What are the language design decisions Java made differently from Scheme to explain why this is so long?
What Java language design decisions make this so long?

```java
public class HelloWorld {
    public static void main (String [] args) {
        System.out.println("Hello!");
    }
}
```

**Question 1**

1. Static typing: big win for truthiness
2. All procedures must be inside a class
3. Default visibility is not public (package protected)
4. Use squiggly brackets to denote blocks, semi-colons to end statements
5. Not providing a special, convenient way to print output, but requiring an I/O object and invoking a method

---

**Exam 1**

Score Distribution

<table>
<thead>
<tr>
<th>Score Range</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>90-100</td>
<td>9</td>
</tr>
<tr>
<td>80-89</td>
<td>5</td>
</tr>
<tr>
<td>70-79</td>
<td>0</td>
</tr>
<tr>
<td>&lt;=70</td>
<td>4</td>
</tr>
</tbody>
</table>

I will re-ask (in slightly different from) at least some of the questions on Exam 1 on Exam 2.

**Recap: Substitution Principle Summary**

- **Param Types**: Psub ≥ Psuper (contravariant) for inputs
- **Preconditions**: pre_sub ⇒ pre_super for inputs
- **Result Type**: Rsub ≤ Rsuper (covariant) for outputs
- **Postconditions**: post_sub ⇒ post_super for outputs
- **Properties**: properties_sub ⇒ properties_super

These properties ensure code that is correct using an object of supertype is correct using an object of subtype.

---

**Substitution Principle**

What should the spec of the KillingBear class' `kill (Object o)` method be?

- **BlackBear**
  - `public boolean kill (Object o)`
  - // MODIFIES: this, o
  - // EFFECTS: If o is up a tree, climbs the tree, eats o, and returns true. Otherwise, if o is reachable, eats o. Otherwise, returns false.

- **Killer**

- **Climber**

- **KillingBear**

- **GrizzlyBear**
  - `public boolean kill (Object o)`
  - // MODIFIES: p, nearby trees, incidentals
  - // EFFECTS: If this can eat p, eat p & return true. Otherwise...

- **Bear**

*Is this the only way?*
Eiffel’s Rules

(Described in Bertrand Meyer paper for ps4)

Eiffel Rules

The types of the parameters in the subtype method may be subtypes of the supertype parameters.

How can Girl override set_roommate?

set_roommate (Girl g)
set_roommate (Boy b)

Opposite of substitution principle!

Eiffel and I Can’t Get Up?

s: skier; g: girl; b: boy;
s := g;
...
s.set_roommate (b);

Meyer’s paper is all about the contortions Eiffel needs to deal with non-substitutable subtypes

Substitution Principle vs. Eiffel

Substitution Principles vs. Java

Overloading and Overriding

• **Overriding**: replacing a supertype’s method in a subtype
  – Dynamic dispatch finds method of actual type
• **Overloading**: providing two methods with the same name but different parameter types
  – Statically select most specific matching method of apparent type
Overloading Example

public class Overloaded extends Object {
    public int tryMe (Object o) {
        return 17;
    }

    public int tryMe (String s) {
        return 23;
    }

    public boolean equals (String s) {
        return true;
    }
}

Overloading

public class Overloaded {
    public int tryMe (Object o) {
        return 17;
    }

    public int tryMe (String s) {
        return 23;
    }

    public boolean equals (String s) {
        return true;
    }
}

Overloading 2

public class Overwhelming {
    public int tryMe (Object o, String s) {
        return 17;
    }

    public int tryMe (String s, Object o) {
        return 23;
    }

    public static void main(String[] args) {
        Overwhelming over = new Overwhelming ();
        System.err.println (over.tryMe ("test1", "test2"));
    }
}

Overkill

- Overloading and overriding together can be overwhelming!
- Avoid overloading whenever possible: names are cheap and plentiful
- One place you can’t easily avoid it: constructors (they all have to have the same name)
  — But, can make static “factory” methods instead (this is usually better)

Use @Override annotations so compiler will check that you are actually overriding!

Java Buzzword Description

“A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.”

【Sun95】

Later in the course, we will discuss how well it satisfies these “buzzwords”.