### Intuition

- Some interfaces have significant overlap in functionality
  - bicycles and vehicles
  - students and persons
  - both have names and both can be selected for juries
  - rectangles and shapes
  - both have a color
- These are all examples of an "is a" relationship
- This is a common (but poor) intuitive litmus test
- Interfaces define types, i.e., sets of possible values

### Extending Interfaces

- One interface can extend another

```java
interface X extends A, B { . . . }
```

- X implicitly includes all methods declared in A, B, and transitively above A and B

### Recall: Narrowing vs Widening

- Recall primitive types (e.g., long, int)
- Widening
  - Assign a "small" value to a variable of "big" type
  - This is always ok and can be done implicitly
  ```java
  void f(int i) {
      long x = i; // widening: always ok
  }
  ```
- Narrowing
  - Assign a "big" value to a variable of "small" type
  - The correctness of this cannot be checked by the compiler and so requires an explicit cast
  ```java
  void f(long x) {
      int i = x; // narrowing: compiler error
      int j = (int)x; // ok? programmer promise!
  }
  ```

### Narrowing and Widening Objects

- Subinterfaces are "smaller" types than superinterfaces
Argument Passing

- Method argument declared types must match signature
  ```java
  interface Course {
      void enroll(Student s) { . . . }
  }
  interface Jury {
      void select(Person p) { . . . }
  }
  ```
- Automatic (implicit) widening
  ```java
  Student s = ..;
  cse421.enroll(s); //ok (exact match)
  someJury.select(s); //ok (automatic widening)
  ```
- Cast for (explicit) narrowing
  ```java
  Person p = ..;
  someJury.select(p); //ok (exact match)
  cse421.enroll(p); //compiler complains (narrowing)
  ```

Simple Rule

- A variable / parameter of declared type T can refer to an object of dynamic type “at or below” T

The Problem

- Compiler enforces only:
  - Method signatures match in sub/super interfaces
- Not enforced by compiler:
  - Method specifications match
  - Mathematical models match
  - Constraint match
- That is, a subinterface can change the description of a method’s behavior!

BigNatural and BigInteger

```java
interface BigNatural {
    //@ens n = #n+1
    void increment();
    //@ens n = #n-1
    void decrement();
}
interface BigInteger {
    //@ens n = #n+1
    void increment();
    //@ens n = #n-1
    void decrement();
}
```
Dealing With This Problem

- Resolve/C++ approach:
  - Extensions can only add new operations
  - Extensions can not change the model or specs of extended component
- Alternative: Behavioral subtyping
  - Allow changes to method specs
  - Require new specs to be consistent with specs of extended component
- Beware:
  - Getting behavioral subtyping right is tricky and, even then, subtle problems can arise
  - More conservative approach is safer

Behavioral Subtyping

- Informally, A is a behavioral subtype of B when it does everything B does (and maybe more)
  - Everywhere a B is expected, an A can be used instead
- Must satisfy the Substitution Principle:
  - Any correct client that uses a B is still correct when given an A instead
- Example:
  - A class uses Creature (eg void foo(Creature c))
  - Actual argument might be a Creature, Person, Student, or Undergrad
  - Implementation of foo() should still be correct!
- Note: This is a requirement on the component provider (of A), not on the client

Substitution Principle

- If Undergrad is a subtype of Student
  - Any correct client of Student is still correct when given an Undergrad
- If Undergrad not a subtype of Student
  - There exists some correct client of Student that is no longer correct when given an Undergrad

Behavioral Subtyping Rules

- Subtype constraint ⇒ supertype constraint
  - Hence the informal “is a” litmus test
  - This condition alone, however, is not sufficient
- Each method in subinterface:
  - Requires less than in superinterface
    - Must work under more conditions
    - Add disjuncts (or) to requires clause
    - "contravariance of arguments"
  - Ensures more than in superinterface
    - Must guarantee more to client
    - Add conjuncts (and) to the ensures clause
    - "covariance of return"

More Formally...

- A’s constraint is “stronger”
  - Inv_A ==> Inv_B
- For each method, A “requires less”
  - Pre_A <= Pre_B
  - Pre_A <= Pre_B
- For each method, A “ensures more”
  - (Post_A ^ Pre_B) ==> Post_B
  - (Post_A ^ Pre_B) ==> Post_B
- Asides:
  - Omitted requires/ensures stands for true
  - Anything ==> true

A is a Behavioral Subtype of B if...

```java
//@mathmodel M_A == //@mathmodel M_B
//@constraint Inv_A ==> //@constraint Inv_B
interface A {
    interface B {
        //@requires Pre_A <= //@requires Pre_B
        //@ensures Post_A => //@ensures Post_B
        int f(int x, int y); int f(int x, int y);
        //@alters this => //@alters this
        //@alters this => //@alters this
        void g(String s); void g(String s);
    }
    @}mathmodel M_A
} @constraint Inv_A
A is a Behavioral Subtype of B if...

interface A {
    PrefA <= InvA ==>
    interface B {
        PrefB <= InvB ==>

        void g(String s);
    }

    PostA ^ PrefA ==>
    int f(int x, int y);
}

PostB ^ PrefB ==>
int f(int x, int y);

PregA <==
this SubsetOf
PregB ^ PregA ==>
void g(String s);

Example: A is Behavioral Subtype of B
//@mathmodel integer m
//@cons m mod 10 = 0
interface A {
    //@req x*y >= 0
    //@mathmodel integer m
    //@cons m is even
    interface B {
        //@req x = 0 or y = 0
        //@ens 0 < m
        int f(int x, int y);
    }

    PostA <==
    this SubsetOf
    PostB ^ PostA ==>
    void g(String s);
}

PostB <==
this
PostB

BigNatural Extends BigInteger?
//@mathmodel integer n
//@constraint n >= 0
interface BigNatural {
    //@alters n
    //@mathmodel integer n
    //@constraint
    interface BigInteger {
        //@alters n
        //@ens n = #n+1
        void increment();
    }

    Postn ^ PreB ==> Postn ^ PreA ==>
    void increment();

    Postn ^ PreB ==> Postn ^ PreA ==>
    void decrement();
}

Example: BigNatural & BigInteger
Should BigNatural extend BigInteger?
For behavioral subtyping, ask:
- Is BigNatural’s invariant stronger?
- Do all BigNatural methods require less?
- Do all BigNatural methods ensure more?

Example: BigNatural & BigInteger
Should BigNatural extend BigInteger?
- Is invariant stronger? Yes!
- BigNatural invariant is n >= 0
- BigInteger invariant is true
- Do methods require less? Yes!
- increment() requires the same (true) in both
- decrement() requires the same (true) in both
- Do methods ensure more? No!
- BigNatural decrement() ensures n=max(0,#n-1)
- BigInteger decrement() ensures n=#n-1
- Example client code that illustrates the problem
  boolean alwaysTrue(BigInteger i) {
      String oldi = i.toString();
      i.decrement();
      i.increment();
      return (oldi.equals(i.toString()));
  }
  alwaysTrue is correct for BigInteger, not for BigNatural
Example: Square & Rectangle

- These interfaces have similar abstract state (mathematical model)
  - two components: length, width
- These interfaces have similar public behavior (methods)
  - getArea(): returns the area (ie length * width)
  - widthStretch(): changes width of figure
  - lengthStretch(): changes length of figure
- Should we use inheritance?
  - Square extends Rectangle?
  - Rectangle extends Square?

Example: Square is a Rectangle?

- Is invariant stronger? Yes!
  - Square invariant is length = width and both are >= 0
  - Rectangle invariant is length and width both >= 0
- Do methods require less? Yes!
  - all methods require true in both classes
- Do methods ensure more? No!
  - Square widthStretch(s) ensures length = i * #length
  - Rectangle widthStretch() ensures length = #length
- Example client code that illustrates the problem
  ```java
  boolean alwaysTrue(Rectangle r) {
      double initialArea = r.getArea();
      double finalArea = r.widthStretch(2).getArea();
      return (finalArea == 2*initialArea);
  }
  ```

Example: Rectangle is a Square?

- Is invariant stronger? No!
  - Square invariant is length = width and both are >= 0
  - Rectangle invariant is length and width both >= 0
- Do methods require less? Yes!
  - all methods require true in both classes
- Do methods ensure more? No!
  - Square widthStretch(s) ensures length = i * #length
  - Rectangle widthStretch() ensures length = #length
- Example client code that illustrates the problem
  ```java
  boolean alwaysTrue(Square s) { 
      double initialArea = s.getArea();
      double finalArea = s.widthStretch(2).getArea();
      return (finalArea == 4*initialArea);
  }
  ```

Square Extends Rectangle?

```java
interface Square { 
    float getArea(); 
    void widthStretch(int i); 
}
```

```java
interface Rectangle { 
    float getArea(); 
    void widthStretch(int i); 
}
```

Rectangle Extends Square?

```java
interface Rectangle { 
    float getArea(); 
    void widthStretch(int i); 
}
```

```java
interface Square { 
    float getArea(); 
    void widthStretch(int i); 
}
```

Java Support for Subtyping

- Java does not enforce behavioral contracts
  - Type checking ensures only that arguments match
- Support for behavioral subtyping limited to very weak promises, such as:
  - If B has a visible method f(), A has a visible method f() with same signature
  - A can not decrease visibility of f()!
  - Arguments must match exactly (too much!)
- Return type can be a subtype (covariance)
  - If B's method f() can not throw an exception of type E, neither can A's f()
  - A can not increase the list of possible exceptions
- We'll talk about exceptions later...
Summary

- Interface extensions
  - Declaration syntax
  - Vocabulary: super/sub, base/derived, parent/child
  - Widening (up) is automatic
  - Narrowing (down) requires explicit cast
- Behavioral subtyping
- Substitution principle
- Subtyping rules
  - Strengthen the constraint
  - Weaken the requires of each method
  - Strengthen the ensures of each method
- Java rules (syntax)
  - Does not allow contravariance of argument types
  - Does allow covariance of return type