Supplementary Material

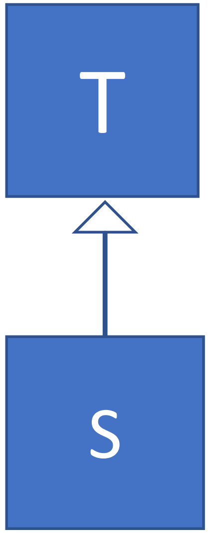
Lecture 6

Maria Zontak

SOLID Object Oriented Design

- **S**RP: Single Responsibility Principle
  A class should have only a single responsibility
  (easy check: only one potential change in the software's specification should be able to affect the specification of the class)
- **O**CP: Open/Closed Principle
  Software entities … should be open for extension, but closed for modification
  (easy check: method implementation should NOT change when new types are added).
- **L**SP: Liskov Substitution Principle
  Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- **I**SP: Interface Segregation Principle
  Many client-specific interfaces are better than one general-purpose interface
- **D**IP: Dependency Inversion Principle
  One should "Depend upon Abstractions. Do not depend upon concretions."

Liskov Substitution Principle



- Defines a notion of substitutability for objects:

If *S* is a subtype of *T*, then objects of type *T* in a program may be replaced with objects of type *S* without altering any of the desirable properties of that program

→

- Preconditions cannot be strengthened in S (S will replace T, hence should work with input constraints that suited T or less)
- Postconditions cannot be weakened in a S (S will replace T, hence should yield a result as safe as the result of T or more)
- Invariants of T must be preserved in S (everything should proceed without any change)

Violation of Liskov Substitution Principle – Example:

```
From Effective Java, Second Edition, Chapter 3
// Broken - violates Liskov substitution principle (page 40)
@Override
public boolean equals(Object o) {
      if (o == null || o.getClass() != getClass()) return false;
      Point p = (Point) o;
      return p.x == x && p.y == y;
}
```

Violation of Liskov Substitution Principle – Illustration

We want to write a method to tell whether an integer point is on the unit circle.

Here is one way we could do it:

```
// Initialize UnitCircle to contain all Points on the unit circle
private static final Set<Point> unitCircle;
static {
        unitCircle = new HashSet<Point>();
        unitCircle.add(new Point( 1, 0));
        unitCircle.add(new Point( 0, 1));
        unitCircle.add(new Point(-1, 0));
        unitCircle.add(new Point( 0, -1));
}
public static boolean onUnitCircle(Point p) {
        return unitCircle.contains(p);
}
```

Violation of Liskov Substitution Principle – Illustration

Suppose you extend `Point` in some trivial way that does NOT add a value component.

For example, by having its constructor keep track of how many instances have been created:

```
public class CounterPoint extends Point {
	private static final AtomicInteger counter =new AtomicInteger();
	public CounterPoint(int x, int y) {
		super(x, y);
		counter.incrementAndGet();
	}
	public int numberCreated() { return counter.get(); }
}
```

Violation of Liskov Substitution Principle – Illustration

- The *Liskov substitution principle* says that any important property of a type should also hold for its subtypes, so that any method written for the type should work equally well on its subtypes
- Suppose we pass a `Counter-Point` instance to the `onUnitCircle` method. If the Point class uses a `getClass()` based `equals` method, the `onUnitCircle` method will return false regardless of the `CounterPoint` instance's x and y values. This is so because collections, such as the `HashSet` used by the `onUnitCircle` method, use the `equals` method to test for containment, and no `CounterPoint` instance is equal to any `Point`.
- A proper `instanceof` - based `equals` method on `Point`, the same `onUnitCircle` method will work fine when presented with a `CounterPoint`. (Note however that `instanceof` - based `equals` violates contract of [`equals`]
- While there is no satisfactory way to extend an instantiable class and add a value component, there is a fine workaround. Follow the advice of Item 16, "Favorcomposition over inheritance."

Static Binding or Early Binding

- **Compile- Time/ Static Binding** - Type of the object is determined at compiled time (by the compiler)
- **Overloading follows Compile Time Binding** (see discussion on Abstracting Visitor in Design Patterns lecture notes)
- **Binding of private, static and final methods always happen at compile time** since these methods cannot be overridden (overriding yields dynamic binding)
- Another interesting example: **static binding happens when `super` is used**

The `super` keyword allows to access a superclass's methods and fields from a subclass, even if they are overridden in the subclass. **In the case of instance methods, static binding must be used.** If a method is overridden in a subclass, dynamic binding would cause the subclass's version of the method to be invoked rather than the superclass's version. When the method is invoked with `super,` **the compiler knows precisely which class contains the method to invoke. Static binding allows a superclass's version of an instance method to be invoked independent of the actual class of the object at run-time.**

- The Java compiler creates one instance initialization method for each constructor in the source for a class. This special kind of instance method is invoked only when an object is created. Like private methods and methods invoked with super, **instance initialization methods are invoked using static binding.**