

Applied session (Bootcamp)

Applied Session objectives - Bootcamp Week 4

In this week's Bootcamp, you will:

- implement interfaces
- group classes into packages

Bootcamp specifications



Assessment: This applied session is worth 3% of your final mark for FIT2099. It will be awarded in the form of a checkpoint to be completed during the applied session next week as it directly contributes to the bootcamp's final delivery in Week 6.



Please, read all the requirements listed below first, then start working on the design before doing any coding.

Common Requirements (REQ 1 & 2)

Again, there will be 2 common requirements for each bootcamp and they are reflected in the [marking rubric](#) accordingly. These 2 requirements will remain the same and consistent across all bootcamps and will be assessed every time. Please read them very carefully to get yourself familiar with the common requirements and be ready for future bootcamps as well!

Link: <https://edstem.org/au/courses/25302/lessons/84442/slides/576448>

Specific Tasks (REQ 3)

There will be 4 tasks under REQ3 for each bootcamp. These are designated to help you consolidate and assess your understanding of that specific week's content and are aligned with the Applied session objectives. The 4 tasks will be different for each week and will be built on top of the previous week's bootcamp. These tasks will cover both coding and design skills we anticipate you to hold throughout the learning journey.

Link: <https://edstem.org/au/courses/20991/lessons/71782/slides/477006>



Remember, if you encounter any issues or have any questions, please don't hesitate to reach out to a TA for help! If you still have questions after the applied session, you can post them on the Ed discussion forum or attend a consultation.

Bootcamp Common REQs (REQ 1 & 2)

Requirement 1. Your code on Gitlab

You must have a **minimum number of non-trivial commits** (refer to the rubric) to your GitLab repository. If your previous bootcamp work hasn't been assessed yet, and you want to continue this week's bootcamp, please create a new branch `bootcamp-week-4` from your `main` branch. Ensure that you merge it back into the `main` branch before the interview/assessment. **We only assess work inside the `main` branch.**



If you haven't figured out how to create a new branch, please, go through the material on [how to create a Git branch](#) and/or ask your TA for a demonstration.

Requirement 2. UML Class Diagram

Just like last week, let's start this bootcamp session by creating a UML class diagram for the system you plan to implement. But before you dive into coding, it's important to get feedback from a TA on your diagram. Remember, the system you're creating will involve multiple tasks, so make sure you read through the lab sheet carefully. Your diagram should cover all classes and their relationships, including associations, dependencies, and other UML relationships. You don't have to worry about adding attributes and methods to your diagram.



Please make sure to save your design documents, such as the UML class diagram, in the "docs/design" folder. Don't forget to add a prefix of "w4-" when naming the file (for example, "w4-uml.png"). Lastly, save the UML class diagram as a PNG or PDF file.

Bootcamp REQ3: Implementation

Requirement 3. Implementation

This week, we will improve our code quality by adding the package and documentation (Javadoc). Furthermore, as it is getting colder during the camping, we want to ignite the torch or oil lantern that we have in our campsite - by igniting these items, we can feel warmer. More details about it will be discussed in the following points:

Requirement 3a. Package and Javadoc

The first task that we need to complete for this week's bootcamp is to introduce a new layer of encapsulation: packages!

Packages allow us to organise our source code better by making it easier for developers to navigate through and locate files, especially in large software projects. In terms of object-oriented programming, as mentioned earlier, packages also help us with enforcing encapsulation. Java provides a default (also known as package-private) access modifier that allows certain members of a class to be accessed by another class in the same package. Therefore, if classes need to work together, grouping related classes in the same package is a good idea, as packages do not expose the class members with the package-private access modifier to unrelated external classes.

To give you an idea, in this project, one package that we can create is the "items" package, which should include the `Bedroll` class, `Bottle` class, `FlintAndSteel` class, and `Item` class. Now, you should create other packages that you think are relevant for this project.



Make sure to follow the correct naming convention for packages as well! (see <https://google.github.io/styleguide/javaguide.html#s5-naming>) Apart from that, here is a good article why we really need to construct packages: <https://www.linkedin.com/pulse/unveiling-why-packages-used-java-bhavishya-ambati-tga2c/>



Please make sure that packages are also **visible in your UML class diagram**. The classes should be added into the **correct packages, consistent with the implementation**. Please remember to tidy up the UML class diagram to improve the **clarity and readability of the class diagram**.

For the second task, you should document your project with [Javadoc](#). You need to make sure that the class members that have at least the protected access modifier (i.e. protected and public) are well-documented. This is because if your code were to be released as a library to be used by other developers, class members with protected or public access modifiers would be visible to the client. For example, protected class members can be accessed by the client's class if it extends your class. From the reasoning above, you do not need to document class members that are private or package-private (default access modifier), as they are not visible to the client.

In addition to documenting the class members, don't forget to provide Javadoc documentation for the classes! For classes, the Javadoc documentation should be put above the class declaration.



Make sure to follow good practices when documenting your project. For example, for a method with either a **protected** or **public** access modifier, you should document what the parameters accepted by the method are and what its return value will be. Remember to use the **correct tags** for your documentation (e.g. @param, @return, etc.)!

Requirement 3b. Refactoring - allowableActions()

This week's content is closely related to Interface, so let's refactor our code from last week. Notice that we have an `allowableActions()` method inside Camper, Campsite, and Backpack, and it is very likely that we want to add this method to classes we add in the future. So, how can we standardise this method name for any future classes that want to have this method? One way to achieve that is by having an interface, which we can call the **ActionCapable** interface.

```
public interface ActionCapable { 8 usages 8 implementations
    public List<Action> allowableActions(Camper camper);
}
```



As we can see here, our `allowableActions()` have Camper as the parameter value - we will understand why we need it once we complete Requirement 3d - let's adjust any classes that have `allowableAction()` to implement this new interface.

Another thing we missed from last week is that we don't have any `allowableAction()` in our Item class yet. Let's make the **Item** class implement the interface that we just created.

Requirement 3c. Status

Now, let's move on to how we can use [enumeration](#). Let's say we want to ignite an **OilLantern**, we'd need to check whether the camper has a **FlintAndSteel** object inside their backpack. The easiest way to do this is by iterating through every object inside the backpack and checking whether it is **FlintAndSteel** or not. However, if we have another item that can ignite the OilLantern, then we may need to add another if-statement in our code and we may even need to use `instanceof` in the implementation, which should be [avoided](#).

One way around it is by using **enumeration**. We'd suggest reading this [assignment support module](#) first before we continue our code to get the idea of why we implement Ability as an enumeration.



An **Enum** type is a special data type that enables a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week. Readings: <https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>

In short, an item can have zero or more abilities. For example, `FlintAndSteel` can be used to `IGNITE_FIRE`. Try to **add a set of Ability inside our Item** as a private attribute and **add the following methods** to our Item class.

```
protected void addCapability(Enum<Ability> capability) { this.statuses.add(capability); }

protected void removeCapability(Enum<Ability> capability) { this.statuses.remove(capability); }

public boolean hasCapability(Enum<Ability> capability) { return this.statuses.contains(capability); }
```



Notice that we encapsulate the `addCapability()` and `removeCapability()` as protected because we want to limit the boundary of attaching and detaching the Ability inside the Item and its child classes only.

As mentioned above, `FlintAndSteel` has the ability to burn something, so let's create a new enumeration, name it `Ability`, and add one value, which is `IGNITE_FIRE`. As we have already created the methods needed to add and remove an item's capability, we can easily attach the `Ability.IGNITE_FIRE` to the `FlintAndSteel` itself. For now, we can **add this line**: `this.addCapability(Ability.IGNITE_FIRE)` to the `FlintAndSteel`'s constructor.

The next question is, how do we know whether the camper has items with a certain ability, inside the backpack or not? Let's add this code to the Camper class.

```
public boolean hasCapability(Enum<Ability> capability){
    for (Item item : backpack.getAllItems()) {
        if (item.hasCapability(capability)) {
            return true;
        }
    }
    return false;
}
```

We might notice that we haven't implemented the `backpack.getAllItems()` method yet. In essence, this method would simply return the list of items stored in the Backpack class. However, since lists in Java are mutable, directly returning the list would allow external modification of its contents, which isn't desirable. To prevent this, we should use a [defensive copy](#) in this situation.



The general idea of defensive copy is pretty straightforward - instead of passing the actual value, we just create a copy of it before returning the value.

Now that we know of the "defensive copying" technique, let's **create the `getAllItems()` in our Backpack class**.

Requirement 3d. Flammable Object

Let's go back to our new item in this camp again, which is OilLantern. The oil lantern that we bring to the campsite has a **weight of 1.5 kg**.

We want to ignite our oil lanterns. Let's introduce a new [interface](#) for that, named Flammable. As we are aware, the interface is a blueprint of a class, and from this scenario, we do have something in common amongst all the flammable things: they can be ignited, so we can have a method called ignitedBy (as shown in the picture below).

```
public interface Flammable {  
    public String ignitedBy(Camper camper);  
}
```



We need Camper as the parameter value because every time we ignite any flammable object, we would like to decrease the Camper's coldness level by some values.

The next thing to consider is which class should implement the `Flammable` interface. The answer is any object that can catch fire; in this case, that means the OilLantern class. When a class implements an interface, it must provide its own versions of the methods defined in that interface. In this example, each time we light the lantern, the **coldness level will drop by 2**.

```
@Override  
public String ignitedBy(Camper camper) {  
    camper.decreaseColdnessLevel(REDUCE_COLDNESS_VALUE);  
    return this + " is ignited by " + camper + " and it reduces the coldness by " + REDUCE_COLDNESS_VALUE;  
}
```



Notice that we have `REDUCE_COLDNESS_VALUE` in the picture above. We call this a constant, which is similar to a variable since we can assign a value to it, but once assigned, the value cannot be changed. In Java, we use the `final` keyword to implement constants. Also, we must use `UPPER_CASE` when naming constants.

Let's make another flammable item, which is a **torch**, weighs 1kg and can be found at a campsite, and every time we ignite it, it will **reduce the coldness by 1**.

Let's go back to the **IgniteAction** again, this action will take any flammable materials, and we will call `ignitedBy()` in our `execute` method.

```

import campings.Camper;
import campings.Campsite;
import campings.items.Flammable;

public class IgniteAction extends Action{
    private Flammable flammable;

    public IgniteAction(Flammable flammable){
        this.flammable = flammable;
    }

    @Override
    public String execute(Camper camper, Campsite campsite) {
        return flammable.ignitedBy(camper);
    }

    @Override
    public String menuDescription(Camper camper) {
        return camper + " will ignite " + flammable.getClass().getSimpleName();
    }
}

```

Based on the provided image, the key concept is to apply the [dependency injection technique](#) by injecting `Flammable` objects into the `IgniteAction` class. This allows us to utilise the `ignitedBy` method defined in the `Flammable` interface. We will explore dependency injection in more detail during week 6.

Lastly, we need to modify the `allowableActions()` methods in both `OilLantern` and `Torch` classes to accommodate the `IgniteAction`. Keep in mind, these items can only be ignited if the camper possesses the `IGNITE_FIRE` ability - so make sure to check the camper's capabilities using the `hasCapability()` method.

Last thing, we just need to update `OilLantern` and `Torch`'s `allowableActions()` so we can use the `IgniteAction`. Remember that we can only ignite the `OilLantern` and `Torch` if and only if the camper has the ability of `IGNITE_FIRE` - try to refer back to the `hasCapability()` method that we have inside the camper itself.



Checkpoint for Week 4 bootcamp:

- Package
- Javadoc
- ActionCapable interface - will be implemented by Campsite, Camper, Backpack, and Item
- Flammable interface - will be implemented by OilLantern and Torch
- New Ability (IGNITE_FIRE) as FlintAndStell can produce a fire which will be used to ignite the lantern and torch.

Conditions for your bootcamp submission

(Same as last week)



Be mindful of your bootcamp submission time as that depends on your allocated applied session time.

Moodle submissions are not required as all bootcamp assessments will be conducted during the applied session.

1. Last-Minute commits will not be considered

- Any commits made **during class** when the bootcamp is due for marking **will not** be considered.
- The version of your work at the official submission deadline is what will be assessed.
- Make sure all your work is committed and pushed **well before** the marking session begins.

2. Researching Previous Semester Works on GitHub

- Although we do not recommend this, you **are allowed** to refer to past semester's work by conducting your own research on GitHub. Please, note that the quality of any referenced work is **not guaranteed**—it could be a high-distinction submission or a failed attempt.
- This is an opportunity for you to learn how to **properly cite code**.
- You **must write all the code yourself** instead of copying and pasting.
- If you choose to build on someone else's work, you **must** ensure your contribution is original.
- If you are found copying code directly or excessively relying on another project, an **academic integrity alert** will be triggered.
- During your interview, you must show **a complete and solid understanding** of both the code you wrote and any referenced code. **Failure to do so will result in a score of 0 for the interview.**

3. Using ChatGPT & Proper Citation

- If you use ChatGPT to generate code, you **must** provide proper citations.
- Citations must be included in:
 - **In-line comments** within your code
 - **As per the citation guidelines for each assessment task in Moodle. For example:**



AI & Generative AI tools may be used in GUIDED ways within this assessment / task as per the guidelines provided.

In this task, AI can be used as specified for one or more parts of the assessment task as per the instructions.

Within this class, you are welcome to use foundation models (ChatGPT, GPT, DALL-E, Stable Diffusion, Midjourney, GitHub Copilot, and anything after) in a totally unrestricted fashion, for any purpose, at no penalty. However, you should note that all large language models still have a tendency to make up incorrect facts and fake citations, code generation models have a tendency to produce inaccurate outputs, and image generation models can occasionally come up with highly offensive products. You will be responsible for any inaccurate, biased, offensive, or otherwise unethical content you submit regardless of whether it originally comes from you or a foundation model. If you use a foundation model, its contribution must be acknowledged in the submission (in the form of a txt, pdf or word file titled FoundationModelsUsed added to your "docs" folder in GIT); you will be penalised for using a foundation model without acknowledgement.

Having said all these disclaimers, the university's policy on plagiarism still applies to any uncited or improperly cited use of work by other human beings, or submission of work by other human beings as your own. Moreover, missing contribution logs/GIT commits and/or failing any handover interview will be treated as a potential breach of academic integrity which will be further investigated.

Where used, AI must be used responsibly, clearly documented and appropriately acknowledged ([see Learn HQ](#)).

Any work submitted for a mark must:


1. represent a sincere demonstration of your human efforts, skills and subject knowledge that you will be accountable for.
2. adhere to the guidelines for AI use set for the assessment task.
3. reflect the University's commitment to academic integrity and ethical behaviour.

Inappropriate AI use and/or AI use without acknowledgement will be considered a breach of academic integrity.

- During your interview, you must demonstrate **a solid understanding** of any ChatGPT-generated code. **Failure to do so will result in a score of 0 for the interview.**

Marking rubric (Checkpoint)

In the **next** applied class, you will need to demonstrate progress through **a brief checkpoint review**. In this, you will showcase your development, progress and understanding of this week's bootcamp. Your teacher will use the following rubric to mark the progress of this bootcamp.

 A pre-requisite for marking is that Req (1) should be awarded at 1 or 0.5 point.

Req (1). GIT Commits

Excellent - Work has been progressively committed to GIT (more than 5 well formatted commit messages), changes and progress have been correctly commented, and progression of ideas and development seems clear.

1 points

Satisfactory - There is some evidence of progression of ideas and development (between 5 and 2 commit messages). There is some inconsistency or vagueness in commit messages.

0.5 points

Unsatisfactory - There is little or no evidence of progression of ideas and development, work is not commented correctly (vague commit messages) or commit messages do not correspond to the changes in the code or all the code seems to have been uploaded in one go.

0 points

Req (2). UML Design documents - clarity

Excellent - A design document is provided, is clear and syntactically correct. The design seems to fully address the requirements.

2 points

Satisfactory - A design document is provided and it seems to address most of the requirements. All required classes are present and the UML syntax is generally correct. Some key relationship(s) between classes are/is missing and there may be one or more minor syntax errors which do not affect the overall clarity of the documentation.

1 points

Unsatisfactory - A design document is not provided, OR the design does not address the requirements OR the documentation is unclear and not in accordance with UML syntax conventions

or the application does not run OR Req (1) got 0 points.

0 points

Req (3). Implementation

Excellent - The implementation (the code) seems to fully address the requirements as per the UML design document (applying Object-Oriented principles).

2 points

Satisfactory - The implementation addresses the requirements with some design issues (not following the recommended design principles or the UML design document).

1 points

Unsatisfactory - The implementation does not really address the requirements even though some attempt was done or the application does not run OR Req (1) got 0 points.

0 points

Expected Output (Text Format)



You do not have to follow the expected output printing format strictly. As long as you have implemented all functionalities correctly and printed out the required information, please feel free to use your own formatting.

Try to check if we unpack our flint-and-steel, do we still have the action to ignite oil lantern/torch or not?

```
#####
```

```
Here are the items that Cloudy has in the Backpack (10.00 / 10.00kg):
```

```
Bedroll (KAMUI) has weight of 7.00 kg - to rest.
```

```
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
```

```
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.
```

```
OilLantern (Feuerhand) has weight of 1.50 kg - to reduce the cold
```

```
Here are the items that we have on campsite:
```

```
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.
```

```
Torch (Pine wood) has weight of 1.00 kg - to light the surrounding
```

```
#####
```

```
a: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
```

```
b: Cloudy (hydration level: 20, coldness level: 20) will pack Torch to the backpack
```

```
c: Cloudy (hydration level: 20, coldness level: 20) will unpack Bedroll from the backpack
```

```
d: Cloudy (hydration level: 20, coldness level: 20) will unpack Bottle from the backpack
```

```
e: Cloudy (hydration level: 20, coldness level: 20) will unpack FlintAndSteel from the backpack
```

```
f: Cloudy (hydration level: 20, coldness level: 20) will unpack OilLantern from the backpack
```

```
g: Cloudy (hydration level: 20, coldness level: 20) will ignite OilLantern
```

```
c
```

```
Cloudy (hydration level: 20, coldness level: 20) removed Bedroll from the backpack
```

```
#####
```

```
Here are the items that Cloudy has in the Backpack (3.00 / 10.00kg):
```

```
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
```

```
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.
```

```
OilLantern (Feuerhand) has weight of 1.50 kg - to reduce the cold
```

```
Here are the items that we have on campsite:
```

```
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.
```

```
Torch (Pine wood) has weight of 1.00 kg - to light the surrounding
```

```
Bedroll (KAMUI) has weight of 7.00 kg - to rest.
```

```
#####
```

```
a: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
```

```
b: Cloudy (hydration level: 20, coldness level: 20) will pack Torch to the backpack
```

```
c: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
```

```
d: Cloudy (hydration level: 20, coldness level: 20) will unpack Bottle from the backpack
```

```
e: Cloudy (hydration level: 20, coldness level: 20) will unpack FlintAndSteel from the backpack
```

```
f: Cloudy (hydration level: 20, coldness level: 20) will unpack OilLantern from the backpack
```

g: Cloudy (hydration level: 20, coldness level: 20) will ignite OilLantern
b
Cloudy (hydration level: 20, coldness level: 20) packed Torch to the backpack

#####

Here are the items that Cloudy has in the Backpack (4.00 / 10.00kg):
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.
OilLantern (Feuerhand) has weight of 1.50 kg - to reduce the cold
Torch (Pine wood) has weight of 1.00 kg - to light the surrounding

Here are the items that we have on campsite:
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.
Bedroll (KAMUI) has weight of 7.00 kg - to rest.

#####

a: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
b: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
c: Cloudy (hydration level: 20, coldness level: 20) will unpack Bottle from the backpack
d: Cloudy (hydration level: 20, coldness level: 20) will unpack FlintAndSteel from the backpack
e: Cloudy (hydration level: 20, coldness level: 20) will unpack OilLantern from the backpack
f: Cloudy (hydration level: 20, coldness level: 20) will ignite OilLantern
g: Cloudy (hydration level: 20, coldness level: 20) will unpack Torch from the backpack
h: Cloudy (hydration level: 20, coldness level: 20) will ignite Torch
f
OilLantern is ignited by Cloudy (hydration level: 20, coldness level: 18) and it reduces the coldne

#####

Here are the items that Cloudy has in the Backpack (4.00 / 10.00kg):
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.
OilLantern (Feuerhand) has weight of 1.50 kg - to reduce the cold
Torch (Pine wood) has weight of 1.00 kg - to light the surrounding

Here are the items that we have on campsite:
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.
Bedroll (KAMUI) has weight of 7.00 kg - to rest.

#####

a: Cloudy (hydration level: 20, coldness level: 18) will pack Bedroll to the backpack
b: Cloudy (hydration level: 20, coldness level: 18) will pack Bedroll to the backpack
c: Cloudy (hydration level: 20, coldness level: 18) will unpack Bottle from the backpack
d: Cloudy (hydration level: 20, coldness level: 18) will unpack FlintAndSteel from the backpack
e: Cloudy (hydration level: 20, coldness level: 18) will unpack OilLantern from the backpack
f: Cloudy (hydration level: 20, coldness level: 18) will ignite OilLantern
g: Cloudy (hydration level: 20, coldness level: 18) will unpack Torch from the backpack
h: Cloudy (hydration level: 20, coldness level: 18) will ignite Torch
h
Torch is ignited by Cloudy (hydration level: 20, coldness level: 17) and it reduces the coldness by

#####

Here are the items that Cloudy has in the Backpack (4.00 / 10.00kg):
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.
OilLantern (Feuerhand) has weight of 1.50 kg - to reduce the cold
Torch (Pine wood) has weight of 1.00 kg - to light the surrounding

Here are the items that we have on campsite:
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.
Bedroll (KAMUI) has weight of 7.00 kg - to rest.

a: Cloudy (hydration level: 20, coldness level: 17) will pack Bedroll to the backpack
b: Cloudy (hydration level: 20, coldness level: 17) will pack Bedroll to the backpack
c: Cloudy (hydration level: 20, coldness level: 17) will unpack Bottle from the backpack
d: Cloudy (hydration level: 20, coldness level: 17) will unpack FlintAndSteel from the backpack
e: Cloudy (hydration level: 20, coldness level: 17) will unpack OilLantern from the backpack
f: Cloudy (hydration level: 20, coldness level: 17) will ignite OilLantern
g: Cloudy (hydration level: 20, coldness level: 17) will unpack Torch from the backpack
h: Cloudy (hydration level: 20, coldness level: 17) will ignite Torch
d
Cloudy (hydration level: 20, coldness level: 17) removed FlintAndSteel from the backpack

Here are the items that Cloudy has in the Backpack (3.50 / 10.00kg):
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
OilLantern (Feuerhand) has weight of 1.50 kg - to reduce the cold
Torch (Pine wood) has weight of 1.00 kg - to light the surrounding

Here are the items that we have on campsite:
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.
Bedroll (KAMUI) has weight of 7.00 kg - to rest.
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.

a: Cloudy (hydration level: 20, coldness level: 17) will pack Bedroll to the backpack
b: Cloudy (hydration level: 20, coldness level: 17) will pack Bedroll to the backpack
c: Cloudy (hydration level: 20, coldness level: 17) will pack FlintAndSteel to the backpack
d: Cloudy (hydration level: 20, coldness level: 17) will unpack Bottle from the backpack
e: Cloudy (hydration level: 20, coldness level: 17) will unpack OilLantern from the backpack
f: Cloudy (hydration level: 20, coldness level: 17) will unpack Torch from the backpack
c
Cloudy (hydration level: 20, coldness level: 17) packed FlintAndSteel to the backpack