line *flags*.

```
yo webapp --coffee
```

To notify the system that we expect an option, we use the `this.option()` method. This method accepts a `name` (String) and an optional hash of options.

The `name` value will be used to retrieve the option at the matching key `this.options[name]`.

The options hash (the second argument) accepts multiple key-value pairs:

- `desc` Description for the option
- `alias` Short name for option
- `type` Either Boolean, String or Number (can also be a custom function receiving the raw string value and parsing it)
- `default` Default value
- `hide` Boolean whether to hide from help

Here is an example:

```
module.exports = class extends Generator {
  // note: arguments and options should be defined in the constructor.
  constructor(args, opts) {
    super(args, opts);

    // This method adds support for a `--coffee` flag
    this.option("coffee");

    // And you can then access it later; e.g.
    this.scriptSuffix = this.options.coffee ? ".coffee" : ".js";
  }
};
```

Outputting Information

Outputting information is handled by the `this.log` module.

The main method you'll use is simply `this.log` (e.g. `this.log('Hey! Welcome to my awesome generator')`). It takes a string and outputs it to the user; basically it mimics `console.log()` when used inside of a terminal session. You can use it like so:

```
module.exports = class extends Generator {
  myAction() {
    this.log("Something has gone wrong!");
  }
};
```

There's also some other helper methods you can find in the API documentation (https://yeoman.github.io/environment/TerminalAdapter.html).