static public boolean moreElements(ArrayList<String> a, ArrayList<String> b) // REQUIRES: a and b are not null // MODIFIES: nothing // EFFECTS: Returns true if a has more elements than b. {
    return a.size() > b.size();
}

public static void main(String[] args) {
//            TODO Auto-generated method stub
    ArrayList<String> a; ArrayList<String> b; …
    a.add("Hello"); b.add("Ciao"); b.add("Goodbye");
    System.out.println("More elements: " + moreElements(a, b));
}

Dangers of Subtyping

public class SillyList<E> extends ArrayList<E> {
@override
    public int size() // REQUIRES: The alert level has reached DEFCON 5, all the missiles have // been targeted, and the President has issued a verified launch // command. // MODIFIES: Everything // EFFECTS: Launches the missiles. Returns the expected number of // elements in the object after all the computer's memory has been // destroyed by radiation. {
        launchMissiles();
        return 0; \n    }
}

public static void main(String[] args) {
    ArrayList<String> a = new SillyList<String>();
    ArrayList<String> b = new SillyList<String>();
    a.add("Hello"); b.add("Ciao");
    System.out.println("More elements: " + moreElements(a, b));
}

How can we solve this?

static public String pasteTogether(String a, String b) // REQUIRES: a and b are not null // EFFECTS: Returns a String that is a followed by b. {
    return a.concat(b);
}

Could pasteTogether launch the missiles?

public final class String extends Object implements Serializable, Comparable<String>, CharSequence {
    …
}
Reasoning with Subtyping

**Easy approach #1**: don’t allow subtyping!
Make all classes `final` (like `java.lang.String`)

**Easy approach #2**: give up on reasoning
Reason based on the apparent type specification and don’t make any claims about what happens with subtypes.

**Hard approach**: impose constraints on subtypes to allow reasoning

**Substitution Principle**: If `B` is a subtype of `A`, everywhere the code expects an `A`, a `B` can be used instead *and the program still satisfies its specification*

**Subtype Condition 1: Signature Rule**

We can use a subtype method where a supertype methods is expected:
- Subtype must implement all of the supertype methods
- Argument types must not be more restrictive
- Result type must be at least as restrictive
- Subtype method must not throw exceptions that are not subtypes of exceptions thrown by the supertype

**Subtype Condition 2: Methods Rule**

*Precondition* of the subtype method must be *weaker* than the precondition of the supertype method.

\[ m_A.pre \Rightarrow m_B.pre \]

*Postcondition* of the subtype method must be *stronger* than the postcondition of the supertype method.

\[ m_B.post \Rightarrow m_A.post \]

**Subtype Condition 3: Properties**

Subtypes must preserve all properties described in the overview specification of the supertype.
Properties Example

```java
public class StringSet
    // Overview: An immutable set of Strings.

public class MutStringSet extends StringSet
    // Overview: A mutable set of Strings.
```

MutStringSet cannot be a subtype of StringSet, since it does not satisfy property that once a StringSet object is created its value never changes.

Would it be okay for a subtype of a mutable type to be immutable?

Properties Example

```java
public class MutStringSet
    // Overview: A mutable set of Strings.

public class ImmutableStringSet extends MutStringSet
    // Overview: An immutable set of Strings.
```

ImmutableStringSet could be a subtype of MutStringSet according to the properties rule. ...but would be very difficult to satisfy the methods rule!

Substitution Principle Summary

- **Signatures**: subtype methods must be type correct in supertype callsites: result is a subtype (covariant), parameters are supertypes (contravariant)
- **Methods**: subtype preconditions must be weaker than supertype preconditions (covariant); subtype postconditions must be stronger than supertype postconditions (contravariant)
- **Properties**: subtype must preserve all properties specified in supertype overview

Substitution Principle Summary

- **Param Types**: Psub \(\geq\) Psuper  \(\text{contravariant}\) for inputs
- **Preconditions**: pre\_sub \(\Rightarrow\) pre\_super  \(\text{covariant}\) for inputs
- **Result Type**: Rsub \(\leq\) Rsuper  \(\text{covariant}\) for outputs
- **Postconditions**: post\_sub \(\Rightarrow\) post\_super  \(\text{contravariant}\) for outputs
- **Properties**: properties\_sub \(\Rightarrow\) properties\_super

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

Substitution Mystery

```java
mt1 = mt2.m (mt3);
```

If the Java compiler accepts this code, which of these are guaranteed to be true:

- a. The apparent type of mt2 is MysteryType2
- b. At the last statement, the actual type of mt2 is MysteryType2
- c. MysteryType2 has a method named m
- d. The MysteryType2.m method takes a parameter of type MysteryType3
- e. The MysteryType2.m method returns a subtype of MysteryType1
- f. After the last statement, the actual type of mt1 is MysteryType1