Subtype Polymorphism, Subtyping vs. Subclassing, Liskov Substitution Principle

Outline of today's class

- Subtype polymorphism
- Subtyping vs. subclassing

- Liskov Substitution Principle (LSP)
- Function subtyping
- Java subtyping
- Composition: an alternative to inheritance

Overriding vs. Overloading

- Method overloading is when two or more methods in the same class have the exact same name but different parameters
  - When overloading, one must change either the type or the number of parameters for a method that belongs to the same class. Overriding means that a method inherited from a parent class will be changed.
- Method overriding is when a derived class requires a different definition for an inherited method,
  - The method can be redefined in the derived class.
  - *In overriding* a method, everything remains exactly the same except the method definition – what the method does is changed slightly to fit in with the needs of the child class.
  - the method name, the number and types of parameters, and the return type will all remain the same.
  - Happens at runtime

Subtype Polymorphism

- Subtype polymorphism – the ability to use a subclass where a superclass is expected
  - Thus, dynamic method binding
    - `class A { void m() { … } }`
    - `class B extends A { void m() { … } }`
    - `class C extends A { void m() { … } }`
    - Client: `A a; … a.m();` // Call `a.m()` can bind to any of `A.m`, `B.m` or `C.m` at runtime!

override **`A.m`**

- Subtype polymorphism is the essential feature of object-oriented languages
  - Java subtype: B extends A or B implements I
  - A Java subtype is not necessarily a **true** subtype!

Benefits of Subtype Polymorphism

- Example: Application draws shapes on screen

- Possible solution in C:

```
enum ShapeType { circle, square };
struct Shape { ShapeType t };
struct Circle
{ ShapeType t; double radius; Point center; };
struct Square
{ ShapeType t; double side; Point topleft; };
```

Benefits of Subtype Polymorphism

```
void DrawAll(struct Shape *list[], int n) {
  int i;
  for (i=0; i< n; i++) {
      struct Shape *s = list[i];
      switch (s->t) {
         case square: DrawSquare(s); break;
         case circle: DrawCircle(s); break;
      }
  }
}
```

What’s bad about this solution?

Benefits of Subtype Polymorphism

- Example: OO Solution in Java:

```
abstract class Shape { public void draw(); }
class Circle extends Shape { … draw() }
class Square extends Shape { … draw() }
class Triangle extends Shape { … draw() }
void DrawAll(Shape[] list) {
  for (int i=0; i < list.length; i++) {
     Shape s = list[i];
     s.draw();
  }
}
```

Benefits of Subtype Polymorphism

- Enables extensibility and reuse
  - In our example, we can extend **`Shape`** hierarchy with no modification to the client of hierarchy, **`DrawAll`**
  - Thus, we can reuse **`Shape`** and **`DrawAll`**
- Subtype polymorphism enables the Open/closed principle
  - Software entities (classes, modules) should be open for extension but closed for modification
  - Credited to Bertrand Meyer

Benefits of Subtype Polymorphism

- “Science” of software design teaches Design Patterns

- Design patterns promote design for extensibility and reuse

- Nearly all design patterns make use of subtype polymorphism

Outline

- Subtype polymorphism
- Subtyping vs. subclassing

- Liskov Substitution Principle (LSP)
- Function subtyping
- Java subtyping

- Composition: an alternative to inheritance

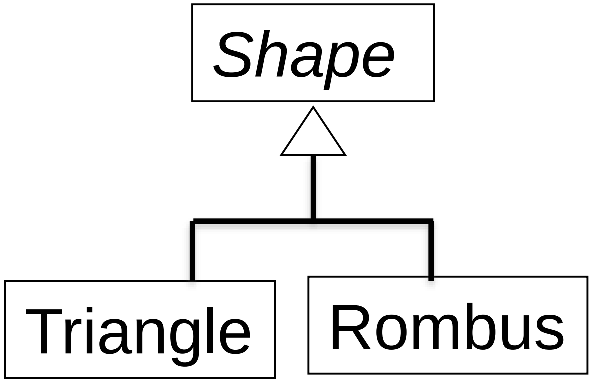
What is True Subtyping?

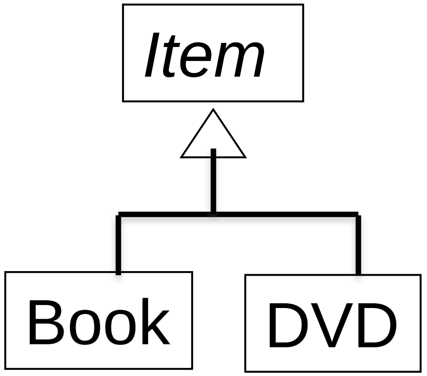
- Subtyping, conceptually
  - B is subtype of A means every B is an A
  - In other words, a B object can be substituted where an A object is expected

- The notion of true subtyping connects subtyping in the real world with Java subtyping

Examples of Subtypes

- Subset subtypes
  - Integer is a subtype (subset) of Number
  - range [0..10] is a subtype of range [-10…10]
- Other subtypes
  - Every book is a library item
  - Every DVD is a library item
  - Every triangle is a shape
  - Etc.





True Subtypes are Substitutable

- Subtypes are substitutable for supertypes
  - Instances of subtypes won't surprise client by requiring "more" than the supertype
  - Instances of subtypes won't surprise client by returning "less" than its supertype

- Java subtyping is realized through subclassing
  - Java subtype is not the same as true subtype!

Subtyping and Subclassing

- Subtyping and substitutability --- specification notions
  - B is a subtype of A if and only if a B object can be substituted where an A object is expected, in any context
- Subclassing and inheritance --- implementation notions
  - B **`extends`** A, or B **`implements`** A
  - B is a Java subtype of A, but not necessarily a true subtype of A!

Subtyping and Subclassing

- Subtype
  - Substitution
  - B is a subtype of A iff an object of type B can masquerade as an object of type A in any context
- Subclass
  - Inheritance
  - Abstracts out repeated code
  - To create a new class just code the differences
  - Every subclass is a Java subtype
    - But not necessarily a true subtype

True Subtype

- We say that (class) B is a true subtype of A if B is a subclass of A and has a stronger specification than A
  - Maybe weaker requirements
  - Maybe stronger results

- Be aware of this when designing inheritance hierarchies!

- Java subtypes that are not true subtypes can be confusing and *dangerous*
  - Can cause subtle, hard to find bugs

Subclassing. Inheritance Makes it Easy to Add Functionality

```
class Product {
  private String title;
  private String description;
  private float price;
  public float getPrice() { return price; }
  public float getTax() {
     return getPrice()*0.08f;
  }
}
```

… and we need a class for Products that are on sale

Code cloning is a bad idea! Why?

```
class SaleProduct {
  private String title;
  private String description;
  private float price;
  private float factor; // extends Product
  public float getPrice() {
    return price*factor; } // extends Product
  public float getTax() {
     return getPrice()*0.08f;
  }
}
```

Subclassing

- What's a better way to add this functionality?

```
class SaleProduct extends Product {
    private float factor;
    public float getPrice() {
        return super.getPrice()*factor;
    }
}
```

… Subclassing keeps small extensions small

Benefits of Subclassing

- Don't repeat unchanged fields and methods
  - Simpler maintenance: fix bugs once
  - Differences are clear (not buried under mass of similarity!)
  - Modularity: can ignore private fields and methods of superclass
- Can substitute new implementations where old one is expected (the benefit of subtype polymorphism)
- Another example: **`Timestamp extends Date`**
- Disadvantage
  - May break equality
    - See Duration example from previous lecture
  - If we implement equality for SaleProduct in the most intuitive way, equality won't be symmetric when comparing a SaleProduct and a Product!

Subclassing Can Be Misused

- Poor planning leads to muddled inheritance hierarchies. Requires careful planning

- If a class is not a true subtype of its superclass, it can surprise client
- If class depends on implementation details of superclass, changes in superclass can break subclass. "Fragile base class problem"

Classic Example of Subtyping vs. Subclassing: Every Square is a Rectangle, right?

Thus, **`class Square extends Rectangle { … }`**

But is a **`Square`** <u>a true subtype</u> of **`Rectangle`**? In other words, is **`Square`** <u>substitutable</u> for **`Rectangle`** in client code?

```
class Rectangle {
  // effects: this_post.width=w,this_post.height=h
  public void setSize(int w, int h);
  // returns: area of rectangle
  public int area();
}
```

Every Square is a Rectangle, right?

```
class Square extends Rectangle { … }
  // requires: w = h
  // effects: this_post.width=w,this_post.height=h
  Choice 1: public void setSize(int w, int h);

  // effects: this_post.width=w,this_post.height=w
  Choice 2: public void setSize(int w, int h);

  // effects: this_post.width=s,this_post.height=s
  Choice 3: public void setSize(int s);

  // effects: this_post.width=w,this_post.height=h
  // throws: BadSizeException if w != h
  Choice 4: public void setSize(int w, int h);
```

Every Square is a Rectangle, right?

- Choice 1 is not good
  - It requires more! Clients of Rectangle are justified to have **`Rectangle r; … r.setSize(5,4)`**

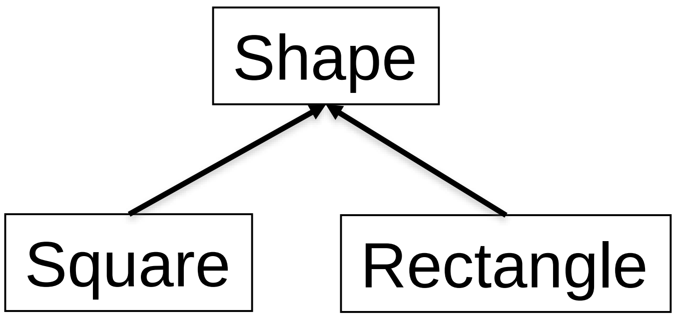
  - In formal terms: spec of **`Square's setSize`** is not stronger than spec of **`Rectangle's setSize`**
    - It weakens Rectangle's setSize spec.
  - Thus, Square can't be substituted for a Rectangle

Every Square is a Rectangle, right?

- Choice 4?
  - It throws an exception that clients of Rectangle are not expecting and not handling
  - Thus, a Square might cause a problem if substituted for a Rectangle
- Choice 3?
  - Clients of Rectangle can write … **`r.setSize(5,4).`** Square works with **`r.setSize(5)`**
  - Overload not an override
- Choice 2?
  - Client: **`Rectangle r = new Square();`** … **`r.setSize(5,4); assert(r.area()==20)`**
  - Again, Square surprises client with behavior that is different from Rectangle's

Every Square is a Rectangle, right?

- Square is not a true subtype of Rectangle
  - Rectangles are expected to have height and width that can change independently
  - Squares violate that expectation. Surprise clients
- Is Rectangle a true subtype of Square?
  - No. Squares are expected to have equal height and width. Rectangles violate this expectation
- One solution: make them unrelated



Liskov Substitution Principle (LSP)

- Due to Barbara Liskov, Turing Award 2008
- LSP: A subclass should be substitutable for superclass. I.e., every subclass should be a true subtype of its superclass
- Ensure that B is a true subtype of A by reasoning at the specification level
  - B should not remove methods from A
  - For each B.m that "substitutes" A.m, B.m's spec does not weaken A.m's spec
    - Client: **`A a; … a.m(int x,int y);`** Call **`a.m`** can bind to B's **`m`**. B's **`m`** should not surprise client
  - Any property guaranteed by supertype must be guaranteed by subtype
    - The subtype is permitted to strengthen and add properties
  - Anything provable about A is provable about B
  - If instance of subtype is treated purely as supertype – only supertype methods and fields queried – then result should be consistent with an object of the supertype being manipulated

Liskov Substitution Principle (LSP)

- Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Liskov Substitution Principle Rules

- Contravariance of method arguments in the subtype. (The arguments may be wider than the parent's method required, e.g. going from float to double)
- Covariance of return types in the subtype. (The return types may be narrower than the parent's return types, e.g. going from double to float)
- No new exceptions should be thrown, unless the exceptions are subtypes of exceptions thrown by the parent.
- Preconditions cannot be strengthened in the subtype. (You cannot require more than the parent)
- Postconditions cannot be weakened in the subtype. (You cannot guarantee less than the parent)
- Invariants must be preserved in the subtype.
- History Constraint – the subtype must not be mutable in a way the supertype wasn't. For instance MutablePoint cannot inherit ImmutablePoint without violating the History Constraint (as it allows mutations which its supertype didn't)

Substitution Principle for Classes

- If B is a true subtype of A, a B can always be substituted for an A
- Any property guaranteed by supertype must be guaranteed by subtype
  - Subtype can strengthen and add properties
  - Anything provable about A is provable about B
  - If an instance of subtype is treated purely as a supertype (only methods and fields queried) then result should be consistent with results from supertype
- No specification weakening
  - No method removal
  - Overriding methods have a stronger spec

Substitution principle for methods

- Constraints on methods
  - For each method in supertype, subtype must have a corresponding override method
  - May also introduce new methods
- Each override method must have a stronger or equal spec
  - Ask nothing extra of client
    - Weaker or equal precondition
    - Requires class is at most as strict as supertype requires
  - Guarantee as much as supertype
    - Effects clause is at least as strict as supertype
    - No new entries in modifies clause
  - The overriding method satisfies the supertype spec
- No new exceptions in domain


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Box is a BallContainer?

```
class BallContainer {
 // modifies: this
 // effects: adds b to this container if b is not
 //          already in
 // returns: true if b is added, false otherwise
 public boolean add(Ball b);
 …
}
class Box extends BallContainer { // good idea?
 // modifies: this
 // effects: adds b to this Box if b is not
 //          already in and this Box is not full
 // returns: true if b is added, false otherwise
 public boolean add(Ball b);
 …
}
```

Exercise: Reason About Specs

```
class Rectangle {
  // effects: this_post.width=w,this_post.height=h
  public void setSize(int w, int h);
}
class Square extends Rectangle { …
  // requires: w = h
  // effects: this_post.width=w,this_post.height=h
  public void setSize(int w, int h);
}
```

Summary So Far

- Java subtypes (realized with extends, implements) must be true subtypes
  - Java subtypes that are not true subtypes are dangerous and confusing
- When B is a Java subtype of A, ensure
  - B, does not remove methods from A
  - A substituting method B.m has stronger spec than method A.m which it substitutes
  - Guarantees substitutability

Outline of today's class

- Subtype polymorphism
- Subtyping vs. subclassing

- Liskov Substitution Principle (LSP)
- Function subtyping
- Java subtyping
- Composition: an alternative to inheritance

Intuition:
Type Signature is a Specification

- Type signature (parameter types + return type) is a contract too

E.g., `double f(String s, int i) {…}`

Precondition: arguments are a `String` and an `int`

Postcondition: result is a `double`

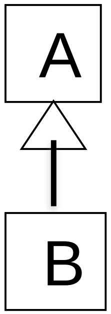
- We need reasoning about behavior and effects, so we added requires, effects, etc.

Function Subtyping

- In programming languages function subtyping deals with substitutability of functions
  - Question: under what conditions on the parameter and return types A,B,C and D, is function A f(B) substitutable for C f'(D)
  - Reasons at the level of the type signature
  - Rule: A f(B) is a function subtype of C f'(D)
    - iff A is a subtype of C and B is a supertype of D
    - Guarantees substitutability!

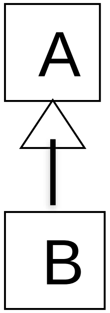
Type Signature of Substituting Method is Stronger

- Method parameters (inputs):
  - Parameter types of A.m may be replaced by supertypes in subclass B.m. ”contravariance”
    - E.g., A.m(String p) and B.m(Object p)
  - B.m places no extra requirements on the client!
    - E.g., client: `A a; … a.m(q)`. Client knows to provide `q` a String.
    - Thus, client code will work fine with `B.m(Object p)`, which asks for less: an Object, and clearly, every String is an Object
  - Java does not allow change of parameter types in an overriding method



Type Signature of Substituting Method is Stronger

- Method returns (results):
  - Return type of A.m may be replaced by subtype in subclass B.m. "covariance"
    - E.g., Object A.m() and String B.m()
  - B.m does not violate expectations of the client!
    - Result type of A.m() may be replaced by a subtype in B.m() in the subclass
      - Doesn't violate client expectations
    - E.g., `Object o = a.m()`. Client expects an `Object`. Thus, `String` will work fine
  - No new exceptions. Existing exceptions can be replaced by subtypes
  - Java does allow a subtype return type in an overriding method!



Properties Class from the JDK

`Properties` stores String key-value pairs. It extends **`Hashtable`** so **`Properties`** is a Java subtype of **`Hashtable`**. What's the problem?

```
class Hashtable {
 // modifies: this
 // effects: associates value with key
 public void put(Object key, Object value);
 // returns: value associated with key
 public Object get(Object key);
}
class Properties extends Hashtable { // simplified
 // modifies: this
 // effects: associates String value with String key
 public void put(String key, String value) {
    super.put(key, value);
 }
 // returns: value associated with key
 public String get(String key) {
    return (String) super.get(key);
 }
```

Exercise

```
class Hashtable {
   public void put(Object key, Object value);
   public Object get(Object key);
}

class Properties extends Hashtable {
   public void put(String key, String value);
   public String get(String key);
}
```

Exercise

```
class Product {
   Product recommend(Product p);
}
```

Which one is a function subtype of **`Product.recommend?`**

```
class SaleProduct extends Product {
   Product recommend(SaleProduct p);
   SaleProduct recommend(Product p);
   Product recommend(Object p);
   Product recommend(Product p) throws
                                NoSaleException;
}
```

Exercise

```
class Product {
   Product recommend(Product p);
}
```

Which one is a function subtype of **`Product.recommend?`**

```
class SaleProduct extends Product {
   Product recommend(SaleProduct p); // bad
   SaleProduct recommend(Product p); // OK
   Product recommend(Object p); // OK but overload
   Product recommend(Product p) throws
                        NoSaleException; // bad
}
```

Reasoning about Specs

- Function subtyping reasons with type signatures
- Remember, type signature is a specification!
  - Precondition: requires arguments of given type
  - Postcondition: promises result of given type
- Compiler checks function subtyping
- Behavioral specifications add reasoning about behavior and effects
  - Precondition: stated by requires clause
  - Postcondition: stated by modifies, effects, returns and throws clauses
- To ensure A is a true subtype of B, we must reason about behavioral specifications (as we did earlier)

Reason about Specs

- Behavioral subtyping generalizes function subtyping
- B.m is a true subtype (behavioral subtype) of A.m
  - B.m has <u>weaker</u> precondition than A.m
    - This generalizes the requirement of function subtyping: "B.m's parameter is a <u>super</u>type of A.m's parameter"
    - Contravariance
  - B.m has <u>stronger</u> postcondition than A.m
    - Generalizes "B.m's return is a <u>sub</u>type of A.m's return"
    - Covariance
  - These 2 conditions guarantee B.m's spec is stronger than A.m's spec, and B.m is substitutable for A.m

Outline of today's class

- Subtype polymorphism
- Subtyping vs. subclassing

- Liskov Substitution Principle (LSP)
- Function subtyping
- Java subtyping
- Composition: an alternative to inheritance

Java Subtypes

- Java types are defined by classes, interfaces, primitives
- Java subtyping stems from declarations
  - B **`extends`** A
  - B **`implements`** A
- In a Java subtype, a “substituting” method is an overriding method
  - Has same parameter types
  - Has compatible (same or subtype) return type
  - Has no additional declared exceptions

Overloading vs. Overriding

- If a method has same name, but different parameter types, it overloads not overrides

```
class Hashtable {
   public void put(Object key, Object value);
   public Object get(Object key);
}
class Properties extends Hashtable {
   public void put(String key, String value);
   public String get(String key);
}
```

overload Hashtable's put and get

Overloading vs. Overriding

- A method family contains multiple implementations of same name + parameter types (but not return type!)
- Which method family? is determined at compile time based on compile-time types
  - E.g., family put(Object key, Object value)
    - or family put(String key, String value)
- Which implementation from the method family runs, is determined at runtime based on the runtime type of the receiver

How does a Method Call Execute?

- For example, x.foo(5);
- Compile time
  - Determine what class to look in – compile time class
  - Determine the method signature (method family)
  - Find all methods in the class with the right name
    - Includes *inherited* methods
    - Keep only methods that are accessible
      - E.g. a private method is not accessible to calls from outside the class
    - Keep only methods that are applicable
      - The types of the actual arguments (e.g. 5 has type int above) must be subtypes of the corresponding formal parameter type
    - Select the most specific method
      - m1 is more specific than m2 if each argument of m1 is a subtype of the corresponding argument of m2
    - Keep track of the method's signature (argument types) for run-time

How does a Method Call Execute?

- Run time
  - Determine the run-time type of the receiver
    - x in this case
    - Look at the object in the heap to find out what its run-time type is
  - Locate the method to invoke
    - Starting at the run-time type, look for a method with the right name and argument types that are identical to those in the method found statically, i.e. method family
    - If it is found in the run-time type, invoke it.
    - Otherwise, continue the search in the superclass of the run-time type
    - This procedure will always find a method to invoke, due to the checks done during static type checking

Example

```
class GenericAnimal {
   public String talk() {
   return "Noise"; }
}

class Bird extends GenericAnimal {
   public String talk(){
   return "Chirp"; }
}

class Cat extends GenericAnimal {
   public String talk(){
   return "Meow"; }
}

class Dog extends GenericAnimal {
   public String talk(){
   return "Woof"; }
}

class GizmoTheCat extends Cat {
   public String talk(){
   return "Hello, I would like some oatmeal."; }
}
```

```
public class AnimalTalk {
        public static void main(String[] args) {
                GenericAnimal A = new GenericAnimal();
                System.out.println(A.talk());

                Bird B = new Bird();
                System.out.println(B.talk());

                Cat C = new Cat();
                System.out.println(C.talk());

                GizmoTheCat G = new GizmoTheCat();
                System.out.println(G.talk());

                // what does this print?
                GizmoTheCat G2 = new GizmoTheCat();
                GenericAnimal F = G2;  // Compile time type? Runtime type?
                System.out.println(F.talk());

        }
}
```

Remember **Duration**

Two method families.

```
class Object {
   public boolean equals(Object o);
}
class Duration {
   public boolean equals(Object o);
   public boolean equals(Duration d);
}
Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler choses family equals(Duration d)
```

Remember **Duration**

```
class Object {
   public boolean equals(Object o);
}
class Duration {
   public boolean equals(Object o);
   public boolean equals(Duration d);
}
Object d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// At compile-time: equals(Object o) (method family)
// At runtime: Duration.equals(Object o)
```

Remember **Duration**

```
class Object {
   public boolean equals(Object o);
}
class Duration {
   public boolean equals(Object o);
   public boolean equals(Duration d);
}
Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler choses equals(Object o)
// At runtime: Duration.equals(Object o)
// receiver type is Duration at runtime
```

Remember **Duration**

```
class Object {
   public boolean equals(Object o);
}
class Duration {
   public boolean equals(Object o);
   public boolean equals(Duration d);
}
Duration d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler choses equals(Object o)
// At runtime: Duration.equals(Object o)
// receiver type is Duration at runtime
```

Exercise

```
class Y extends X { … }

class A {
   X m(Object o) { … }
}
class B extends A {
   X m(Z z) { … }
}
class C extends B {
   Y m(Z z) { … }
}
```

```
A a = new B();
Object o = new Object();
// Which m is called?
X x = a.m(o);
```

```
A a = new C();
Object o = new Z();
// Which m is called?
X x = a.m(o);
```

Exercise

```
class Y extends X { … }
class W extends Z { … }
class A {
   X m(Z z) { … }
}
class B extends A {
   X m(W w) { … }
}
class C extends B {
   Y m(W w) { … }
}
```

```
A a = new B();
W w = new W();
// Which m is called?
X x = a.m(w);
```

```
B b = new C();
W w = new W();
// Which m is called?
X x = b.m(w);
```

Static and Dynamic Types

- B extends A and C extends B.
- The dynamic type of an object (the type used in the new) is its actual *runtime* type
  - it defines the actual methods that are present for an object.
- The static type of an object reference (a variable) is a *compile-time* type
  - it defines, or rather declares, which methods can be called on the object that the variable references.
- The static type of a variable should always be of the same type or a supertype of the dynamic type of the object it references.
- Java Language Spec on method invocation is complex

Java Subtyping Guarantees

- A variable's runtime type (i.e., the class of its runtime object) is a Java subtype of the variable's declared class (Not true in C++!)

  `Object o  = new Date();` // OK

  `Date d = new Object();` // Compile-time error

- Thus, objects always have implementations of the method specified at the call site
  - Client: **`B b;`** … **`b.m()`** // Runtime object has **`m()`**
  - If all subtypes are true subtypes, spec of runtime target **`m()`** is stronger than spec of **`B.m()`**

Subclassing is Difficult

Before:

```
class B {
   private int c=0;
   void inc1() { c++; }
   void inc2() { c++; }
}
class A extends B {
   @Override
   void inc2() {
     inc1();
   }
}
```

After a tiny change:

```
class B {
   private int c=0;
   void inc1() { inc2(); }
   void inc2() { c++; }
}
class A extends B {
   @Override
   void inc2() {
     inc1();
   }
}
```

Fragile Base Class Problem

- Previous slide showed an example of the Fragile Base Class Problem
- Occurs when the implementation of a subclass depends on implementation details in the superclass. Seemingly innocuous changes in the superclass can break the subclass

Subclassing is Difficult

- A set that <u>counts</u> the number of <u>attempted</u> additions:

```
class InstrumentedHashSet extends HashSet {
  private int addCount = 0;
  public InstrumentedHashSet(Collection c) {
   super(c);
  }
  public boolean add(Object o) {
   addCount++; return super.add(o);
  }
  public boolean addAll(Collection c) {
   addCount += c.size(); return super.addAll(c);
  }
  public int getAddCount() { return addCount; }
}
```

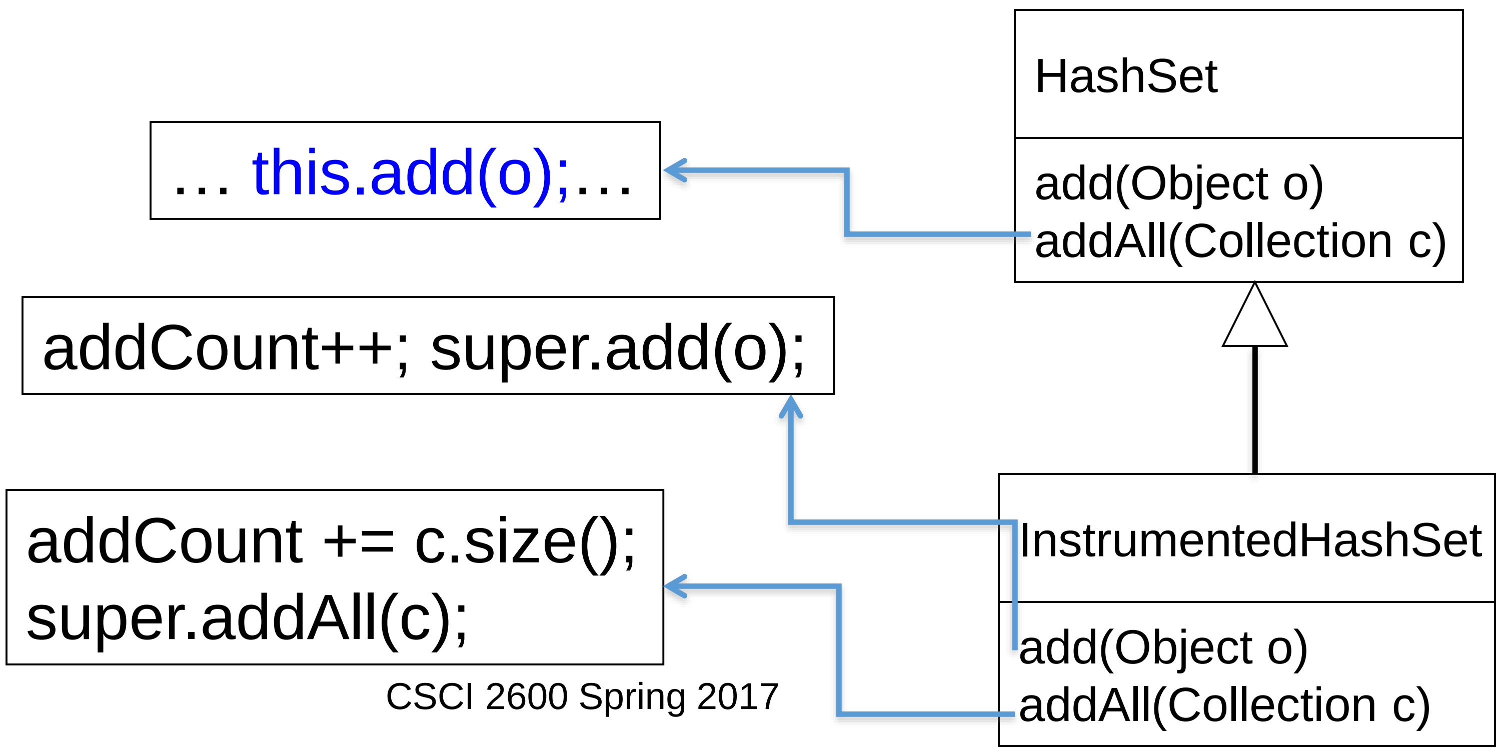
Subclassing is Difficult

- **`InstrumentedHashSet`** is a true subtype of **`HashSet`**. But… Something goes quite wrong here

```
class InstrumentedHashSet extends HashSet {
  private int addCount = 0;
  public InstrumentedHashSet(Collection c) {
   super(c);
  }
  public boolean add(Object o) {
   addCount++; return super.add(o);
  }
  public boolean addAll(Collection c) {
   addCount += c.size(); return super.addAll(c);
  }
  public int getAddCount() { return addCount; }
}
```

Subclassing is Difficult

```
InstrumetedHashSet s=new InstrumentedHashSet();
System.out.println(s.getAddCount()); // 0
s.addAll(Arrays.asList(“One”,”Two”));
System.out.println(s.getAddCount()); // Prints?
```



The Yo-yo Problem

- **`this.add(o)`** in superclass **`HashSet`** calls **`InstrumentedHashSet.add`**! Callback.

- Example of the yo-yo problem. Call chain "yo-yos" from subclass to superclass back to subclass
  - **`InstrumentedHashSet.addAll`** calls **`HashSet.addAll`** calls **`InstrumentedHashSet.add`**
- Behavior of **`HashSet.addAll`** depends on subclass **`InstrumentedHashSet`**!

Java Subtyping with Interfaces

Why Set and not HashSet?
Avoid implementation detail

```
class InstrumentedHashSet implements Set {
  private final Set s = new HashSet();
  private int addCount = 0;
  public InstrumentedHashSet(Collection c) {
   this.addAll(c);
  }
  public boolean add(Object o) {
   addCount++; return s.add(o);
  }
  public boolean addAll(Collection c) {
   addCount += c.size(); return s.addAll(c);
  }
  public int getAddCount() { return addCount; }
  // … Must add all methods specified by Set
}
```

Java Subtyping with Interfaces

- interface inheritance
  - Client codes against type signature of interface methods, not concrete implementations
  - Behavioral specification of an interface method often unconstrained
    - Often, any (later) implementation is stronger!
  - Facilitates composition and wrapper classes as in the **`InstrumentedHashSet`** example

Java Subtyping with Interfaces

- In JDK and the Android SDK
  - Implement multiple interfaces, extend single <u>abstract</u> superclass (very common!)
    - Abstract classes minimize number of methods new implementations must provide
    - Abstract classes facilitate new implementations
    - Using abstract classes is optional, so they don't limit freedom
  - Extending a concrete class is rare and often problematic (e.g., Properties, Timestamp, which we saw in the Equality lecture)

Why prefer **`implements A`** over **`extends A`**

- A class has exactly one superclass. In contrast, a class may implement multiple interfaces. An interface may extend multiple interfaces
- Interface inheritance gets the benefit of subtype polymorphism
  - And avoids the pitfalls of subclass inheritance, such as the fragile base class problem, etc.
- Multiple interfaces, single abstract superclass gets most of the benefit

Outline

- Subtype polymorphism
- Subtyping vs. subclassing

- Liskov Substitution Principle (LSP)
- Function subtyping
- Java subtypes
- Composition: an alternative to inheritance

Composition

- **`Properties`** is not a true subtype of **`Hashtable`**. Thus, cannot subclass. An alternative solution?
- Subclassing is a bad idea for the **`InstrumentedHashSet`** too. An alternative?
- **`Box`** is not a true subtype of **`BallContainer.`** Cannot subclass.
- Composition!

Properties Class from the JDK

`Properties` stores String key-value pairs. It extends **`Hashtable`** so **`Properties`** is a Java subtype of **`Hashtable`**. What's the problem?

```
class Hashtable {
 // modifies: this
 // effects: associates value with key
 public void put(Object key, Object value);
 // returns: value associated with key
 public Object get(Object key);
}
class Properties extends Hashtable { // simplified
 // modifies: this
 // effects: associates String value with String key
 public void put(String key, String value) {
    super.put(key, value);
 }
 // returns: value associated with key
 public String get(String key) {
    return (String) super.get(key);
 }
```

Exercise

```
class Hashtable {
   public void put(Object key, Object value);
   public Object get(Object key);
}

class Properties extends Hashtable {
   public void put(String key, String value);
   public String get(String key);
}
```

Exercise

```
class Product {
   Product recommend(Product p);
}
```

Which one is a function subtype of **`Product.recommend?`**

```
class SaleProduct extends Product {
   Product recommend(SaleProduct p);
   SaleProduct recommend(Product p);
   Product recommend(Object p);
   Product recommend(Product p) throws
                                  NoSaleException;
}
```

Exercise

```
class Product {
   Product recommend(Product p);
}
```

Which one is a function subtype of **`Product.recommend?`**

```
class SaleProduct extends Product {
   Product recommend(SaleProduct p); // bad
   SaleProduct recommend(Product p); // OK
   Product recommend(Object p); // OK but overload
   Product recommend(Product p) throws
                             NoSaleException; // bad
}
```

Properties

Wrapper class

The delegate

```
class Properties { // simplified

 private Hashtable ht = new Hashtable();

 // modifies: this
 // effects: associates value with key
 public void setProperty(String key,String value)
 {
    ht.put(key,value);
 }
 // returns: value associated with key
 public void getProperty(String key)
 {
    return (String) ht.get(key);
 }
```

InstrumentedHashSet

The delegate

```
class InstrumentedHashSet {
  private final Set s = new HashSet();
  private int addCount = 0;
  public InstrumentedHashSet(Collection c) {
   s.addAll(c);
  }
  public boolean add(Object o) {
   addCount++; return s.add(o);
  }
  public boolean addAll(Collection c) {
   addCount += c.size(); return s.addAll(c);
  }
  public int getAddCount() { return addCount; }
}
```

Box

The delegate

```
class Box {
  private BallContainer ballContainer;
  private double maxVolume;

  public Box(double maxVolume) {
    this.ballContainer = new BallContainer();
    this.maxVolume = maxVolume;
  }
  public boolean add(Ball b) {
   if (b.getVolume() + ballContainer.getVolume()
        > maxVolume)
    return false;
   else
    return ballContainer.add(b);
  }
  …
```

Composition

- <u>Implementation reuse without inheritance</u>
  - More common than reuse through subclassing
- Easy to reason about
- Works around badly-designed classes
- Disadvantages
  - Adds level of indirection
  - Tedious to write
  - Does not preserve subtyping

Composition Does not Preserve Subtyping

- **`InstrumentedHashSet`** is not a **`Set`** anymore
  - So can't substitute it
- It may be a true subtype of **`Set`**!
  - But Java doesn't know that
- That nice trick with interfaces to the rescue
  - Declare that the class implements interface **`Set`**
  - Requires that such interface exists

Nice Trick with Interfaces

```
class InstrumentedHashSet implements Set {
  private final Set s = new HashSet();
  private int addCount = 0;
  public InstrumentedHashSet(Collection c) {
   this.addAll(c);
  }
  public boolean add(Object o) {
   addCount++; return s.add(o);
  }
  public boolean addAll(Collection c) {
   addCount += c.size(); return s.addAll(c);
  }
  public int getAddCount() { return addCount; }
  // … Must add all methods specified by Set
}
```