GENERATOR RUNTIME CONTEXT

One of the most important concepts to grasp when writing a Generator is how methods are running and in which context.

Prototype methods as actions

Each method directly attached to a Generator prototype is considered to be a task. Each task is run in sequence by the Yeoman environment run loop.

In other words, each function on the object returned by `Object.getPrototypeOf(Generator)` will be automatically run.

Helper and private methods

Now that you know the prototype methods are considered to be a task, you may wonder how to define helper or private methods that won't be called automatically. There are three different ways to achieve this.

1. Prefix method name by an underscore (e.g. `_private_method`).

```
class extends Generator {
  method1() {
    console.log('hey 1');
  }

  _private_method() {
    console.log('private hey');
  }
}
```

2. Use instance methods:

```
class extends Generator {
  constructor(args, opts) {
    // Calling the super constructor is important so our generator is
    super(args, opts)

    this.helperMethod = function () {
      console.log('won\'t be called automatically');
    };
  }
}
```

3. Extend a parent generator:

```
class MyBase extends Generator {
  helper() {
    console.log('methods on the parent generator won\'t be called aut
  }
}

module.exports = class extends MyBase {
  exec() {
    this.helper();
  }
};
```

The run loop

Running tasks sequentially is alright if there's a single generator. But it is not enough once you start composing generators together.

That's why Yeoman uses a **run loop**.

The run loop is a queue system with priority support. We use the Grouped-queue (https://github.com/SBoudrias/grouped-queue) module to handle the run loop.

Priorities are defined in your code as special prototype method names. When a method name is the same as a priority name, the run loop pushes the method into this special queue. If the method name doesn't match a priority, it is pushed in the `default` group.

In code, it will look this way:

```
class extends Generator {
  priorityName() {}
}
```

You can also group multiple methods to be run together in a queue by using a hash instead of a single method:

```
Generator.extend({
  priorityName: {
    method() {},
    method2() {}
  }
});
```

(Note that this last technique doesn't play well with JS `class` definition)

The available priorities are (in running order):

1. `initializing` - Your initialization methods (checking current project state, getting configs, etc)
2. `prompting` - Where you prompt users for options (where you'd call `this.prompt()`)
3. `configuring` - Saving configurations and configure the project (creating `.editorconfig` files and other metadata files)
4. `default` - If the method name doesn't match a priority, it will be pushed to this group.
5. `writing` - Where you write the generator specific files (routes, controllers, etc)
6. `conflicts` - Where conflicts are handled (used internally)
7. `install` - Where installations are run (npm, bower)
8. `end` - Called last, cleanup, say *good bye*, etc

Follow these priorities guidelines and your generator will play nice with others.

Asynchronous tasks

There are multiple ways to pause the run loop until a task is done doing work asynchronously.

The easiest way is to **return a promise**. The loop will continue once the promise resolves, or it'll raise an exception and stop if it fails.

If the asynchronous API you're relying upon doesn't support promises, then you can rely on the legacy `this.async()` way. Calling `this.async()` will return a function to call once the task is done. For example:

```
asyncTask() {
  var done = this.async();

  getUserEmail(function (err, name) {
    done(err);
  });
}
```

If the `done` function is called with an error parameter, the run loop will stop and an exception will be raised.

Where to go from here?

Now that you know a bit more about the running context of yeoman, you can continue by reading user interactions (/authoring/user-interactions.html).