LSP中关于subtyping

Behavioral subtyping

- **Compiler-enforced rules in Java (static type checking)**
  - Subtypes can add, but not remove methods 子类型可以增加方法，但不可删
  - Concrete class must implement all undefined methods 子类型需要实现抽象类型中的所有未实现方法
  - Overriding method must return same type or subtype 子类型中重写的方法必须有相同类型的返回值或者符合co-variance的返回值
  - Overriding method must accept the same parameter types 子类型中重写的方法必须使用同样类型的参数或者符合contra-variance的参数
  - Overriding method may not throw additional exceptions 子类型中重写的方法不能抛出额外的异常，抛出相同或者符合co-variance的异常
- **Also applies to specified behavior (methods):**
  - Same or stronger invariants 更强的不变量
  - Same or weaker preconditions 更弱的前置条件
  - Same or stronger postconditions 更强的后置条件

Liskov Substitution Principle (LSP)

- **LSP is a particular definition of a subtyping relation, called (strong) behavioral subtyping 强行为子类型化**
- **In programming languages, LSP is relied on the following restrictions:**
  - Preconditions cannot be strengthened in a subtype. 前置条件不能强化
  - Postconditions cannot be weakened in a subtype. 后置条件不能弱化
  - Invariants of the supertype must be preserved in a subtype. 不变量要保持
  - Contravariance of method arguments in a subtype 子类型方法参数：逆变
  - Covariance of return types in a subtype. 子类型方法的返回值：协变
  - No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype. 异常类型：协变

> covariance 父类型--->子类型，变得越来越具体或不变
>
> contra-variance 父类型--->子类型，变得越来越具体，或不变

Covariance (协变)

父类型→子类型：越来越具体specific
返回值类型：不变或变得更具体
异常的类型：也是如此。

- **See this example:**

```
class T {
  Object a() { … }
}

class S extends T {
  @Override
  String a() { … }
}
```

- **More specific classes may have more specific return types**
- **This is called covariance of return types in the subtype.**

```
class T {
  void b( ) throws Throwable {…}
}

class S extends T {
  @Override
  void b( ) throws IOException {…}
}

class U extends S {
  @Override
  void b( ) {…}
}
```

- **Every exception declared for the subtype's method should be a subtype of some exception declared for the supertype's method.**

Contravariance (反协变、逆变)

- **What do you think of this code?**

```
class T {
  void c( String s ) { … }
}

class S extends T {
  @Override
  void c( Object s ) { … }
}
```

父类型→子类型：越来越具体specific
参数类型：要相反的变化，要不变或越来越抽象

- **Logically, it is called contravariance of method arguments in the subtype.**
- **This is actually not allowed in Java, as it would complicate the overloading rules. 目前Java中遇到这种情况，当作overload看待 ☹**

The method `c(Object)` of type S must override or implement a supertype method

Arrays are covariant

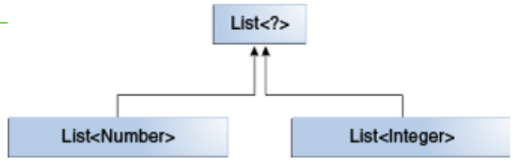
Generics are not covariant

Wildcards --> ?

- `ArrayList<String>` is a subtype of `List<String>`
- `List<String>` is not a subtype of `List<Object>`

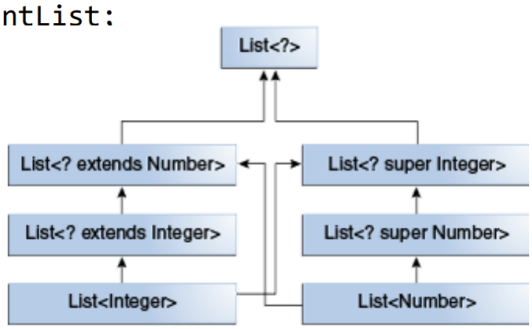
Consider LSP for generics with wildcards

- `List<Number>` is a subtype of `List<?>`



- `List<Number>` is a subtype of `List<? extends Object>`
- `List<Object>` is a subtype of `List<? super String>`

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number>  numList = intList:
```



delegation

委派模式：通过运行时动态绑定，实现对 其他类中代码的动态复用

一个类不需要继承另一个类的全部方法，通过委托机制调用部分方法

Composite over inheritance principle

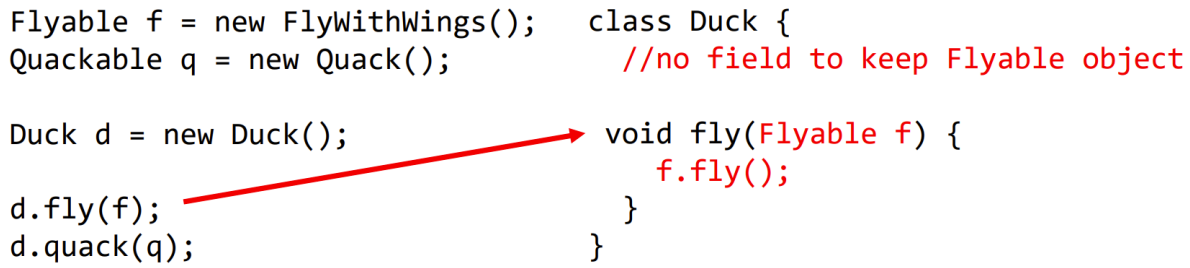
- **Or called Composite Reuse Principle (CRP)**
  - Classes should achieve polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement desired functionality) rather than inheritance from a base or parent class.
  - It is better to compose what an object can do (`has_a or use_a`) than extend what it is (`is_a`).

- **Delegation can be seen as a reuse mechanism at the object level, while inheritance is a reuse mechanism at the class level. “委托”发生在object层面，而“继承”发生在class层面**

- **CRP原则更倾向于使用委派而不是继承来实现复用。**

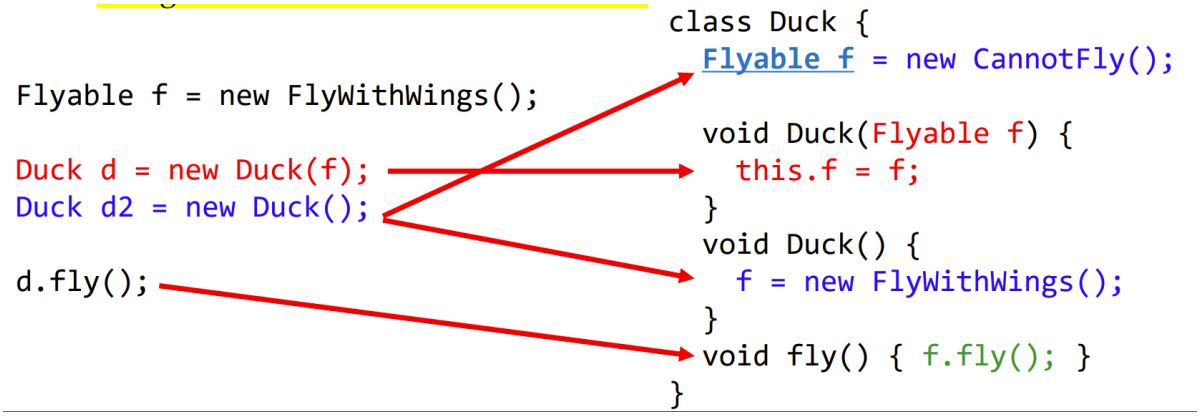
Dependency: 临时性的delegation

通过传参的方式委托



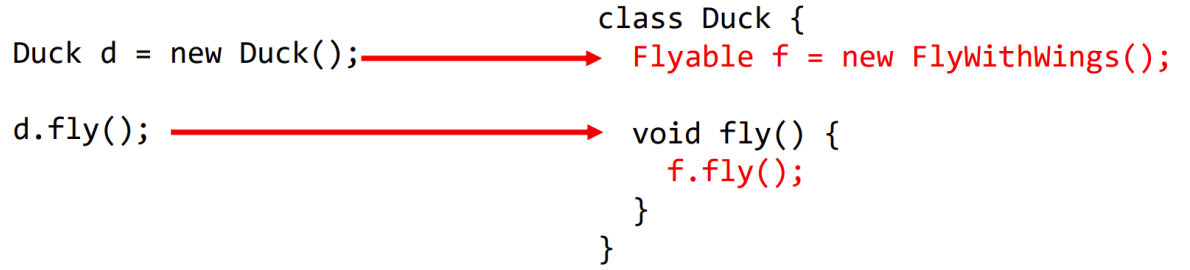
Association: 永久性的delegation

通过在类中添加属性



Composition: 更强的association，但难以变化

Composition是Association的一种特殊类型，其中Delegation关系通过类内部field初始化建立起来，无法修改。已经给你指定运行类型了



Aggregation: 更弱的association，可动态变化

Aggregation也是Association的一种特殊类型，其中Delegation关系通过客 户端调用构造函数或专门方法建立起来。可以通过方法改变属性的运行类型