Agenda

- Java I/O
- Polymorphism

- Reading:
  Chapter 9 - Inheritance and Interfaces

Java File I/O

Opening a File

- When we construct an input stream or output stream object, the JVM associates the file name, standard input stream, or standard output stream with an object. This is **opening a file.**
- When we are finished with a file, we optionally call the *close* method to release the resources associated with the file.
- In contrast, the standard input stream (*System.in*), the standard output stream (*System.out*), and the standard error stream *(System.err*) are open automatically when the program begins. They are intended to stay open and should not be closed.
- Calling the close method is optional. When the program finishes executing, all the resources of any unclosed files are released.
- It is good practice to call the *close* method.

File Types

- Java supports two types of files:
  - text files: data is stored as characters
  - binary files: data is stored as raw bytes
- The type of a file is determined by the classes used to write to the file.
- To read an existing file, you must know the **file's type** in order to select the appropriate classes for reading the file.

Text Files

Reading Text Files

- A text file is treated as a stream of characters.
- ***FileReader*** is designed to read character files.
- A *FileReader* object does not use buffering, so we will use the ***BufferedReader*** class and the *readLine* method to read more efficiently from a text file.

ReadTextFile.java

```
import java.io.*; // import java.io
public class ReadTextFile {
    public static void main( String [] args )  throws IOException{
        FileReader fr = new FileReader("dataFile.txt");
        BufferedReader br = new BufferedReader(fr);
        String stringRead = br.readLine();
        while( stringRead != null ) { // EOF
            System.out.println(stringRead);
            stringRead = br.readLine();
        }
        br.close();
}
```

Writing to Text Files

- Several situations can exist:
  - the file does not exist
  - the file exists and we want to replace the current contents
  - the file exists and we want to append to the current contents
- We specify whether we want to replace the contents or append to the current contents when we construct our *FileWriter* object.

WriteTextFile.java

```
import java.io.*; // import java.io
public class writeTextFile{
    public static void main( String [] args ) throws IOException {
        FileWriter fw = new FileWriter( "output.txt", false);
        BufferedWriter bw = new BufferedWriter(fw);
        bw.write( "I never saw a purple cow,");
        bw.newLine();
        bw.write( "I never hope to see one;");
        bw.newLine();
        bw.write( "But I can tell you, anyhow,\n");
        bw.write( "I'd rather see than be one!\n");
        bw.close();
    }
}
```

false means if a file by the specified name already exists, the file will be overwritten.

AppendTextFile.java

```
import java.io.*;
public class AppendTextFile{
    public static void main( String [] args) throws IOException {
        FileWriter fw = new FileWriter( "output.txt", true);
        BufferedWriter bw = new BufferedWriter(fw);
        bw.write( "I never saw a purple cow," );
        bw.newLine( );
        bw.write( "I never hope to see one;" );
        bw.newLine( );
        bw.write( "But I can tell you, anyhow,\n" );
        bw.write( "I'd rather see than be one!\n" );
        bw.close( );
    }
}
```

true means to append to an existing file

Polymorphism

Polymorphism

- A code expression can invoke different methods depending on the types of objects being manipulated
- Example: function overloading like method `min()` from `java.lang.Math`
  - The method invoked depends on the types of the actual arguments

```
int a, b, c;
double x, y, z;
…
c = min(a, b); // invokes integer min()
z = min(x, y); // invokes double min
```

Polymorphism

- Two types of polymorphism
  - Syntactic polymorphism—Java can determine which method to invoke at compile time
    - Efficient
    - Easy to understand and analyze
    - Also known as primitive polymorphism
    - The matching between the method call and the correct method is called "binding"
    - The binding can determined during compile time, so it is static (early) binding

In previous example in last page:

c = min(a, b);   *// invokes integer min(), int a, b, c*

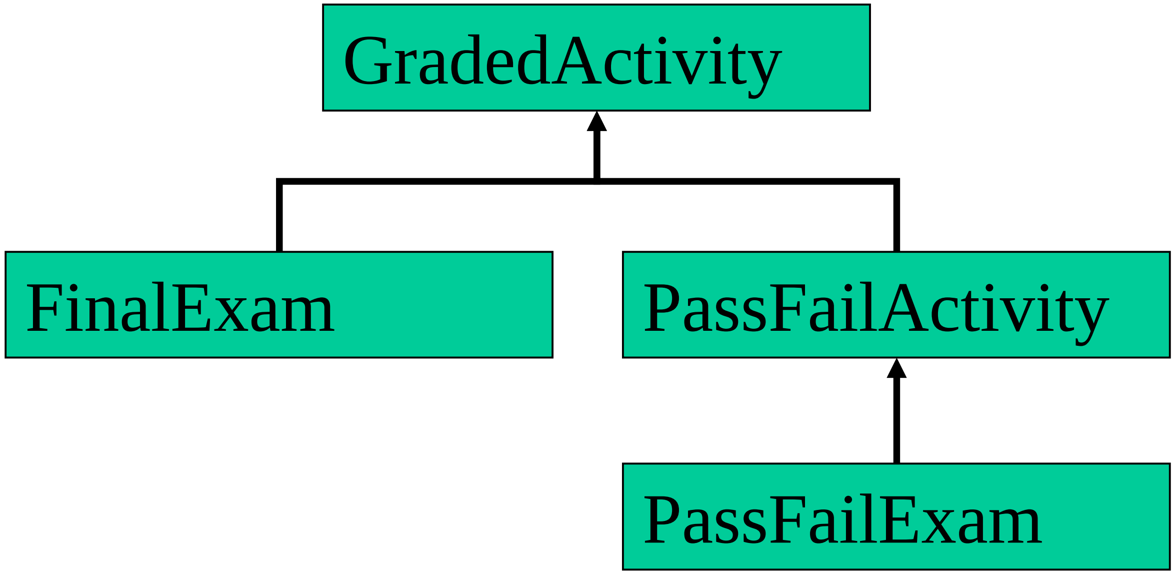
z = min(x, y);   *// invokes double min(), double x, y, z*

Polymorphism (Cont'd)

- Pure polymorphism
  - the method to invoke can only be determined at execution time
  - The binding is dynamic (late) binding
  - When we use a superclass reference variable to reference a subclass object, the binding process is a dynamic binding

Example: Chains of Inheritance

- Classes often are depicted graphically in a *class hierarchy*.
- A class hierarchy shows the inheritance relationships between classes.

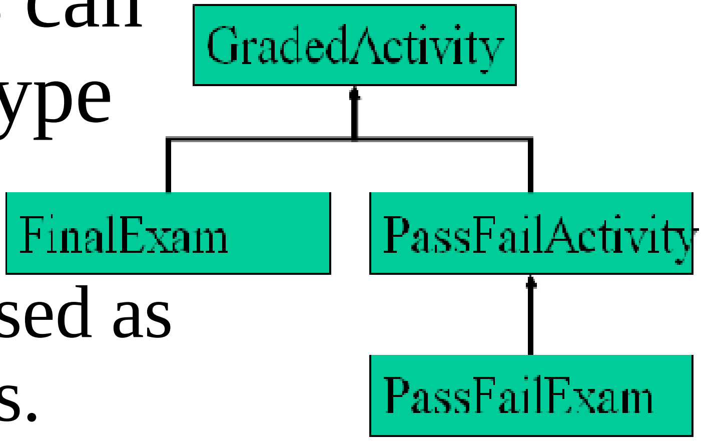


Polymorphism

- A reference variable can reference objects of classes that are derived from the variable's class.

```
GradedActivity exam;
exam = new GradedActivity();
```

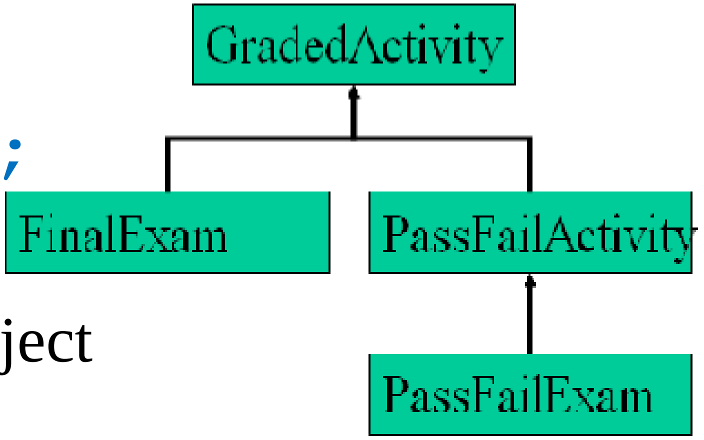
- A reference variable of the superclass can **also** be used to reference a **subclass** type object (Next slide coming up!)
  - The `GradedActivity` class is also used as the superclass for the `FinalExam` class.
  - An object of the `FinalExam` class ***is a*** `GradedActivity` object.



Polymorphism

- Example

```
GradedActivity exam;
exam = new FinalExam(50, 7);
```



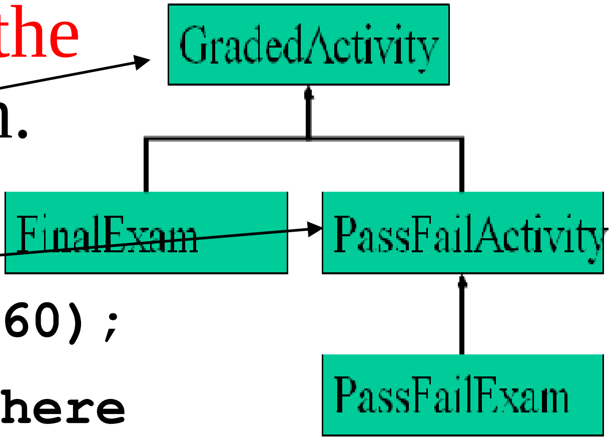
- The second statement creates a `FinalExam` object and **stores the object's address in the `exam` variable**.

- This is an example of polymorphism.

- The term *polymorphism* means the ability to take many forms.

Polymorphism and Dynamic Binding

- If the object of the subclass has overridden a method in the superclass:
  - If the variable makes a call to that method, the subclass's version of the method will be run.



```
GradedActivity exam = new PassFailActivity(60);
exam.setScore(70);  //the subclass version here
System.out.println(exam.getGrade());
```

- Java performs *dynamic binding* or *late binding* when a variable contains a polymorphic reference.

- The Java Virtual Machine determines at runtime which method to call, depending on the type of object that the variable references.

Basis of Polymorphism (Ingredients)

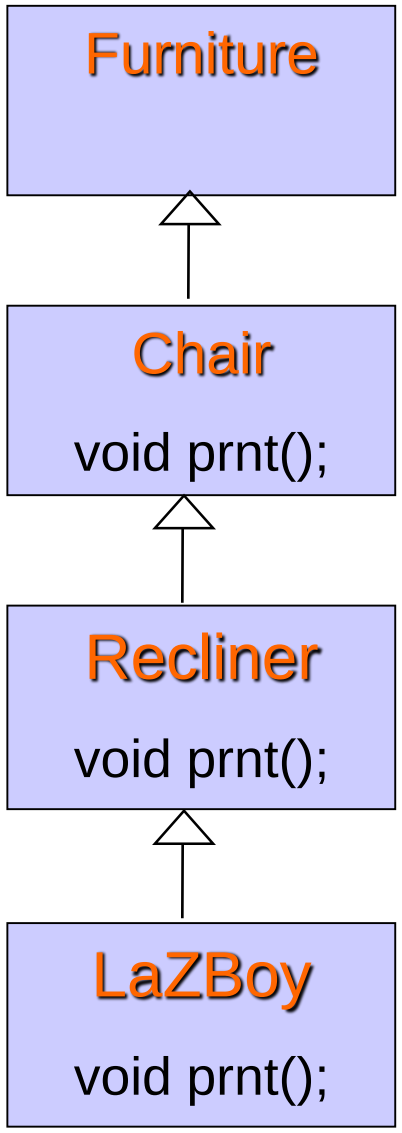
1. Inheritance
2. Method overriding
3. Polymorphic assignment // SuperClassVariable = SubclassObject;
4. Polymorphic methods
   - In Java, all methods are polymorphic.
   - That is, choice of method depends on the object.

Polymorphic assignment

```
SuperClassVariable = SubclassObject;
```

Example: Polymorphism

```
abstract class Furniture {
    public int numlegs;
}

abstract class Chair extends Furniture {
    public String fabric;
    abstract void prnt();
}

class Recliner extends Chair {
    void prnt() {
       System.out.println(“I’m a recliner”);
    }
}

class LaZBoy extends Recliner {
    void prnt() {
        System.out.println(“I’m a lazboy”);
    }
}
```



What is the output?

```
Chair cha;
cha = new LaZBoy();
cha.prnt();
```

```
Furniture furn;
furn = new Recliner();
furn.prnt();
```

```
Furniture furn;
furn = new LaZBoy();
furn.prnt();
```

Members use compile-time binding

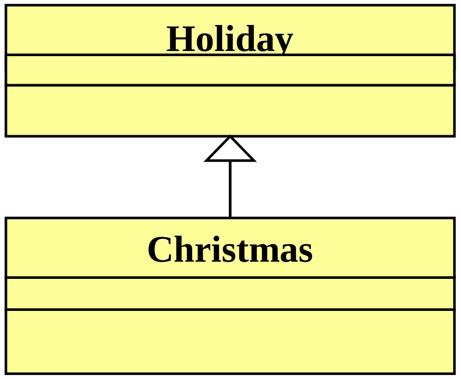
```
class Base{
        int X=99;
        public void prnt(){
                System.out.println("Base");
        }
}

class Rtype extends Base{
        int X=-1;
        public void prnt(){
                System.out.println(“Rtype");
        }
}
```

What is the output?

```
Base b=new Rtype();
System.out.println(b.X);
b.prnt();
```

Polymorphism Recap

- Polymorphism: A polymorphic reference v is declared as class C, but unless C is final or base type, v can refer to an object of class C or to an object of *any class derived* from C.
- A method call v.<method_name>(<args>) invokes a method of the class of an object referred to by v (not necessarily C):



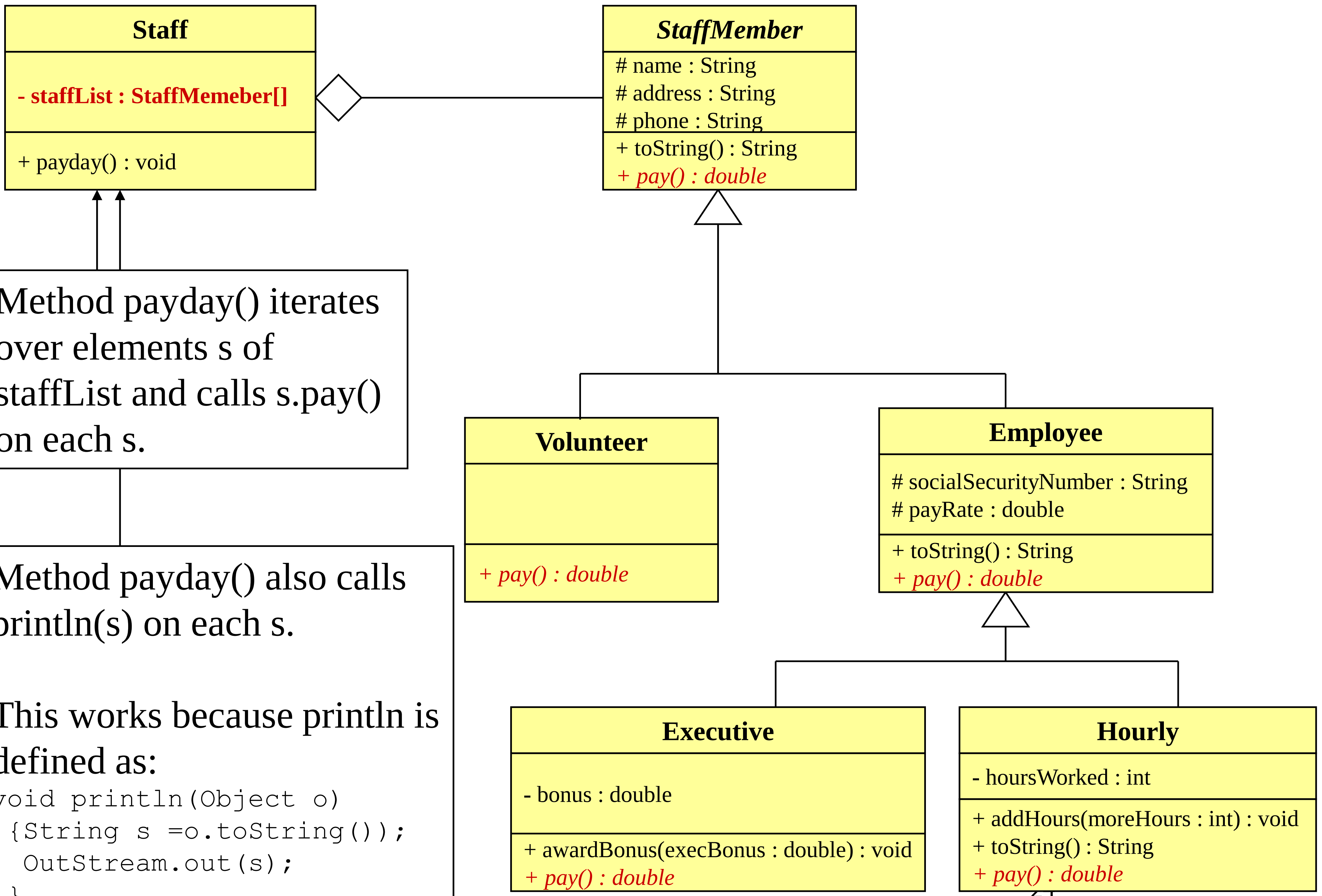
Ex1:
```
Holiday day =
        new Christmas();
day.celebrate();
…
```

Ex2:
```
void process(Holiday day)
 { …
   day.celebrate();
   …  }
Christmas day = ...;
process(day)
```

- A very common usage of polymorphism: If classes C1, C2, ..., Cn are all derived from C, define an array A of elements of C.

  The entries A[i] can then refer to objects of classes C1, ..., Cn.

The pay-roll of a firm



Method payday() iterates over elements s of staffList and calls s.pay() on each s.

Method payday() also calls println(s) on each s.

This works because println is defined as:

```
void println(Object o)
 {String s =o.toString());
  OutStream.out(s);
 }
```

Recall on your lab 5

```
// Lab5 tester class
public class Lab5Tester {
    public static void main(String[] args) {
        CsusStudent student = new CsusStudent("John Doe", 123, "123 
Somewhere", "415-555-1212", "johndoe@somewhere.com");
        Csc20Student csc20Student = new Csc20Student("John Doe", 123, "123 
Somewhere", "415-555-1212", "johndoe@somewhere.com",true,15);
        System.out.println(student + "\n");
        System.out.println(csc20Student + "\n");
    }
}
```

Show the output of the following program.

Homework

```
abstract class Furniture { abstract void prnt();}
class Recliner extends Furniture {
    void prnt() { System.out.println("I'm a recliner");}
}
class LaZBoy extends Recliner {
    void prnt() { System.out.println("I'm a lazboy");}
}
public class furnitureTest2 {
    public static void main(String[] args) {
        Furniture [] A = { new Recliner(), new Recliner(), new LaZBoy()};
        for (int i=0; i<3; ++i)
            A[i].prnt();
    }
}
```

Homework

Show the output of the following program.

```
class A { int x = 1; }
class B extends A { }
class C extends B { int x = 2;}
public class classTest {
    public static void main(String[] args) {
        A w = new A(); System.out.println(w.x);
        B u = new B(); System.out.println(u.x);
        C v = new C(); System.out.println(v.x);
        A [] a = { new A(), new B(), new C()};
        for (int i=0; i<3; ++i)
            System.out.println(a[i].x);
    }
}
```