

Exploring the design space of object-oriented language

- **Review**
 - Simula-style vs. Smalltalk's actors model
 - Class-based vs. prototype-based
 - Inheritance
 - Contravariance and covariance
 - **Inheritance (continued)**
 - Contravariance and covariance
 - Covariance: Child's method's parameters less general
 - Contravariance: Child's method's parameters more general
 - Which is better? Depends on why we're using inheritance.
 - Two reasons to use inheritance:
 - Code reuse. Parent already defines part of the behavior we want, so we subclass it and implement the rest in the subclass.

Example: Someone has already written code to check whether the capacity of a car has been exceeded. We want to do this for all kinds of vehicles, and want to reuse the work that has already been done for cars.

```

Bool Car::capacityExceeded( Passenger )
Bool Vehicle::capacityExceeded( Payload )

```
 - Contravariance helps reuse, but breaks inclusion polymorphism.
 - Code interface. Want to capture that the subclass "is-a" parent class. The parent defines something general, and we want to provide an implementation for a specific case.

Example: Someone has already defined a class for Vehicle and a function capacityExceeded(Payload). We want to implement this check for the specific case where the vehicle is a car and the payload is a passenger.

```

Bool Vehicle::capacityExceeded( Payload )
Bool Car::capacityExceeded( Passenger )

```
 - Covariance helps define interfaces in terms of a class hierarchy for inclusion polymorphism, but limits reuse.
 - Java's interfaces
 - What is an interface? Like a class, but:
 - No instance variables
 - If you declare member variables, they are implicitly made static
 - Constant-valued variables only
 - No method bodies
 - Why interfaces?
 - Remember all the problems we ran into with multiple inheritance?
 -
- What does Java use for parameters? Contravariance of covariance?

- Example 1 (covariance):

```
interface Vehicle {
    String manifest(Payload);
}
```

```
class Car implements Vehicle {
    public String manifest(Passenger p) { ... } // not enough
    public String manifest(Payload p) { ... } // need this
}
```

- Example 2 (contravariance):

```
interface Vehicle {
    String manifest(Passenger);
}
```

```
class Van implements Vehicle {
    public String manifest(Payload p) { ... } // not enough
    public String manifest(Passenger p) { ... } // need this
}
```

- Clearly not covariance or contravariance
- Java requires that the signature matches exactly.
- What if we have both methods in Example 1?

```
class Main {
    public static void Main(String args[]) {
        Car car = new Car();
        Payload l1 = new Passenger(18);
        System.out.println(car.manifest(l1));
        Passenger l2 = new Passenger(21);
        System.out.println(car.passenger(l2));
    }
}
```

- First call goes to String manifest(Payload).
- Second call goes to String manifest(Passenger).
- This should be surprising. Doesn't Java have inclusion polymorphism?
- In fact this is overloading (an ad-hoc polymorphism technique).
- The method is chosen from the static context of the call.
- Dynamic dispatch (inclusion polymorphism) only on the object that is being called.
- **Single vs. multiple dispatch**
 - As just discussed, Java has single dispatch.
 - C++ and Smalltalk also use single dispatch.
 - Dynamic dispatch (inclusion polymorphism) only on the object that the method is being called on.
 - But the object that the method is being called on is really just a special, implicit first parameter "self".
 - Multiple dispatch (multimethods): Dynamic dispatch on all parameters.
 - Languages with multiple dispatch: Common Lisp, Dylan, Nice, Slate, Cecil.

- Languages that support multiple dispatch via extensions:
 - Scheme, Python, Perl, Java, Ruby
- With multiple dispatch, it doesn't make sense to say that a method belongs more to one class than to another.
 - Having the method belong to one class makes sense in Smalltalk because it can only modify the state of one object. It can affect its parameters only by sending messages to them.
 - Multimethods make sense when the method affects both parameters
 - Example: start-working-together(advisor, student)
- Next: How do multimethods affect encapsulation?