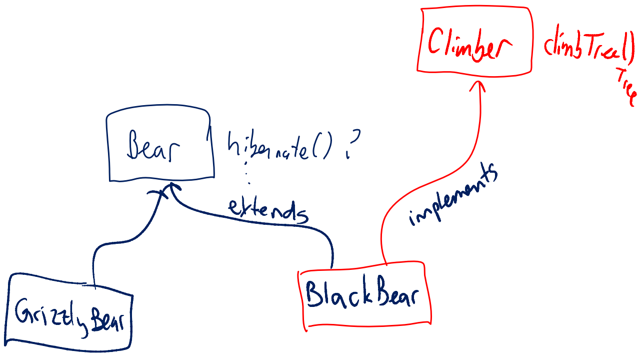
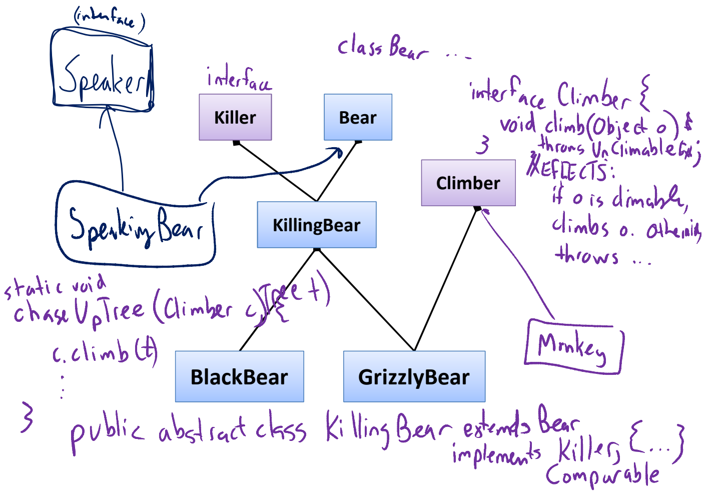
cs2220: Engineering Software
Class 13:
Behavioral Subtyping

Fall 2010
University of Virginia
David Evans

What's the difference between a **black bear** and a **grizzly bear?**





```
interface Climber {
   void climb(Object o) throws UnClimableEx;
   EFFECTS: if o is climbable, climbs o. otherwise throws ...
}
```

```
static void
chaseUpTree(Climber c, Tree t) {
   c.climb(t)
   :
}
```

```
public abstract class KillingBear extends Bear
                      implements Killer, Comparable {...}
```

Exam 1

Question 1

Give one concrete example where the Java programming language designers sacrificed expressiveness for truthiness. An ideal answer would illustrate your example with code snippets showing something that is difficult to express concisely because of the Java language's emphasis on truthiness.

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

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?

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

class variable
out

print ("Hello")

→1. Static manifest type
2. All code in a method
3. All methods in classes

Question 1

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

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

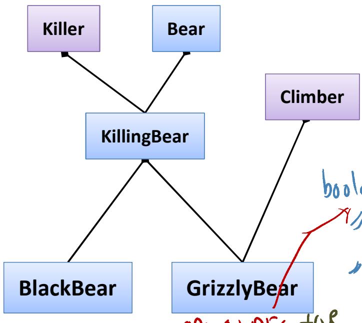
90-100	9
80-89	5
70-79	0
<=70	4

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*
Preconditions	pre_sub ~~⇒~~ ⇐ pre_super	for inputs
Result Type	Rsub ≤ Rsuper	*covariant*
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.



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

Person ≤ Object

boolean kill (Person p)
// MODIFIES: p, this, nearby trees, incidentals
// EFFECTS: If this can eat p, eat p & return false.
Otherwise ...

// REQUIRES: true
o is yummy to a ~~KillingBear~~ GrizzlyBear

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.

GrizzlyBear:
public boolean kill(Object o)
// REQUIRES: o is yummy to a GrizzlyBear
// MODIFIES: this, o, nearby trees and
//   incidentals
// EFFECTS: If o is up a tree, knocks down
//   the tree, eats o, and returns true.
//   Otherwise, returns false.

Substitution Principle

Is this the only way?

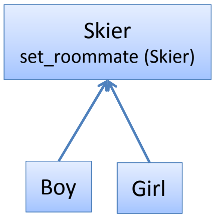
Vegetable ~~Person~~ brocolli;
KillingBear k;
:
((Grizzly) k).kill (brocolli);

Assume all Person objects are yummy to Grizzly B.

Eiffel's Rules

(Described in Bertrand Meyer paper for ps4)

Eiffel Rules

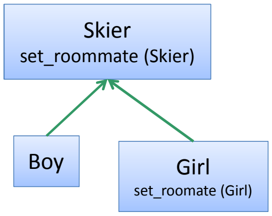


The types of the parameters in the subtype method may be subtypes of the supertype parameters.

How can **Girl** override **set_roomate**?
set_roommate (Girl g)
set_roommate (Boy b)

Opposite of substitution principle!

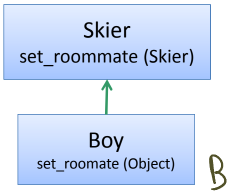
Eiffel and I Can't Get Up?



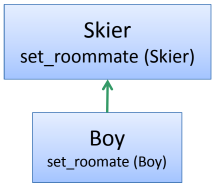
```
s: skier; g: girl; b: boy;
s := g;
...
s.set_roommate (b);
```

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

Substitution Principle vs. Eiffel

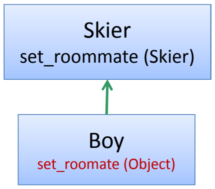


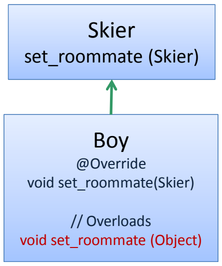
$B \leq A$



	Substitution Principle	**Eiffel**
Parameters	PB >= PA	$PB \subseteq PA$
Preconditions	pre_A ⇒ pre_B	$pre_B \Rightarrow pre_A$
Result	RB <= RA	$RB \subseteq RA$
Postconditions	post_B ⇒ post_A	$post_B \Rightarrow post_A$

Substitution Rules 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;
   }
}
```

public boolean equals (Object) is inherited from Object

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;
   }
}
```

```
static public void main (String args[]) {
   Overloaded over = new Overloaded ();
   System.err.println (over.tryMe (over));
   System.err.println (over.tryMe (new String ("test")));

   Object obj = new String ("test");
   System.err.println (over.tryMe (obj));
   System.err.println (over.equals (new String ("test")));
   System.err.println (over.equals (obj));

   Object obj2 = over;
   System.err.println (obj2.equals (new String ("test")));
}
```

```
17
23
17
true
false
false
```

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"));
  }
}
```

Compiler error:
The method tryMe(Object, String) is ambiguous for the type Overwhelming

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!

from Class 2...

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".