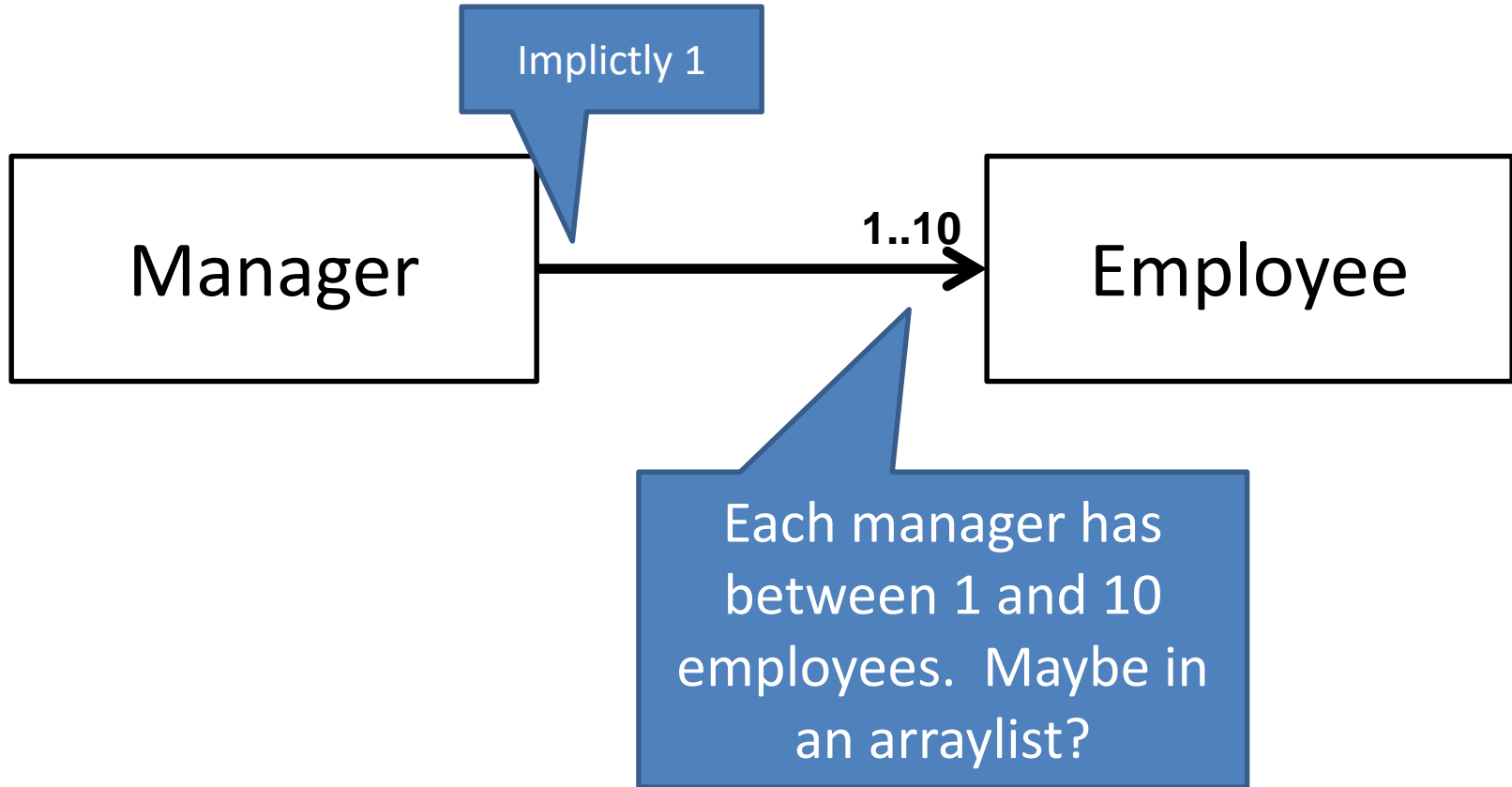


# CSSE 220

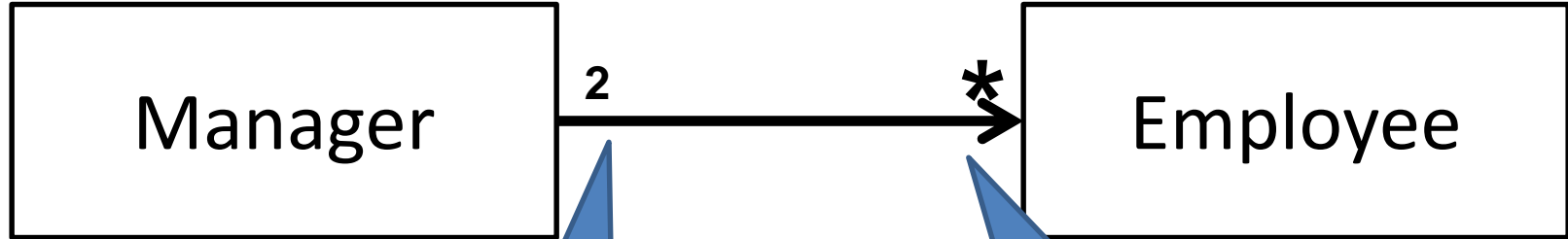
Object-Oriented Design  
Files & Exceptions

Check out *FilesAndExceptions* from SVN

# New UML Notation: Cardinality



# More Cardinality

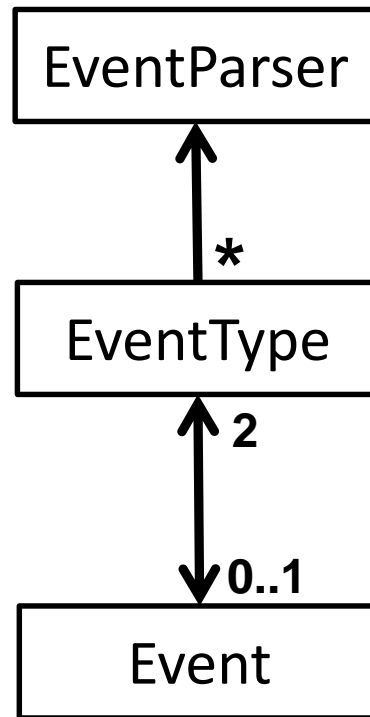


Every employee has exactly 2 managers. Note that this can be used even if there is no reference from Employee to Manager

Managers have any number of employees.

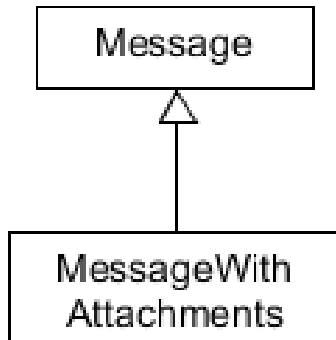
The \* means “zero to infinity” – any arbitrary number. You can also occasionally see something like 4..\* to mean 4 or more.

# What does this diagram mean?

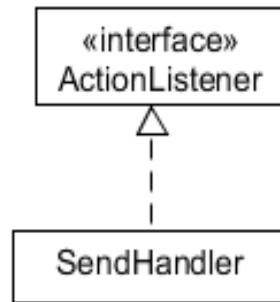


# Summary of UML Class Diagram Arrows

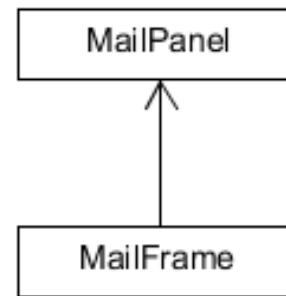
Inheritance  
(is-a)



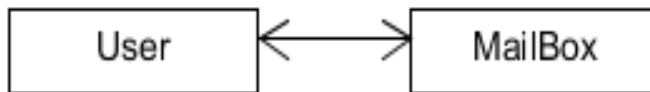
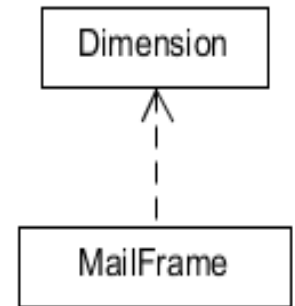
Interface  
Implementation  
(is-a)



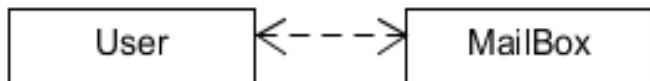
Association  
(has-a-field)



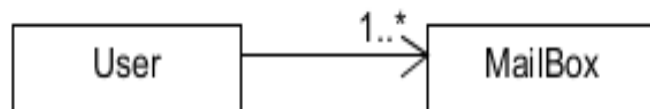
Dependency  
(depends-on)



Two-way Association



Two-Way Dependency



Cardinality  
(one-to-one, one-to-many)  
One-to-many is shown on left

A practical technique

# **OBJECT-ORIENTED DESIGN**

# Object-Oriented Design

- We won't use full-scale, formal methodologies
  - Those are in later SE courses
- We will practice a common object-oriented design technique using **CRC Cards**
- Like any design technique,  
**the key to success is practice**

# Key Steps in Our Design Process

- 1. Discover classes** based on requirements
- 2. Determine responsibilities** of each class
- 3. Describe relationships** between classes



# Discover Classes Based on Requirements

- Brainstorm a list of possible classes
  - Anything that might work
  - No squashing

# Discover Classes Based on Requirements

- Prompts:
  - Look for **nouns**
  - Multiple objects are often created from each class
    - So look for **plural concepts**
  - Consider how much detail a concept requires:
    - A lot? Probably a class
    - Not much? Perhaps a primitive type
- Don't expect to find them all → add as needed

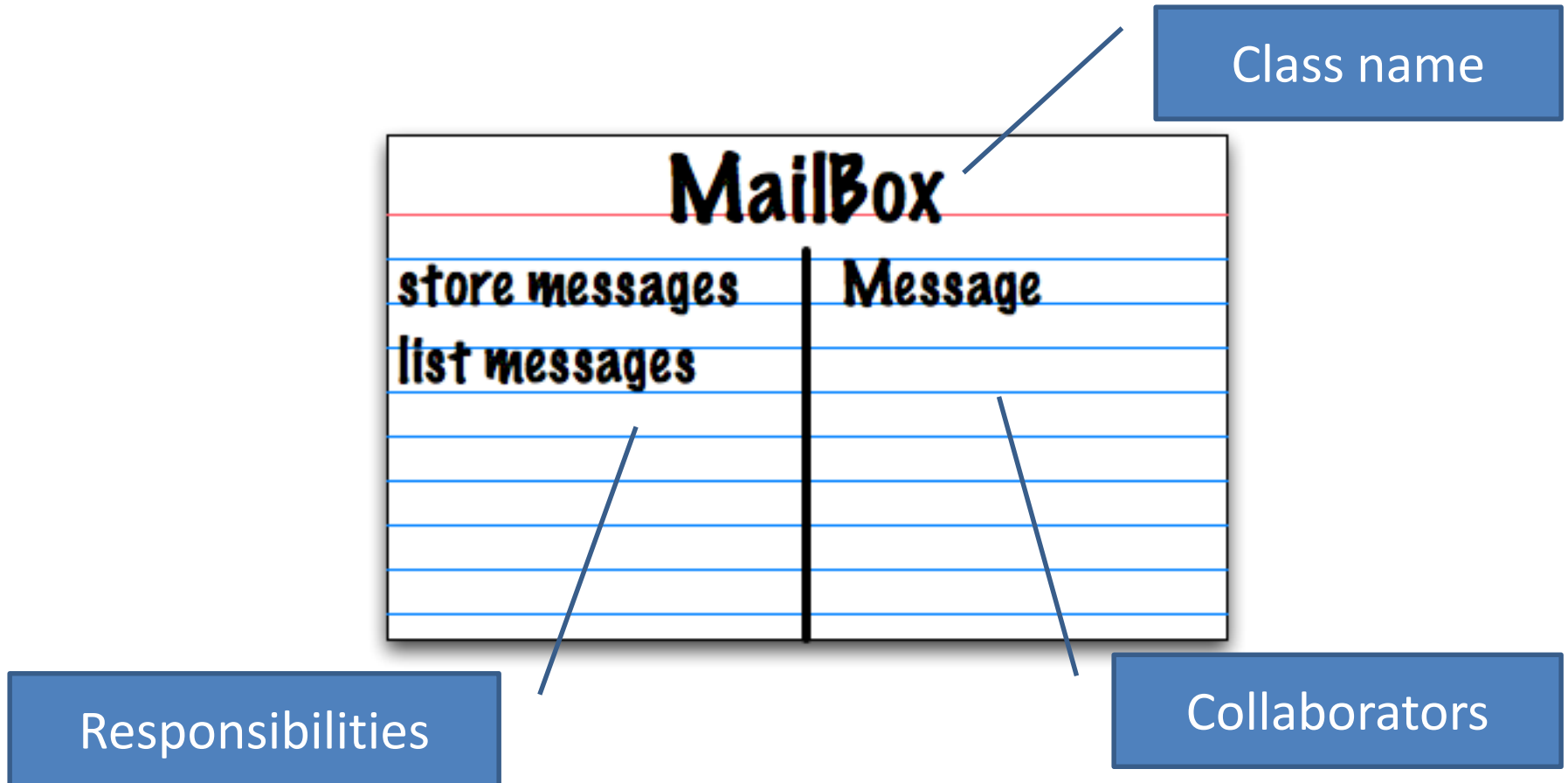
Tired of hearing this yet?

# Determine Responsibilities

- Look for **verbs** in the requirements to identify **responsibilities** of your system
- Which class handles the responsibility?
- Can use **CRC Cards** to discover this:
  - **Classes**
  - **Responsibilities**
  - **Collaborators**

# CRC Cards

- Use one index card per class



# CRC Card Tips

- **Spread the cards out** on a table
  - Or sticky notes on a whiteboard instead of cards
- **Use a “token”** to keep your place
  - A quarter or a magnet
- **Focus on high-level responsibilities**
  - Some say < 3 per card
- **Keep it informal**
  - Rewrite cards if they get too sloppy
  - Tear up mistakes
  - Shuffle cards around to keep “friends” together

# CRC Card Technique

1. Pick a **responsibility** of the program
2. Pick a **class** to carry out that responsibility
  - Add that responsibility to the class's card
3. Can that class carry out the responsibility by itself?
  - Yes → Return to step 1
  - No →
    - Decide which classes should help
    - List them as **collaborators** on the first card

Use the email messaging system description given on today's handout to create CRC cards.

# Describe the Relationships

- Classes usually are related to their collaborators
- Draw a UML class diagram showing how
- Common relationships:
  - **Inheritance**: only when subclass **is a** special case
  - **Dependency**: transient use of a type, usually for method parameters, **“has a” temporarily**
  - **Association**: **“has-a” field** of the specified type

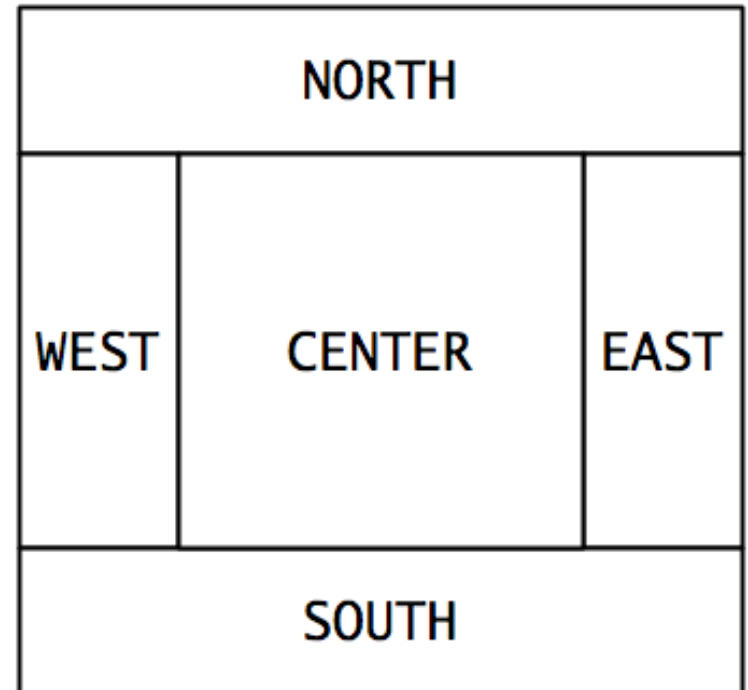
When JFrame's and JPanel's defaults just don't cut it.

# **SOME NOTES ON LAYOUT MANAGERS**



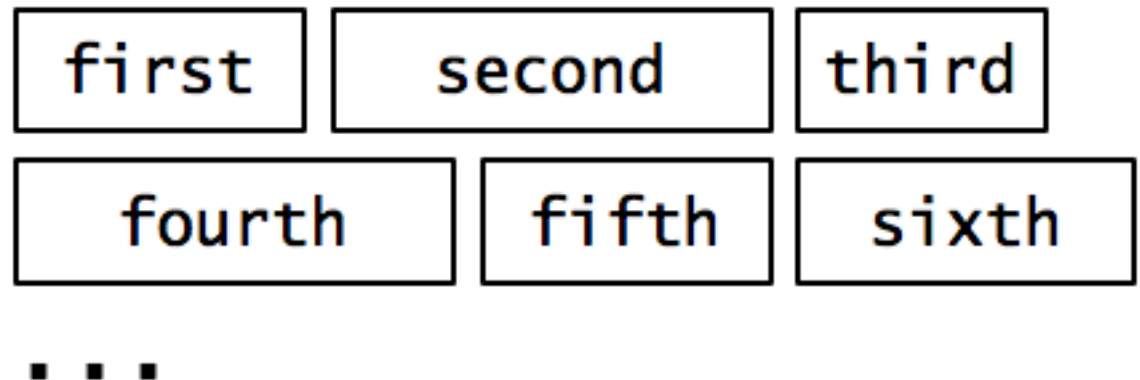
# Recall: How many components can a JFrame show by default?

- Answer: 5
- We use the two-argument version of **add**:
- `JPanel p = new JPanel();`  
`frame.add(p, BorderLayout.SOUTH);`
- **JFrame**'s default **LayoutManager** is a **BorderLayout**
- **LayoutManager** instances tell the Java library how to arrange components
- **BorderLayout** uses up to five components



# Recall: How many components can a JPanel show by default?

- Answer: arbitrarily many
- Additional components are added in a line
- **JPanel's default LayoutManager is a FlowLayout**



# Setting the Layout Manager

- We can set the layout manager of a JPanel manually if we don't like the default:

```
JPanel panel = new JPanel();  
panel.setLayout(new GridLayout(4,3));  
panel.add(new JButton("1"));  
panel.add(new JButton("2"));  
panel.add(new JButton("3"));  
panel.add(new JButton("4"));  
// ...  
panel.add(new JButton("0"));  
panel.add(new JButton("#"));  
frame.add(panel);
```



# Lots of Layout Managers

- A **LayoutManager** determines how components are laid out within a container
  - **BorderLayout**. When adding a component, you specify center, north, south, east, or west for its location. (Default for a JFrame.)
  - **FlowLayout**: Components are placed left to right. When a row is filled, start a new one. (Default for a JPanel.)
  - **GridLayout**. All components same size, placed into a 2D grid.
  - Many others are available, including **BoxLayout**, **CardLayout**, **GridBagLayout**, **GroupLayout**
  - If you use **null** for the **LayoutManager**, then you must specify every location using coordinates
    - More control, but it doesn't resize automatically

Reading & writing files

When the unexpected happens

# **FILES AND EXCEPTIONS**

# Review of Anonymous Classes

- Look at GameOfLifeWithIO
  - GameOfLife constructor has 2 listeners, two *local anonymous* class
  - ButtonPanel constructor has 3 listeners which are *local anonymous* classes
- Feel free to use as examples for your project

# File I/O: Key Pieces

- Input: **File** and **Scanner**
- Output: **PrintWriter** and **println**
- ☺ Be kind to your OS: **close()** all files
- Letting users choose: **JFileChooser** and **File**
- Expect the unexpected: **Exception** handling
- Refer to examples when you need to...

# Exception – What, When, Why, How?

- What:
  - Used to signal that something in the code has gone wrong
- When:
  - An error has occurred that cannot be handled in the current code
- Why:
  - Breaks the execution flow and passes exception up the stack



# Exception – How?

- Throwing an exception:  
`throw new EOFException("Missing column");`
- Handling (catching) an exception:  

```
try {  
    //code that could throw an exception  
}  
catch (ExceptionType ex) {  
    //code to handle exception  
}
```
- When caught you can:
  - Recover from the error OR exit gracefully

# What happens when no exception is thrown?

```
Scanner inScanner;  
try {  
    inScanner =  
        new Scanner(new File("test.txt"));  
    //code for reading lines  
} catch (IOException ex) {  
    JOptionPane.  
        showMessageDialog("File not found.");  
} finally {  
    inScanner.close();  
}
```

The diagram illustrates the execution flow of the provided Java code. It features four callout boxes with arrows pointing to specific lines of code:

- A blue arrow points to the line `inScanner = new Scanner(new File("test.txt"));` with the text "If this line is successful".
- A blue arrow points to the line `//code for reading lines` with the text "Code continues on".
- A red arrow points to the line `JOptionPane.showMessageDialog("File not found.");` with the text "The catch never executes".
- A blue arrow points to the line `inScanner.close();` with the text "This runs after code in try completes".

# What happens when exception is thrown?

```
Scanner inScanner;  
try {  
    inScanner =  
        new Scanner(new File("test.txt"));  
    //code for reading lines  
} catch (IOException ex) {  
    JOptionPane.  
        showMessageDialog("File not found.");  
} finally {  
    inScanner.close();  
}
```

If this line throws exception

Code after exception never executes

This is the next line executed

After catch is executed, this runs

# When exception is not handled?

```
public String readData(String filename)
```

```
    throws IOException {
```

```
        Scanner inScanner =
```

If this line throws exception

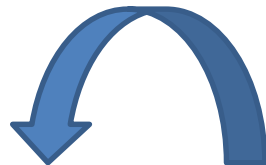
```
            new Scanner(new File(filename));
```

```
        //code for reading lines
```

```
        inScanner.close();
```

Code does not execute,  
Method breaks immediately

```
    }
```



main -> readAllFiles -> readData

If unhandled, exception bounces to method that called it, then up the chain.

# A Checkered Past

- Java has two sorts of **exceptions**
  - 1. Checked exceptions:** compiler checks that calling code isn't ignoring the problem
    - Used for **expected** problems
  - 1. Unchecked exceptions:** compiler lets us ignore these if we want
    - Used for fatal or avoidable problems
    - Are subclasses of RuntimeException or Error

# A Tale of Two Choices

Dealing with **checked** exceptions

## 1. Can **propagate** the exception

- Just declare that our method will pass any exceptions along...

```
public void loadGameState() throws IOException
```

- Used when our code isn't able to rectify the problem

## 1. Can **handle** the exception

- Used when our code can rectify the problem

# Handling Exceptions

- Use try-catch statement:

```
try {  
    // potentially “exceptional” code  
} catch (ExceptionType var) {  
    // handle exception  
}
```

Can repeat this part for as many different exception types as you need.

- Related, try-finally for clean up:

```
try {  
    // code that requires “clean up”  
} finally {  
    // runs even if exception occurred  
}
```