bcf

bcf Developers

Table of contents

Introduction

- What is bcf 4
- Features 4
- Quickstart 8

How it works

- Example 11
- Parsers 12

Annotations

- Common 14
  - @Arg 14
  - @Command 15
  - @Default 16
  - @Error 17
- Bukkit 18
  - @Permission 18

Parsers

- Common 20
  - Common Parameters 20
    - default 20
    - required 21
    - suppress 21
    - switch 22
  - Literal 22
  - @Double 23
  - @Int 24
    - min 24
    - max 24
  - @String 24
- Bukkit 25
  - @Player 25
    - mode 25

Contributing

- Contributing 27
  - New ideas or Bug Reports 27
  - Contributing Code 27
  - Contributing Documentation 27
    - Requirements 27
    - Dev Environment 28
    - Change PDF Theme 28

Introduction

What is bcf¶

`bcf` is a Command Management tool suitable for use in Bukkit/Spigot/PaperMC, though it may support other platforms in the near future.

It allows one to easily provide full command completion for all your commands as well as automatically resolve and marshal input from the user so your command gets the data it is expecting to get. For example if your command is expecting a player name then available players will be provided during command completion and your command method will receive the actual player object, or the command sender will receive an error explaining what the error is.

This library is inspired by aikar's Annotation Command Framework (ACF) *(https://github.com/aikar/commands)* which used before I decided on the crazy idea of writing my own to support some extra features I felt I desperately needed for some reason. The name `bcf` is used in recognition of that.

Features¶

- Define your commands by simply extending a BaseCommand derived class (like `BukkitCommand`) and annotating it with a `@Command` to define it, with any aliases separated by a `|`.

  Example

  ```
  @Command("mycmd|my|m")
  public class myCommand extends BukkitCommand {
     ...
  }
  ```

All of `/mycmd`, `/my`, `/m` are valid commands with the latter 2 designated as aliases.

- Annotate your method with `@Arg` to define what arguments it is expecting. This is a string of arguments that may consume 0 or more words from a users input both to provide command completion as well as to fully resolve and pass objects to the method.

Example

```
@Arg("give|g @player(required=true, default=%self, mode=online)")
public void onGive(CommandSender sender, Player player) {
   ...
}
```

- Multiple annotations can be used and each will be checked in turn.

Example

```
@Arg("give|g @player(required=true, default=%self, mode=online)")
@Arg("sudo give @player(required=true, default=%self, mode=online)")
public void onGive(CommandSender sender, Player player) {
   ...
}
```

- Nearly all annotations can be added to your class to apply to all methods (and child classes).

Example

```
@Command("do")
@Arg("for @player(required=true, mode=offline)")
class MyCommand extends BukkitCommand {

    @Arg("tpto @world")
    public void doThis(CommandSender, Player player, World world) {
       ...
    }
}
```

To reach doThis the command is: `/do for <playername> tpto <worldname>`

- Your class can extend another command class to inherit any of its settings. For example a 3rd party plugin could extend your command class to add sub-commands under your own.

Example

```
@Arg("sudo")
class MySubCommand extends MyCommand {

    @Arg("kill")
    public void killPlayer(CommandSender, Player player) {
       ...
    }
}
```

To execute killPlayer the full command now is: `/do for <playername> sudo kill`

- Create command aliases by adding a `@Command` annotation to a derived class. This allows shortcut commands to jump straight to a class. For example instead of `/command view playername` you can have `/cv playername` as an alias
- When a command needs to send an error (for example a parameter is not valid) the class will look for a method annotated with `@Error`. If it fails to find one it will check all its parent classes until it reaches the default. This allows you to override how errors are handled.

Example

```
@Arg("sudo")
class MySubCommand extends MyCommand {
    @Error
    void onError(CommandSender sender, String message) {
        sender.spigot().sendMessage(
                new ComponentBuilder(message).color(ChatColor.GREEN).create()
        );
    }

    ...
}
```

Will show error in green when it occurs under the `sudo` subcommand otherwise will show the default red.

- When no command is reached a method annotated with `@Default` is looked for to handle things. If no method is found then every parent class is checked until it reaches the default which outputs "Invalid Command". This can be used to provide more help.

Example

```
@Arg("sudo")
class MySubCommand extends MyCommand {
    @Default
    void onDefault(CommandSender sender) {
        sender.spigot().sendMessage(
                new ComponentBuilder("A totally unhelpful message.").color(Cl
        );
    }
```

```
    ...
}
```

Will show "A totally unhelpful message" if no method is matched in this class (or its children), otherwise it will show the default message.

- Add permission requirements by annotating your class or methods with `@Permission`. The command sender must either be console or have at least one of the permissions at each level to proceed otherwise both command completion and execution will be ignored as if the arguments did not exist.

Example

```
@Permission("mycmd.admin")
@Permission("mycmd.command.sudo")
@Arg("sudo")
class MySubCommand extends MyCommand {

    @Permission("mycmd.admin")
    @Arg("kill")
    public void killPlayer(CommandSender, Player player) {
       ...
    }

    @Arg("tickle")
    public void ticklePlayer(CommandSender, Player player) {
       ...
    }

    ...
}
```

The command sender neds to have `mycmd.admin` to be able to use either the `kill` or `tickle` subcommands. They need either `mycmd.admin` or `mycmd.command.sudo` to access the `tickle` command. Classes extending this one will respect the Permissons on their parent class.

- Support both required and optional positional parameters. A required parameter must either have a default or must have valid input provided. Optional parameters with no default and no input will be set to null.

Example

```
@Arg("cmd1 @player(required=false)")
public void cmd1(CommandSender, Player player) {
   ...
}

@Arg("cmd2 @player(required=true, default=%self, mode=offline)")
public void cmd2(CommandSender, OfflinePlayer player) {
```

```
    ...
}
```

cmd1 does not require a player which will resolve to null if no input provided. cmd2 requires a player name but if none is provided then the value %self is provided which will resolve to the command sender if they are a player or return an error if not.

- As well as supporting positional parameters we support named parameters called switches. To pass a switch the command sender uses -<switchname> <value(s)> and full command completion is provided. A switch becomes available in the chain of arguments once it is reached and it can have multiple aliases. A non required switch with no default value will be resolved to null. A required switch must be resolved by input sometime after the point it is defined otherwise the command will be rejected. Designating a switch parameter means it is no longer treated as positional.

Example

```
@Arg("transfer @int(min=1,max=3) @player(switch='from|f') @player(switch='to|
public void doTransfer(CommandSender, Player from, Player to, Itemstack item)
    ...
}
```

Valid commands would be:

- /mycmd transfer 3 -from Player1 -to Player2 diamond_sword
- /mycmd transfer 3 diamond_sword -t Player2 -f Player1

This would not be valid:

- /mycmd transfer -from Player1 -to Player2 3 diamond_sword

Quickstart¶

1. Add the following Maven repository to your pom.xml

```
<!-- Bundabrg's Repo -->
<repository>
    <id>bundabrg-repo</id>
    <url>https://repo.worldguard.com.au/repository/maven-public<
    <releases>
        <enabled>true</enabled>
    </releases>
    <snapshots>
        <enabled>false</enabled>
    </snapshots>
</repository>
```

2. Add the following dependency to your `pom.xml`

```
<dependency>
    <groupId>au.com.grieve.bcf</groupId>
    <artifactId>bukkit</artifactId>
    <version>1.2.0</version>
    <scope>provided</scope>
</dependency>
```

3. Shade the library into your own code by adding in your `pom.xml`

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.2.3</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                       <goal>shade</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <relocations>
                    <relocation>
                        <pattern>au.com.grieve.bcf</pattern>
                        <shadedPattern>${project.groupId}.${proj
                    </relocation>
                </relocations>
            </configuration>
        </plugin>
    </plugins>
</build>
```

4. Create a command class that extends BukkitCommand

```
@Command("mycmd")
public class MainCommand extends BukkitCommand {

    @Arg("list")
    public void doList(CommandSender sender) {
```

```
            sender.spigot().sendMessage(
                new ComponentBuilder("Reached List").color(ChatColor
            );
        }
    }
```

5. Create a new CommandManager in your plugin, passing your plugin as a parameter and register your commandclass.

```
// Setup Command Manager
bcf = new BukkitCommandManager(this);

// Register Commands
bcf.registerCommand(MainCommand.class);
```

6. You should now be able to use `/mycmd list` in-game.

Last update:

How it works

Example¶

When someone is typing a command or executing it we can see it as being a list of words. Lets look at the following fictional command:

```
/sudo tp notch ~ ~25 ~ -world nether
```

The command can be viewed as a list of words and can be broken down into:

- `sudo` - The main command itself
- `tp` - A sub-command
- `notch` - A players name
- `~ ~25 ~` - a Location
- `-world nether` - a named parameter defining the world

We call the full command string the `input`.

We could define our class as follows to handle this particular command (ignore the @Arg for the moment)

```
@Command("sudo")
class MainCommand extends BukkitCommand {
    @Arg("tp @world(switch=world, required=false) @player(required=t
    public void doTeleport(CommandSender sender, World world, Player
        .
    }
}
```

This shows the following interesting things:

1. The parameters the method receives are fully resolved objects and not just the input strings. This means the method does not have to deal with all the complexity of providing command completion nor need to perform any validation. If a command is not valid then it won't reach the method and may instead match another more specific method or it will raise an error.

2. `sudo` and `tp` are not passed to the method and are only interesting so far as part of deciding which method to call.

3. There is an extra parameter to the method `sender` which represents the command sender (player or console)
4. The argument `player` consumes a single word from the input, `notch`, and returns a Player object representing an online player.
5. The argument `location` consumes 3 words from the input, `~ ~25 ~`, and returns a Location object
6. The argument `world` is special as it is designated as a `switch` in the @Arg and thus is a named parameter. A named parameter can appear anywhere in the input as long as it comes at or after the point it is defined (in @Arg its defined right after `tp`) and if the input contains its name prefixed with a `-`. So this parameter consumes `-world nether` and will return a World object.

What happens whilst someone is entering the command?

1. After typing `/sudo` an auto-complete of `tp` will show.
2. After entering `tp` an auto-complete list of online players is shown and is filtered as the player types. A `-world` is also shown as an option when there is no partial player name entered.
3. After filling in the player name, if it is valid, a location autocomplete will show using the players existing position if needed. A `-world` is also shown as an option when there is no partial location entered.
4. After filling in the location, if it is valid, a `-world` is shown. The player can choose to hit enter to not fill it in (as its not a required parameter) or they can type `-world` in which case a list of worlds are provided as part of the auto-complete. Note that they could have entered this in at any time after entering `tp`.

Parsers¶

The magic happens through the use of a `Parser` that understands both how to consume 0 or more words from the input into a fully resolved object but also how to auto-complete partial inputs.

The annotation `@Arg` defines a chain of parsers to be used when consuming input. Only once all the required parsers are consumed will a method be invoked.

A parser is typically defined by adding an `@` and its name to the `@Arg` string, and each additional parser is separated by a space.

There is one special parser of note called the `LiteralParser` and it does not start with an `@`. You will note in the example above we have `tp` in arg. This is a `LiteralParser` and it will consume its own name from the input without passing anything to the method. A

`LiteralParser` can also provide multiple aliases by separating each alias with a `|`. So in our example above we could have defined the sub-command as `teleport|tp`. In this case it would accept either as input and will show both during autocomplete.

A parser can have parameters that define its behaviour. These are passed by adding a list of value=key pairs after the parser name inside `(` and `)` brackets. In the example above `@world` and `@player` both have defined parameters whereas the `@location` parser does not.

Many parsers will have common parameters. For example all parsers have a `required` boolean to determine if it is a required argument. They also all have a `default` which is provided when no input is provided for that parser.

A special parameter called `switch` is used on the `@world` parser. This turns the `@world` into a named parameter instead of a positional parameter which means it can be provided in the input anywhere after it is defined as long as the input contains the special sequence `-` followed by any of the names defined in the switch. For `@world` only a single name was given so entering `-world` would then trigger that parser to consume the next input(s). Multiple aliases can be defined by separating each alias with a `|`. An example would be `@world(switch=world|w)` and would allow either `-w` or `-world`.

Some parsers have unique parameters. For example the `@player` parser used in the example has a `mode` parameter that defines if the player should currently be online or can be any player (offline). This affects what is shown in auto-complete as well as if the input is valid.

A Parser will typically consume 1 word from the input but it is possible for a Parser to consume no words as it may rely on data from a previous Parser or on something entirely unrelated to input, and as in the case of the `@location` parser in the example it can consume multiple words from the input.

Finally a BukkitCommand derived class will always provide a CommandSender parameter as a method's first parameter. This will be either the console or the player who is executing the command.

Last update:

Annotations

In Java an annotation is something that can be added onto a Class, Method, Variable or even a method parameter.

Most annotations we use can be added to the Class itself, the Method, or both and they are called its `Target`. Generally an annotation on a class will affect all the methods inside the class as well as any in any derived classes.

Most annotation can also be used multiple times. For example you can add `@Arg` more than one to the same method and when completing/executing a command each `@Arg` will be checked in turn. Each annotation has its own rules to define what multiples of itself does.

Common¶

These annotations are available for all execution environments.

@Arg¶

Target: `Class`, `Method`

Value: `String`

Multiple: allowed

Provides a list of `Parsers` that will consume input and can provide command completions for partial input. Each parser starts with an `@` prefixed to its name along with optional parameters defined inside braces after the parser definition. Any bare strings are treated with as `LiteralParser` and will accept its own name(s) without providing any output to the invoked method.

If a command is executed then the winning `@Arg` will then invoke the method with fully resolved objects, each parser consuming the input and optionally returning an object to be provided as a parameter to the method.

When consuming input, the `@Arg` on a class (if any) will first process the input, with the remaining input then going to any defined on its `Methods`. Any Classes that extend the class will also receive the remaining input in which case they go through the same steps.

Multiple `@Arg` will be checked in turn and can be thought of as multiple choices.

More details about Parsers will be provided in Parsers

Example

```
@Command("mycmd")
class MainCommand extends BukkitCommand {
    @Arg("cmd1")
    public void cmd1(CommandSender sender) {
        ...
    }
}

@Arg("sub1")
@Arg("sub2")
class SubCommand extends MainCommand {
    @Arg("cmd2")
    @Arg("cmd3")
    public void cmd2(CommandSender sender) {
        ...
    }
}
```

This shows two classes, one extending the other, and multiple @Arg at several points.

The following are all valid commands:

- `/mycmd cmd1`
- `/mycmd sub1 cmd2`
- `/mycmd sub1 cmd3`
- `/mycmd sub2 cmd2`
- `/mycmd sub2 cmd3`

@Command¶

Target: `Class`

Value: `String`

Multiple: allowed

Defines a top level command and if possible will be registered with the execution environment. Aliases to the same command can be added by separating the value with `|`. This can be used to create top-level commands that jump straight to a deeply nested sub command.

Multiple `@Commands` will register multiple top level commands.

Example

```
@Command("mycmd|mc")
class MainCommand extends BukkitCommand {
    @Arg("cmd1")
    public void cmd1(CommandSender sender) {
        ...
    }
```

```
}

@Command("qc")
class SubCommand extends MainCommand {
    @Arg("cmd3")
    public void cmd2(CommandSender sender) {
        ...
    }
}
```

Here 2 top level commands are registered. `/mycmd` and `/qc`. An alias `/mc` is also available for `/mycmd`.

Valid commands are:

- `/mycmd cmd1`
- `/mc cmd1`
- `/mycmd cmd3`
- `/qc cmd3`

@Default¶

Target: `Method`

Value: `None`

Multiple: not allowed

Signifies what method to call when needing to invoke a `Default` method for a Class and any of its child classes (unless overridden in that class with its own `@Default`)

When processing input if we do not find any winning method for a class then a `Default` method will be executed. If none are found in a class then its parent class will be searched until it ends up with the built in `Default` that shows a "Invalid Command" message.

This can be used to provide help for unknown commands or can be used to redefine what is shown on no match.

The method must not take any additional parameters apart from those provided by default.

Example

```
@Command("mycmd")
class MainCommand extends BukkitCommand {
    @Arg("cmd1")
    public void cmd1(CommandSender sender) {
        ...
    }

    @Default
    public void onDefault(CommandSender sender) {
        sender.spigot().sendMessage(
                new ComponentBuilder("Some help").color(ChatColor.YELLOW).create()
```

```
        );
    }

}

@Arg("sub")
class SubCommand extends MainCommand {
    @Arg("cmd2")
    public void cmd2(CommandSender sender) {
        ...
    }

    @Default
    public void onDefault(CommandSender sender) {
        sender.spigot().sendMessage(
                new ComponentBuilder("Sub Help").color(ChatColor.RED).create()
        );
    }
}
```

When attempting to execute the invalid command: `/mycmd sub unknown` a message "Sub Help" in red will be shown.

When attempting to execute the invalid command: `/mycmd unknown` a message "Some help" in yellow will be shown.

@Error¶

Target: `Method`

Value: `None`

Multiple: not allowed

Signifies what method to call when needing to invoke an `Error` method for any errors in a Class and any of its child classes (unless overridden in that class with its own `@Error`)

When processing input if an error is encountered it will be passed to a `Error` method. If none are found in a class then its parent will be searched until it ends with the built in `Error` that will show the error in red.

This can be used to customize how errors are handled.

The method must take a string argument containing the error message.

Example

```
@Command("mycmd")
class MainCommand extends BukkitCommand {
    @Arg("cmd1 @int(max=3)")
    public void cmd1(CommandSender sender) {
        ...
```

```
    }

    @Error
    public void onError(CommandSender sender, String message) {
        sender.spigot().sendMessage(
                new ComponentBuilder(message).color(ChatColor.YELLOW).create()
        );
    }

}

@Arg("sub")
class SubCommand extends MainCommand {
    @Arg("cmd2 @int(max=3)")
    public void cmd2(CommandSender sender) {
        ...
    }

    @Error
    public void onError(CommandSender sender, String message) {
        sender.spigot().sendMessage(
                new ComponentBuilder(message).color(ChatColor.GREEN).create()
        );
    }
}
```

When attempting to execute the command: `/mycmd sub cmd2 10` an error message "Number must be smaller or equal to 3" will be shown in green.

When attempting to execute the invalid command: `/mycmd cmd1 10` a message "Number must be smaller or equal to 3" in yellow will be shown.

Bukkit¶

These annotations are available for Bukkit environments.

@Permission¶

Target: `Class`, `Method`

Value: `String`

Multiple: allowed

Provides the permissions the command sender must have for a class or method to be considered.

When multiple `@Permission` is found on the same Target then it will accept any of those permissions. This can be thought of as an OR condition.

Example

```
@Command("mycmd")
class MainCommand extends BukkitCommand {
    @Arg("cmd1")
    public void cmd1(CommandSender sender) {
        ...
    }

    @Permission("mycmd.perm1")
    @Arg("cmd2")
        public void cmd1(CommandSender sender) {
            ...
        }
}

@Permission("mycmd.perm1")
@Permission("mycmd.perm2")
class SubCommand extends MainCommand {
    @Arg("cmd3")
    public void cmd2(CommandSender sender) {
        ...
    }

    @Arg("cmd4")
    @Permission("mycmd.perm2")
    public void cmd2(CommandSender sender) {
        ...
    }
}
```

A user with `mycmd.perm1` will be able to access the following commands:

- `/mycmd cmd1`
- `/mycmd cmd2`
- `/mycmd cmd3`

A user with `mycmd.perm2` will be able to access the followinig commands:

- `/mycmd cmd1`
- `/mycmd cmd3`
- `/mycmd cmd4`

Last update:

Parsers

A `Parser` is something with a name that can consume 0 or more words from the input and can produce 0 or 1 objects to be passed as a parameter to a method. It can also use partial input to provide command completion.

Parsers are provided in a space separated `@Arg` annotation string to the Class and/or Method.

The format of the parser definition in the `@Arg` string is:

```
@parsername(key=value, ...) @nextparser...
```

Where:

- `@parsername` - the name of the parser to user. If it does not start with @ then it will be treated as a `Literal` Parser.
- `(key=value, ...)` - optional parameters can be passed to the parser to define its behaviour. If no parameters are required then the braces can be left off as well.

Common¶

These parsers are available for all execution environments

Common Parameters¶

There are some parameters that are common to most Parsers and will be listed here.

default¶

Provide a default value if none is provided through input. Note that any input at all will stop the default being provided and invalid input will correctly show an error.

Example

```
@Command("mycmd")
public class MainCommand extends BukkitCommand {

    @Arg("cmd1 @int(default=3)")
    public void myCmd1(CommandSender sender, Integer myNum) {
        ...
    }
}
```

The command `/mycmd cmd1` will provide 3 to `myNum` variable.

The command `/mycmd cmd1 10` will provide 10 to `myNum` variable.

The command `/mycmd cmd1 aaa` will display the error "Invalid Number: a"

required¶

If set to true will require a value to be provided either through input or through a `default` parameter.

If set to false (default) then missing input without a `default` will return a null object to the method.

Example

```
@Command("mycmd")
public class MainCommand extends BukkitCommand {

    @Arg("cmd1 @int(required=true, default=3)")
    public void myCmd1(CommandSender sender, Integer myNum) {
        ...
    }

    @Arg("cmd2 @int(required=true)")
    public void myCmd2(CommandSender sender, Integer myNum) {
        ...
    }

    @Arg("cmd3 @int(required=false)")
    public void myCmd3(CommandSender sender, Integer myNum) {
        ...
    }

}
```

The command `/mycmd cmd1` will succeed and provide 3 for variable `myNum`

The command `/mycmd cmd2` will show an error "A number is required"

The command `/mycmd cmd3` will succeed and provide null for variable `myNum`

suppress¶

If set to true then this parser will not return any object to a method but otherwise will behave the same

Example

```
@Command("mycmd")
public class MainCommand extends BukkitCommand {
```

```
    @Arg("cmd1 @int(suppress=true, required=true)")
    public void myCmd1(CommandSender sender) {
        ...
    }
}
```

The command `/mycmd cmd1 5` will execute the method with no additional parameters.

The command `/mycmd cmd1` will return a required parameter error due to the `required=true` parameter.

switch¶

The presence of this parameter will make a Parser into a named parameter instead of a positional one. It lists the name(s) of the switch separated by a `|`.

From the point it is defined it will consume input only if the input has a `-` with one of the names provided in which case the next word(s) of the input will go towards this parser. The returned object will be provided to the method in the order it is defined in the `@Arg` string.

Full command completion is provided for both the name(s) of the switch as well as its values.

Example

```
@Command("mycmd")
public class MainCommand extends BukkitCommand {

    @Arg("cmd1 @int(switch=test|t, required=false) param1 param2 param3")
    public void myCmd1(CommandSender sender, Integer myNum) {
        ...
    }
}
```

The following are all valid:

- `/mycmd cmd1 -test 3 param1 param2 param3`
- `/mycmd cmd1 param1 -test 3 param2 param3`
- `/mycmd cmd1 param1 param2 -test 3 param 3`
- `/mycmd cmd1 param1 param2 param3 -test 3`
- `/mycmd cmd1 param1 param2 param3`

All but the last will return 3 to the variable `myNum`.

The last command will return `null` to the variable `myNum`.

Literal¶

Consumes: 1

Returns: `String` (if `suppress` is `false`)

Completions: `yes`

The simplest Parser is the Literal Parser. This one does not have a special name but rather is used whenever a string is detected instead of the name of a parser prefixed with @.

The Literal parser will use its name as input and has `suppress` set to true by default so will not normally provide any parameter to the method. Multiple options can be provided by separating the names with `|`.

Command completion will show all the options provided.

Example

```
@Command("mycmd")
public class MainCommand extends BukkitCommand {

    @Arg("cmd1|c1 param1 param2|p2(suppress=false) param3")
    public void myCmd1(CommandSender sender, String p2) {
        ...
    }
}
```

The following are all valid commands:

- `/mycmd cmd1 param1 param2 param3`
- `/mycmd c1 param1 param2 param3`
- `/mycmd cmd1 param1 p2 param3`
- `/mycmd c1 param1 p2 param3`

The method parameter `p2` will be filled in with either `param2` or `p2`.

@Double¶

Consumes: 1

Returns: `Double`

Completions: `no`

This Parser will try to read a floating point number from input and will return it as a `Double`

Example

```
@Command("mycmd")
public class MainCommand extends BukkitCommand {

    @Arg("cmd1 @double")
    public void myCmd1(CommandSender sender, Double p1) {
        ...
```

```
    }
}
```

The command: `/mycmd cmd1 1.5` will provided the method parmeter `p1` with a Double with value `1.5`

The command `/mycmd cmd1 aaa` will return an error.

@Int¶

Consumes: 1

Returns: `Integer`

Completions: `sometimes`

This Parser will try to read an integer from input and will return it as an `Integer`.

If a `max` parameter is provided then completion will show up to 20 numbers between `min` and `max`. If `min` is not defined but `max` is then `min` will be considered to be 0 for completion only but will not affect execution.

min¶

Set the minimum value accepted. Defaults to no minimum.

max¶

Set the maximum value accepted. Defaults to no maximum.

Example

```
@Command("mycmd")
public class MainCommand extends BukkitCommand {

    @Arg("cmd1 @int(min=4,max=10)")
    public void myCmd1(CommandSender sender, Integer p1) {
        ...
    }
}
```

The command: `/mycmd cmd1 6` will provided the method parmeter `p1` with a Integer with value `6`

The command `/mycmd cmd1 100` will return an error.

The command `/mycmd cmd1 aaa` will return an error.

@String¶

Consumes: 1

Returns: `String`

Completions: `no`

This Parser will consume a single word from input and return it as a `String` object.

No completions will be provided.

Example

```
@Command("mycmd")
public class MainCommand extends BukkitCommand {

    @Arg("cmd1 @string @string")
    public void myCmd1(CommandSender sender, String p1, String p2) {
        ...
    }
}
```

The command: `/mycmd cmd1 foo bar` will provided the String `foo` for method parmaeter `p1` and `bar` for method parameter `p2`

Bukkit¶

These parsers are available for the Bukkit execution environment

@Player¶

Consumes: 1

Returns: `Player`, `OfflinePlayer`

Completions: `yes`

Reads in a player name and will return either a `Player` or `OfflinePlayer` depending on the setting of the `mode` parameter.

A value of `%self` will refer to the command sender. This is useful to provide as a `default`. When the command sender is the console then an error will be returned "When console a player name is required".

mode¶

Can be either `online` or `offline` (default).

When `mode` is `online` then completion will only show currently online players and when executing will only accept a player that is online. Returns a `Player` object to the method.

When `mode` is `offline` then completion will show all players, online and offlline, and when executing will validate that the player exists. Returns an `OfflinePlayer` object to the method.

Example

```
@Command("mycmd")
public class MainCommand extends BukkitCommand {

    @Arg("cmd1 @player(default=%self, mode=online")
    public void myCmd1(CommandSender sender, Player player) {
        ...
    }

    @Arg("cmd1 @player(mode=offline")
    public void myCmd1(CommandSender sender, OfflinePlayer player) {
        ...
    }
}
```

Last update:

Contributing¶

Here are some ways that you can help contribute to this project.

New ideas or Bug Reports¶

Need something? Found a bug? Or just have a brilliant idea? Head to the Issues *(https://github.com/Bundabrg/bcf/issues)* and create new one.

Contributing Code¶

If you know Java then take a look at open issues and create a pull request.

Do the following to build the code:

```
git clone https://github.com/Bundabrg/bcf
cd bcf
mvn clean package
```

Contributing Documentation¶

If you can help improve the documentation it would be highly appreciated. Have a look under the `docs` folder for the existing documentation.

The documentation is built using `mkdocs`. You can set up a hot-build dev environment that will auto-refresh changes as they are made.

Requirements¶

- python3
- pip3
- npm (only if changing themes)

Install dependencies by running:

```
pip3 install -r requirements.txt
```

Dev Environment¶

To start a http document server on `http://127.0.0.1:8000` execute:

```
mkdocs serve
```

Change PDF Theme¶

Edit the PDF theme under `docs/theme/pdf`. Rebuild by doing the following:

```
cd docs/theme/pdf
npm install
npm run build-compressed
```

This will update `pdf.css` under `docs/css/pdf.css`. Rebuilding the docs will now use the new theme.

Last update: