Main Features

- Player Storage
- Simple Commands
- Localization

Player Storage

BaseIO provides an easy way to store data for the players. To use this feature just create a new class extending `com.unitedworldminers.BaseIO.storage.PlayerStorage`. This will be your storage class. Every player will have an instance of this class. You have to implement `void onNewUser()`, this method will be called everytime BaseIO has to create a new user. With `getUUID()` you can always retrieve the players UUID. `onNewUser` allows to throw a `NewInstanceException` when you don't want to create a storage for that user. But be warned, when retrieving userdata you will have to check for *null* storages.

> BaseIO saves your storage based on your modID! Changing the modID will result in data loss!
> The data itself on the other hand won't be deleted. You can try to modify the storage for that, but there will be no support for this.

To now save some values you just create your fields in your storage class. Every field in that class will be stored, if you don't want this you can give the `transistent` attribute to that field.

An example storage might be:

```
public class MyStorage extends PlayerStorage {
    boolean aBool;
    String aString;
    transient String bString;

    @Override
    protected void onNewUser() throws NewInstanceException {
        aString = "UUID: " + getUUID();
        bString = "This value won't be saved";
    }
}
```

To get a player's storage instance you just use one of the methods of the `PlayerIO` class.

- `get(String modID, UUID uuid)` is the fastest way to get your storage. *ModID* is the id you defined in your *@Mod* annotation. Use this method for cases running multiple times per second (e.g. tick events).
- `get(UUID uuid)` will retrieve your modID automatically, which is more expensive. Use this method only for user-input purposes (e.g. in commands).

- `get(String name)` will retrieve the player's UUID by the name and uses the method above next.
- `get(ICommandSender sender)` is a simplified method for general purpose, but only EntityPlayer is allowed (*FakePlayer* is a subclass of *EntityPlayer*).

Additionally you can retrieve all storages of a player by calling `ensuredGet(UUID uuid)`.
Additionally you can retrieve all your storages by calling `getAll(Class clazz)` with *clazz* as your storage class.

> PlayerIO will save your storage automatically.
> To manually save or reload the data, use PlayerIO.save() or PlayerIO.load().

An example working with your storage:

```
ICommandSender caller;
MyStorage storage = PlayerIO.get(caller);
if (storage.aBool) {
    storage.bString = "Permission granted.";
    PlayerIO.save();
}
```

Simple Commands

BaseIO simplifies command creation with the class `SimpleCommand`. Instead of implementing `ICommand` you extend `SimpleCommand`.

> If you don't like the simplifications made by this feature just override the original method from ICommand.

This is what SimpleCommand can do for you:

- ## Simple command names

  `getCommandName()` and `getCommandAliases()` are combined in `commandNames()` where the first command name is the actual command name and the others are aliases. Example:

- ```
  public List<String> commandNames() {
      //return Collections.singletonList("name1");
      return Arrays.asList("name1", "name2", "name3");
  }
  ```

- ## localized/default command usage

  Provide a localization key in `commandUsage()` and it will be localized as described in Localization. Provide an invalid or no (*null*) localization key and SimpleCommand creates a default command usage for you based on the arguments.

- ## redundancy removal

  `isUsernameIndex` and `compareTo` are removed.

- ## Options

  You can define command options in `getOptions()`. Options are arguments starting with a hyphen that can be used at any position of the arguments. Additionally options might have arguments attached to themselves. When implemented, getOptions() must return a map with the option string as the key and a string array of potential option arguments, the elements describing the option argument at its index. If found, the options will be cut out of the argument array and added to the option set that will be passed to the run method you have to implement in your command.
  ```

Example:

```
@Override
public void run(MinecraftServer server, ICommandSender sender, String[] args, Map<String, String[]> option
    System.out.println("Args: " + String.join(" ", args));
    for (Map.Entry<String, String[]> option: options.entrySet()) {
        System.out.println("Option " + option.getKey() + ": " + (option.getValue == null ? "null" : String.join(" ",
    }
}

@NotNull
@Override
protected Map<String, String[]> getOptions() {
    //return Utils.mapKeys("opt1", "opt2", "opt3");
    return Utils.map("opt1", new String[]{"oa1", "oa2"}, "opt2", null, "opt3", null);
}
```

If now a player uses the command `/mycommand arg1 arg2 -opt1 A B arg3 -opt3` the output would be:

> Args: arg1 arg2 arg3
> Option opt1: A B
> Option opt3: null

Tab Completion

Tab completions can be defined by the return string in `tabCompletions()`. The syntax allowes highly configurable tab completions:

1. The completions for each argument are seperated by a space.
2. The different completions for an argument are seperated by a pipe ("|").
3. Conditions for a completion are written in braces directly before it.
4. If you have multiple completions for a condition you can seperate them by a comma instead of a pipe.

Conditions are arguments that have to be typed anywhere before the completion. With that you are able to implement completely different tab completions based on the arguments the user used. See the example for usage.

Examples:

```
"comp1|comp2|comp3" => The completions "comp1","comp2" and "comp3" for the first argument

"compA compB1|compB2 compC" =>
 "compA" for the first, "compB1" and "compB2" for the second and "compC" for the third argument

"compA1|compA2 (compA1)compB1,compB2|compB3" =>
 "compA1" and "compA2" for the first argument, "compB3" for the second argument, "compB1" and "compB2
```

Variables are available for tab completions. They are defined by a leading "%" and are also customizable. Build-in variables are:

- `%d`: Dimensions (can be resolved to the dimension id by WorldUtils.
- `%p`: Players
- `%null`: No completions (explicitly)

Additionally you can define your own variable by implementing `customTabCompletions`. This method will be called if the tab completion contains an unknown variable and requests a list of completions for this variable. The currently requested variable is found in the parameter `tag` without the leading `%`. Other parameters are the command sender, the currently typed arguments and the completions up to that point (**read-only**).

Example:

```
@Override
protected String tabCompletions() {
   return "compA|%p|%var1";
}

@Override
protected Collection<String> customTabCompletions(String tag, List<String> current, ICommandSender sen
    if (tag.equals("var1")) {
        return Collections.singletonList("result1");
    }
    return null;
}
```

Localization

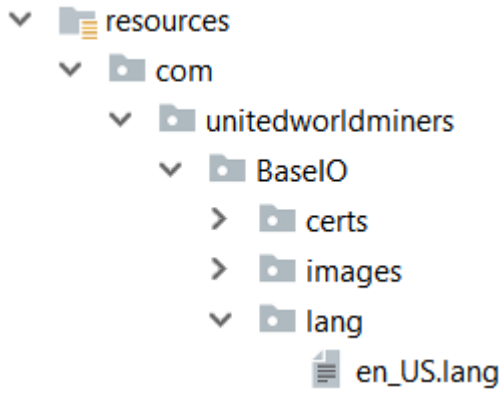
> It is highly recommended that you use this feature as it also allows server admins to customize your strings.

Localization is a feature of MessageUtils.

Setup

To start working with BaseIO's localization feature just call `MessageUtils.setupTranslations(langResourcePath)`. For the most basic setup use `null` as parameter and you're done. Now server admins can add translations for your strings as they wish.

If you want to add your own translations to your mod fill `langResourcePath` with a string path pointing to a folder in your mod resources.



For example BaseIO has `MessageUtils.setupTranslations("com/unitedworldminers/BaseIO/lang")`.
It is recommended to call this method in your init method (`FMLInitializationEvent`).

Creating localization mappings

> The default language file is `en_US.lang`. This file will be used if you aren't providing the correct language.

> You can use almost any string as a key, but it's recommended to abstract them.

The mapping files itself are the same as the ones from Minecraft: `<key>=<mapping>`, an example file can be found on the right sidebar. This files have to be named by the language tag given by Minecraft and must have the extension `.lang`.

In the mappings you can use arguments following the same format of the java formatter (`String.format`). These arguments can be injected in the method calls using localizations.

If you are a modder you put the files in your resource folder that you pointed at in `setupTranslations` .
If you are a server admin you can put the files into `config/lang/<modid>/` .

> You can reload the mappings with `/baseio reloadMessages <modid>` .

Using localizations

Your localizations can be used in

- `MessageUtils.messageToSender()`
- Simplecommand's `commandUsage()`
- Manually: `MessageUtils.getTranslation()`