

## Subtype Polymorphism, Subtyping vs. Subclassing, Liskov Substitution Principle

## Announcements

- HW5 out, due Friday October 30<sup>th</sup>
  - Part 1: Questions on material we'll cover today
  - Part 2: BFS using your graph from HW4
- HW4 resubmit due Tuesday at 2pm
  - Auto-graded part of HW4 will be the average of the two submissions
- Quiz 6 today

2

## Outline of today's class

- Subtype polymorphism
- Subtyping vs. subclassing
- Liskov Substitution Principle (LSP)
- Function subtyping
- Java subtyping
- Composition: an alternative to inheritance

3

## 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!
- 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; };

```

5

## 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 really bad about this solution?

Fall 15 CSCI 2600, A Milanova

6

## 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();
    }
}
```

Fall 15 CSCI 2600, A. Milanova

7

## 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

Fall 15 CSCI 2600, A. Milanova

8

## 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

Fall 15 CSCI 2600, A. Milanova

9

## Outline

- Subtype polymorphism
- **Subtyping vs. subclassing**
- Liskov Substitution Principle (LSP)
- Function subtyping
- Java subtyping
- Composition: an alternative to inheritance

10

## 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

Fall 15 CSCI 2600, A. Milanova

11

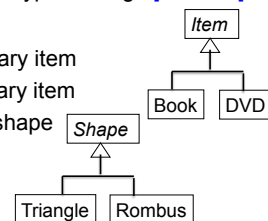
## Examples of Subtypes

### ■ Subset subtypes

- **int** is a subtype of **real**
- 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.



Fall 15 CSCI 2600, A. Milanova

12

## 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**!

13

## 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!

14

## True Subtype

- We say that (class) B is a **true subtype** of A if B has a stronger specification than A
- Heed when designing inheritance hierarchies!
- Java subtypes that are not true subtypes are **confusing** and **dangerous**

Fall 15 CSCI 2600, A Milanova

15

## 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

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

16

## 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;
    }
}
```

Fall 15 CSCI 2600, A Milanova (based on an example by Michael Ernst)

17

## 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

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

18

## 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`

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst) 19

## 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"

Fall 15 CSCI 2600, A Milanova (based on a slide by Michael Ernst)

20

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

Thus, class `Square` extends `Rectangle` { ... }

But is a `Square` a true subtype of `Rectangle`? In other words, is `Square` substitutable 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();  
}
```

Fall 15 CSCI 2600, A Milanova

21

## 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);
```

Fall 15 CSCI 2600, A Milanova (based on a slide by Mike Ernst)

22

## 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`
    - Because the precondition of `Rectangle`'s `setSize` does not imply precondition of `Square`'s `setSize`
  - Thus, a `Square` can't be substituted for a `Rectangle`

Fall 15 CSCI 2600, A Milanova

23

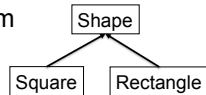
## Every Square is a Rectangle, right?

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

24

## 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



Fall 15 CSCI 2600, A Milanova

25

## 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);
  ...
}
```

26

## 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 is stronger than 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

27

## 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);
}
```

Fall 15 CSCI 2600, A Milanova (based on a slide by Mike Ernst)

28

## 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

29

## Outline of today's class

- Subtype polymorphism
- Subtyping vs. subclassing
- Liskov Substitution Principle (LSP)
- Function subtyping
- Java subtyping
- Composition: an alternative to inheritance

Fall 15 CSCI 2600, A Milanova

30

## 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.

Fall 15 CSCI 2600, A Milanova

31

### 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!

Fall 15 CSCI 2600, A Milanova

32

### 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. More on Java overriding shortly

33

### 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!
    - E.g., client: `A a; ... 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!

34

### 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);
    }
}
```

35

### 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);
}
```

Fall 15 CSCI 2600, A Milanova (based on example by Michael Ernst)

36

## 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;  
}
```

Fall 15 CSCI 2600, A Milanova (based on example by Michael Ernst)

37

## 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 **B** is a true subtype of **A**, we must reason about behavioral specifications (as we did earlier)

38

## Reason about Specs

- Behavioral subtyping generalizes function subtyping
- B.m** is a true subtype (behavioral subtype) of **A.m**
  - B.m** has weaker precondition than **A.m**
    - This generalizes the requirement of function subtyping: "**B.m**'s parameter is a supertype of **A.m**'s parameter"
    - Contravariance
  - B.m** has stronger postcondition than **A.m**
    - Generalizes "**B.m**'s return is a subtype 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

Fall 15 CSCI 2600, A Milanova

40

## 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

Fall 15 CSCI 2600, A Milanova (based on slide by Mike Ernst)

41

## 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);  
}
```

Fall 15 CSCI 2600, A Milanova

42

## 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

Fall 15 CSCI 2600, A Milanova (based on slides by Mike Ernst)

43

## Remember Duration

```
class Object {                                Two method families.
    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 chooses family equals(Duration d)
```

Fall 15 CSCI 2600, A Milanova

44

## 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)
// At runtime: Duration.equals(Duration d)
```

45

## 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 chooses equals(Object o)
// At runtime: Duration.equals(Object o)
```

46

## 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 chooses equals(Object o)
// At runtime: Duration.equals(Duration d)
```

47

## Exercise

```
class Y extends X { ... } A a = new B();
                           Object o = new Object();
class A {
    X m(Object o) { ... } // Which m is called?
}
class B extends A {
    X m(Z z) { ... }
}
class C extends B {
    Y m(Z z) { ... }
}
A a = new C();
Object o = new Z();
// Which m is called?
X x = a.m(o);
```

Fall 15 CSCI 2600, A Milanova

48

## Exercise

```
class Y extends X { ... } A a = new B();
class W extends Z { ... } W w = new W();
class A {
    X m(Z z) { ... } // Which m is called?
}
class B extends A {
    X m(W w) { ... }
}
class C extends B {
    Y m(W w) { ... }
}
```

`X x = a.m(w);`

`B b = new C();`

`W w = new W();`

`X x = b.m(w);`

Fall 15 CSCI 2600, A.Milanova

49

## 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()`

50

## Next time

- More on Java subtyping
- Composition: an alternative to subclassing

Fall 15 CSCI 2600, A.Milanova

51