# Kitchen helpers step-by-step

## Reading the assignment and collecting the information

At a first read-through it is worthwhile to identify the domain:

- The entities that will interact with each other
- The possible actions involving these entities which will make up the use cases.

The entities:

- Kitchen
- Chef
- Cook
- Kitchen Helper
- Ingredient

The actions:

- Hire employees to the kitchen.
- Conduct a shift in the kitchen.
- The chef requests an ingredient.
- The helper checks for and returns an ingredient.

There are more, but let's start out with these as these are the "interesting" parts of the assignment. The plan can be extended with the plainer parts later.

Now let's create a plan.

```
@startuml
skinparam classAttributeIconSize 0

class Kitchen {
  + hireEmployee(employee: Employee): void
  + conductAShift(): void
}
abstract class Employee {}
class Chef {
  + requestIngredient(ingredient: Ingredient): Optional<Ingredient>
}
class Cook {}
class KitchenHelper {
  + giveUpIngredient(ingredient: Ingredient): Optional<Ingredient>
  + yell(): void
}
enum Ingredient {
  POTATO
  MEAT
  CARROT
}

Kitchen *-- Employee
Employee <|-- Cook
Employee <|-- Chef
Employee <|-- KitchenHelper
@enduml
```

An `Employee` abstract class is added so the hiring can be generalized. No fields are added yet. All methods are added as public. This might change on later

iterations. `requestIngredient` and `giveUpIngredient` return an `Optional<Ingredient>` to emphasize the fact that they might not return any.

## Setting up communication

The next big question (before flesh out the classes) is

- how will the **helpers** know if an **ingredient** has been requested?
- how will the **chef** know if an ingredient has been checked by all **helpers**?

There is an invariant here to consider:

- Only the **helpers** who are in the same kitchen as the **chef** should try to fulfill ingredient requests.

For this there needs to be some organization. The entity that has all the information is the **Kitchen**. One option is to have **Kitchen** orchestrate this. However we don't want to give full access for the kitchen to the Chef. So we need to extract only the ingredient request behavior from the kitchen. Let's introduce an interface.

```
@startuml
skinparam classAttributeIconSize 0
interface IngredientStore {
  + requestIngredient(ingredient: Ingredient): Optional<Ingredient>
}

class Kitchen implements IngredientStore {
  - chef: Chef
  - helpers: Set<KitchenHelper>
  - hireChef(chef: Chef): void
  + hireEmployee(employee: Employee)
  + requestIngredient(ingredient: Ingredient): Optional<Ingredient>
}

class Chef {
  - store: IngredientStore
  + setStore(store: IngredientStore): void
}

class KitchenHelper {
  - ingredients: Map<Ingredient, Integer>
  - hasIngredient(ingredient: Ingredient): boolean
  - decreaseAmountOf(ingredient: Ingredient): void
  + giveUpIngredient(ingredient: Ingredient): Optional<Ingredient>
  + yell(): void
}

Kitchen *-- Chef
Kitchen *-- KitchenHelper
@enduml
```

## Scenarios

Case: One of the **helpers** have the **ingredient**:

```
@startuml
chef -> "ingredient store": request(meat)
"ingredient store" -> "helper 1": giveUpIngredient(meat)
"helper 1" -> "helper 1": hasIngredient(meat) -> false
"helper 1" -> "ingredient store": Optional.empty()
"ingredient store" -> "helper 2": giveUpIngredient(meat)
"helper 2" -> "helper 2": hasIngredient(meat) -> true
"helper 2" -> "helper 2": decreaseAmountOf(meat)
"helper 2" -> "ingredient store": Optional.of(meat)
```

```
"ingredient store" -> "chef": Optional.of(meat)
@enduml
```

When the **ingredient** is provided by one of the **helpers** there is no need to check the remaining ones.

Case: None of the **helpers** has the **ingredient**:

```
@startuml
participant chef
participant "ingredient store"
collections helpers

chef -> "ingredient store": request(meat)
"ingredient store" -> helpers: giveUpIngredient(meat)
...
helpers -> "ingredient store": Optional.empty()
"ingredient store" -> helpers: yell
...
"ingredient store" -> chef: Optional.empty()
@enduml
```

## Hiring a chef

When a **chef** is hired the **kitchen** needs to set her **ingredient store** by the kitchen itself.

If there was already a **chef** prior to the new one then the old one needs to set their **ingredient store** to null.

A good candidate for unit testing.

## Employees

Every **employee** has:

- a **name**
- a **birthdate**
- and a **salary**

These information are all stored in the `Employee` abstract class.

**Salary** is stored in an integer. If there is need for subdivision then the BigDecimal class can be used.

Note: Never store money in a floating point format as they have imprecisions in their representation.

Chefs and cooks can both cook in different way. There are different ways with different trade-offs:

- [ ] Have a `Cooker` interface that is implemented both by **chefs** and **cooks**.
- [ ] Have the above interface and also have an abstract class that extends `Employee` and have that class implement the `Cooker` interface.
- [X] Just have the `CookerEmployee` abstract class.

The **knives** are just represented as a “yes or no” thing so they don’t need objects. (This might change in a later sprint)

A `cook()` method is added with the common logic of checking whether the employee has received their knife set. If not so: it throws an exception. Otherwise it calls the `cookWithKnives()` abstract method.

**Kitchen helpers** need to refill their stock.

The **kitchen** holds multiple references to a given **employee** as it might need it in different contexts:

- the **chef**
- all the cookers
- all the employees

This is convenient but introduces complexity to the application. This is where unit tests really pay back the invested effort.

## Final UML

Putting it all together:

```
@startuml
skinparam classAttributeIconSize 0
interface IngredientStore {
  + requestIngredient(ingredient: Ingredient): Optional<Ingredient>
}

class Kitchen implements IngredientStore {
  - employees: Set<Employee>
  - chef: Chef
  - cookers: Set<CookerEmployee>
  - helpers: Set<KitchenHelper>
  - hireChef(chef: Chef): void
  + hireEmployee(employee: Employee): void
  + conductAShift(): void
}

abstract class Employee {
  - name: String
  - birthDate: LocalDate
  - salary: int
  + printTax(): void
}

abstract class CookerEmployee {
  - hasKnife: boolean
  + receiveKnife(): void
  + hasKnife(): boolean
  + cook(): void
  # {abstract} cookWithKnives(): void
}

class Chef {
  - ingredientStore: IngredientStore
  + setStore(store: IngredientStore): void
}

class Cook { }

class KitchenHelper {
  - ingredients: Map<Ingredient, Integer>
  - hasIngredient(ingredient: Ingredient): boolean
  - decreaseAmountOf(ingredient: Ingredient): void
  + giveUpIngredient(ingredient: Ingredient): Optional<Ingredient>
  + refillIngredients(): void
  + yell(): void
}

enum Ingredient {
```

```
  POTATO
  MEAT
  CARROT
}

Kitchen *-- Employee
Employee <|-- CookerEmployee
Employee <|-- KitchenHelper
CookerEmployee <|-- Cook
CookerEmployee <|-- Chef
@enduml
```

# Implementation

A package is created to contain the whole application: `com.codecool.kitchenmanagement`

## Employee inheritance chain

The employee classes are in thier own subpackage: `com.codecool.kitchenmanagement.employees`.

```
package com.codecool.kitchenmanagement.employees;

/**
 * Abstract class representing a kitchen employee.
 */
public abstract class Employee {
    private String name;
    private LocalDate birthDate;
    private int salary;

    /**
     * Creates an employee instance
     *
     * @param name the full name of the employee
     * @param birthDate the employee's year of birth
     * @param salary the monthly salary of the employee
     */
    public Employee(String name, LocalDate birthDate, int salary) {
        this.name = name;
        this.birthDate = birthDate;
        this.salary = salary;
    }

    /**
     * Get the full name of the employee
     *
     * @return the full name of the employee as a single string
     */
    public String getName() {
        return name;
    }

    /**
     * Get the date of birth of the employee
     *
     * @return the date of birth of the employee as a LocalDate
     */
    public LocalDate getbirthDate() {
        return birthDate;
    }

    /**
     * Get the monthly salary of the employee.
     *
     * @return the monthly salary of the employee.
```

```
     */
    public int getSalary() {
        return salary;
    }

    /**
     * Set the monthly salary of the employee.
     *
     * @param salary the new monthly salary of the employee.
     */
    public void setSalary(int salary) {
        this.salary = salary;
    }

    /**
     * Prints the employee's tax to the console.
     */
    public void printTax() {
        int tax = (int) (salary * 0.99);
        System.out.format("My tax is %d.%s", tax, System.lineSeparator());
    }

    /**
     * String representation of the employee.
     * <p>
     * Prints the name and birth year of the employee.
     *
     * @return the employee represented as a printable string
     */
    @Override
    public String toString() {
        return String.format("Employee { name = \"%s\", year of birth = %d }",
                             name, birthDate.getYear());
    }
}
```

The `CookerEmployee` abstract class expands it with the cooking and knife set logic:

```
package com.codecool.kitchenmanagement.employees;

/**
 * Abstract class representing employees who can cook and possibly
 * possess a knife set.
 */
public abstract class CookerEmployee extends Employee {
    private boolean hasKnife; // initialized to false

    /**
     * {@inheritDoc}
     */
    public CookerEmployee(String name, LocalDate birthDate, int salary) {
        super(name, birthDate, salary); // call the constructor of Employee
    }

    /**
     * Give a knife set to the employee
     * <p>
     * An employee without a knife set cannot cook.
     */
    public void receiveKnife() {
        hasKnife = true;
    }

    /**
     * Checks whether the employee has already received their knife
     * set.
     *
     * @return true if the employee has already received their knife
```

```
     *         set.  Return false otherwise.
     */
    public boolean hasKnife() {
        return hasKnife;
    }

    /**
     * Performs the cooking process of the employee.
     *
     * Override the {@code cookWithKnives} method to implement
     * specific cooking logic in child classes.
     *
     * @throws IllegalStateException if the employee hasn't received
     *         their knife set yet.
     */
    public final void cook() {
        if (!hasKnife) {
            var formatString = // extract to static final?
                "Can't cook: The employee %s hasn't received their knife set yet.";
            var message = String.format(formatString, toString());
            throw new IllegalStateException(message);
        }

        cookWithKnives();
    }

    /**
     * Perform class-specific cooking logic.
     */
    protected abstract void cookWithKnives();
}
```

**Kitchen helpers** are **employees**.

```
package com.codecool.kitchenmanagement.employees;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;
import java.util.Random;

import com.codecool.kitchenmanagement.Ingredient;

/**
 * A kitchen helper is an employee with a stock of ingredients.
 *
 * At any given moment a helper has [0-3] of any ingredient.
 */
public class KitchenHelper extends Employee {
    private static final int MAX_NUMBER_OF_INGREDIENTS = 3;
    private final Random random = new Random(); // per-instance random state for refill
    private final Map<Ingredient, Integer> ingredients = new HashMap<>();

    /**
     * Creates a kitchen helper with no ingredients of any kind.
     */
    public KitchenHelper(String name, LocalDate birthDate, int salary) {
        super(name, birthDate, salary);
        for (var ingredient: Ingredient.values()) {
            ingredients.put(ingredient, 0);
        }
    }

    /**
     * Refills the ingredients held by the kitchen helper.
     * <p>
     * Each kind of ingredient is refilled to a maximum of 3 and a
     * minimum of the former value.
```

```
     */
    public void refillIngredients() {
        for (var ingredient: Ingredient.values()) {
            var amount = random.nextInt(MAX_NUMBER_OF_INGREDIENTS+1);
            ingredients.merge(ingredient, amount, Math::max);
        }
    }

    /**
     * Give up an ingredient if the helper possesses it.
     *
     * @param the type of the requested ingredient.
     *
     * @return an optional wrapped ingredient if found.  An empty
     *         optional otherwise.
     *
     * @throws IllegalStateException when the ingredient stocks have
     *         incosistencies. (unlikely)
     */
    public Optional<Ingredient> giveUpIngredient(Ingredient ingredient) {
        if (hasIngredient(ingredient)) {
            decreaseAmountOf(ingredient);
            return Optional.of(ingredient);
        }

        return Optional.empty();
    }

    /**
     * Yells about running out of ingredients.
     */
    public void yell() {
        System.out.println("We're all out!");
    }

    private boolean hasIngredient(Ingredient ingredient) {
        return ingredients.get(ingredient) > 0;
    }

    private void decreaseAmountOf(Ingredient ingredient) {
        var amount = ingredients.get(ingredient);
        if (amount == 0) {
            throw new IllegalStateException("Tried to decrease empty ingredient
stock.");
        }

        ingredients.replace(ingredient, amount-1);
    }
}
```

The **cook** is a cooker employee who's just going about his business. Very straight-forward.

```
package com.codecool.kitchenmanagement.employees;

/**
 * A cooker employee.  Kitchens employ multiple cooks.
 */
public class Cook extends CookerEmployee {
    /**
     * {@inheritDoc}
     */
    public Cook(String name, LocalDate birthDate, int salary) {
        super(name, birthDate, salary);
    }

    /**
     * perform the cooking duties of a cook.
     */
```

```
    public void cookWithKnives() {
        System.out.println("I'm cooking");
        // might expand on it in later sprints
    }
}
```

The **chef** is just a bit more trickier.

```
package com.codecool.kitchenmanagement.employees;

import java.util.Optional;
import java.util.Random;
import java.util.function.Function;

import com.codecool.kitchenmanagement.Ingredient;

public class Chef extends CookerEmployee {
    // instance-wise random for requests
    private final Random random = new Random();
    // initialized to `null`
    private IngredientStore store;

    /**
     * {@inheritDoc}
     */
    public Chef(String name, LocalDate birthDate, int salary) {
        super(name, birthDate, salary);
    }

    /**
     * Set the ingredient store of the chef.
     *
     * @param store the new ingredient store of the chef.
     */
    public void setStore(IngredientStore store) {
        this.store = store;
    }

    /**
     * In some cases the chef asks for an ingredient from the kitchen
     * helpers.
     * <p>
     * Requests and their results are logged.
     */
    @Override
    protected void cookWithKnives() {
        // not specified when, so going 50-50
        var shouldAskForIngredient = random.nextBoolean();

        if (!shouldAskForIngredient || store == null) {
            return;
        }

        var randomIngredient = Ingredient.randomIngredient();
        System.out.format("I need %s\n", randomIngredient);
        Optional<Ingredient> received = store.requestIngredient(randomIngredient);
        String ingredientName = received.map(Ingredient::toString).orElse("NOTHING");
        System.out.println("I got " + ingredientName);
    }
}
```

`store.requestIngredient()` is called by the `cookWithKnives()` override of chef.

## Ingredients

Ingredients form a simple enum with a method added that can choose one at random.

```
package com.codecool.kitchenmanagement;
```

```
import java.util.Random;

/**
 * Ingredients used for cooking.
 */
public enum Ingredient {
    MEAT,
    POTATO,
    CARROT;

    private static final Random RANDOM = new Random();

    /**
     * Return a randomly chosen ingredient.
     *
     * @return one of the ingredients.
     */
    public static Ingredient randomIngredient() {
        var allIngredients = Ingredient.values();
        var numOfIngredients = allIngredients.length;
        return allIngredients[RANDOM.nextInt(numOfIngredients)];
    }
}
```

## Ingredient store

As we don't want to spoil our full kitchen to the chefs, only a narrow functionality of it (`requestIngredient()`) we extract this into an interface:

```
package com.codecool.kitchenmanagement;

import java.util.Optional;

public interface IngredientStore {
    Optional<Ingredient> requestIngredient(Ingredient ingredient);
}
```

## Kitchen

```
package com.codecool.kitchenmanagement;

import java.util.HashSet;
import java.util.Set;
import java.util.Optional;
import java.util.stream.Stream;

import com.codecool.kitchenmanagement.employees.Chef;
import com.codecool.kitchenmanagement.employees.CookerEmployee;
import com.codecool.kitchenmanagement.employees.Employee;
import com.codecool.kitchenmanagement.employees.KitchenHelper;

/**
 * A kitchen where employees cook meals.
 */
public class Kitchen implements IngredientStore {
    private final Set<Employee> employees = new HashSet<>();
    private Chef chef; // starts out as `null`
    private final Set<CookerEmployee> cookers = new HashSet<>();
    private final Set<KitchenHelper> helpers = new HashSet<>();

    /**
     * Hire an employee
     * <p>
     * If the hired employee is a chef then release the former chef if
     * any.
     *
     * @param employee the employee to hire
```

```
     */
    public void hireEmployee(Employee employee) {
        employees.add(employee);

        if (employee instanceof Chef) {
            hireChef((Chef) employee);
        }

        if (employee instanceof CookerEmployee) {
            cookers.add((CookerEmployee) employee);
        }

        if (employee instanceof KitchenHelper) {
            KitchenHelper newHelper = (KitchenHelper) employee;
            helpers.add(newHelper);
        }
    }

    /**
     * Conduct a shift making meals.
     *
     * @throws IllegalStateException if there is no chef hired.
     */
    public void conductAShift() {
        if (chef == null) {
            throw new IllegalStateException("Can't start a shift without a chef.");
        }

        helpers.forEach(KitchenHelper::refillIngredients);

        for (var cooker : cookers) {
            try {
                cooker.cook();
            } catch (IllegalStateException e) {
                // report error but otherwise go on
                System.out.println("Error " + e.getMessage());
            }
        }
    }

    /**
     * Iterate through the kitchen helpers for the requested ingredient.
     * If the ingredient presents then returns with it.
     * In case of missing ingredient invokes kitchen helpers' yell method and retuns
with empty.
     *
     * @param ingredient demanded ingredient
     */
    @Override
    public Optional<Ingredient> requestIngredient(Ingredient ingredient) {
        for (KitchenHelper helper : helpers) {
            Optional<Ingredient> received = helper.giveUpIngredient(ingredient);
            if (!received.isEmpty()){
                return received;
            }
        }
        helpers.forEach(KitchenHelper::yell);
        return Optional.empty();
    }

    private void hireChef(Chef newChef) {
        if (this.chef != null) {
            fireChef();
        }
        this.chef = newChef;
        this.chef.setStore(this);
    }
```

```
    private void fireChef() {
        chef.setStore(null);
        employees.remove(chef);
    }
}
```

The call site of `requestIngredient` is not trivial at first.

1. **Kitchen** calls `cook` on all cooker employees.
2. Which eventually calls `cook` on the **chef**.
3. Which calls `cookWithKnives`.
4. Which calls `requestIngredient` on store (the kitchen as `IngredientStore`).

## Example program

One way of using the classes is the following:

```
package com.codecool.kitchenmanagement;

import java.util.List;

import com.codecool.kitchenmanagement.Kitchen;
import com.codecool.kitchenmanagement.employees.Employee;
import com.codecool.kitchenmanagement.employees.Chef;
import com.codecool.kitchenmanagement.employees.Cook;
import com.codecool.kitchenmanagement.employees.KitchenHelper;

public class Main {
    public static void main(String[] args) {
        var kitchen = new Kitchen();

        LocalDate oneDate = LocalDate.of(1990,1,1);
        var chef = new Chef("Chef McCheferson", oneDate, 1000);
        var cook1 = new Cook("Cook McCookface", oneDate.plusYears(3), 800);
        var cook2 = new Cook("Pan Fryingkind",  oneDate.plusYears(-2), 800);
        var helper1 = new KitchenHelper("Helper Helpowitz", oneDate.plusYears(1), 750);
        var helper2 = new KitchenHelper("Handy Smith",      oneDate.plusYears(3), 750);
        var helper3 = new KitchenHelper("Slim Quicktohelp", oneDate.plusMonths(4), 750);
        chef.receiveKnife();
        cook1.receiveKnife();
        cook2.receiveKnife();

        List<Employee> employees = List.of(chef, cook1, cook2,
                helper1, helper2, helper3);
        employees.forEach(kitchen::hireEmployee);

        kitchen.conductAShift();
    }
}
```

One possible output is:

```
Error Can't cook: The employee Employee { name = "Cook McCookface", year of birth = 1991
} hasn't received their knife set yet.
I'm cooking
Requesting MEAT
Got MEAT
```