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

Announcements
- HW5 out, due Tuesday October 30th
  - Part 1: Questions on material we’ll cover today
  - Part 2: BFS using your graph from HW4
- HW4 resubmit due Friday October 27th 2pm
  - A New Submitty page!!!
  - Auto-graded part of HW4 will be the average of the two submissions
  - PULL hw5 before you COMMIT/PUSH!!!
- Quiz 6 today

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?
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

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.

“What Science” of software design teaches Design Patterns

- Design patterns promote design for extensibility and reuse
- Nearly all design patterns make use of subtype polymorphism

Examples of Subtypes
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!

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

```java
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

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

Code cloning is a bad idea! Why?

```java
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?
  ```java
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?

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

Every Square is a Rectangle, right?

```java
class Square extends Rectangle { … }
    // requires: w = h
    // effects: this.width=w, this.height=h
    Choice 1: public void setSize(int w, int h);
    // effects: this.width=w, this.height=h
    Choice 2: public void setSize(int w, int h);
    // effects: this.width=s, this.height=s
    Choice 3: public void setSize(int s);
    // effects: this.width=w, this.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
  - 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?

- 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.set_width(w), this.set_height(h)
    public void setSize(int w, int h);
}

class Square extends Rectangle {
    // requires: w = h
    // effects: this.set_width(w), this.set_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