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

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:
    ```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; }
    ```
Example: OO Solution in Java:

```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

### 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
  - 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.
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!

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

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

class SaleProduct extends Product {
    private float factor;
    public float getPrice() {
        return super.getPrice() * factor;
    }
    public float getTax() {
        return getPrice() * 0.08f;
    }
}

Subclassing keeps small extensions small

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;
    }
}

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`

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 a true subtype of `Rectangle`? In other words, is `Square` substitutable for `Rectangle` in client code?

```
class Rectangle {
    // requires: w = h
    // effects: this.post.width=w,this.post.height=h
    public void setSize(int w, int h); // effects: this.post.width=w, this.post.height=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); // requires: w = h
    // effects: this.post.width=w, this.post.height=h
    Choice 2: public void setSize(int w, int h); // requires: w = h
    // effects: this.post.width=w, this.post.height=h
    Choice 3: public void setSize(int s); // requires: w = h
    // effects: this.post.width=s, this.post.height=s
    Choice 4: public void setSize(int w, int h); // requires: w = h
    // throws: BadSizeException if w != 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` 
  - 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
```

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
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

Box is a BallContainer?

```java
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);
}
```

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

Exercise: Reason About Specs

```java
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. More on Java overriding shortly

- 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!

Properties Class from the JDK

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

```java
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) {
        super.put(key, value);
    }
    public String get(String key) {
        return (String) super.get(key);
    }
}
```

Exercise

```java
class HasTable {
    public void put(Object key, Object value);
    public Object get(Object key);
}
class Properties extends HasTable {
    public void put(String key, String value) {
        super.put(key, value);
    }
    public String get(String key) {
        return (String) super.get(key);
    }
}
```
Exercise

```java
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;
}
```

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)

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, and 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

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

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(Object o)

Exercise

class Y extends X { … }
class A {
    A a = new B();
    Object o = new Object();
}
class X m(Object o) { … } X x = a.m(o);
}
class B extends A {
    X m(Z z) { … } A a = new C();
}
class C extends B {
    Y m(Z z) { … } Object o = new Z();
}
class A a = new A();
Object o = new Object();
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();
X x = a.m(w);

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()

Next time

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