OO subtyping

- **A type SubA may only be a subtype of A if every instance of SubA can be safely substituted for any instance of A.**
  - SubA instances must handle all the same messages as instances of A
  - SubA methods must return results usable by any client of A

More specifically:

- Given an object O pointed to by a reference of type A...
  - O's methods must **return** a type at least as **specific** as the return type of A's corresponding methods.
  - O's methods must take **parameters** at least as **general** as the parameters of A's corresponding methods.

What should be subtypes?

```
class Fruit { ... }
class Apple extends Fruit { ... }
class Orange extends Fruit { ... }

class FruitPlant
  {  Fruit produce() { ... }  }
class ApplePlant
  {  Apple produce() { ... }  }

class FruitFly
  { void eat(Fruit f) { ... } }
class AppleFly
  { void eat(Apple a) { ... } }
```

Assignments and subtyping

- A reference may refer to any instance of a class, *or any instance of its subclasses*.
- Hence, it is always legal to assign "up" the heirarchy---a subclass instance may be assigned to a superclass reference.
  - Implies: may pass a subtype as a parameter where its supertype is required
  - Implies: may return a subtype when its supertype is required

Which should be statically legal?

```
FruitPlant fp = new FruitPlant();           //  1
ApplePlant ap = new ApplePlant();           //  2

FruitPlant fp2 = ap;                        //  3
ApplePlant ap2 = fp;                        //  4
ApplePlant ap3 = fp2;                       //  5

FruitEatingFly ffly = new FruitEatingFly(); //  6
ffly.eat(fp.produce());                     //  7
ffly.eat(ap.produce());                     //  8

AppleEatingFly afly = new AppleEatingFly(); //  9
afly.eat(fp.produce());                     // 10
afly.eat(ap.produce());                     // 11
```

Java overriding and subtyping rules: more restrictive

- “Natural overriding”: overriding methods may have more specific return type, and more general parameters
- Java: overriding methods must have **exactly the same** return and parameter types
- Changing parameter types **overloads** method instead of overriding

Translating flies...

```
class AppleEatingFly
{   void eat(Apple a) { ... }
}

class FruitEatingFly
     extends AppleEatingFly
{   void eat(Apple a) { ... }
    void eat(Fruit f) { ... }
}
```

What about return types?

```
// OK in Java
class FruitPlant
  {  Fruit produce() { ... }  }
class ApplePlant extends FruitPlant
  {  Fruit produce() { ... }  }

// Not OK---cannot overload on return type!
class FruitPlant
  {  Fruit produce() { ... }  }
class ApplePlant extends FruitPlant
  {  Fruit produce() { ... }
     Apple produce() { ... } }
```

Overriding vs. overloading

- Overriding: subclasses may define a different method to be invoked for a runtime message
  - (Dynamic dispatch on receiver type)

- Overloading allows classes to define different methods of the same name.
  - (Static overload resolution: messages are completely different!)

Which should be legal? Which methods are invoked?

```
FruitEatingFly ffly = new FruitEatingFly();
AppleEatingFly afly = ffly;
Apple appleRef = new Apple();
Fruit fruitRef = anApple;

ffly.eat(appleRef);                     //  1
ffly.eat(fruitRef);                     //  2
afly.eat(appleRef);                     //  3
afly.eat(fruitRef);                     //  4
```

What’s wrong?

```
abstract class AppleEater {
  abstract void eat(Apple a);
}

class FruitEatingFly extends AppleEater {
  void eat(Fruit f) { ... }
}
```

“Generic functions”

- Generic function = function that contains several methods
  - when a GF is called, dynamically select method based on runtime type of receiver
- Java places methods into GFs by name and exact matches on argument types

Generic functions, ct’d.

		Receiver classes/methods	
		AppleEatingFly	FruitEatingFly
Generic functions	eat(Fruit f)	--(does not exist)--	FruitEatingFly:: eat(Fruit f)
	eat(Apple a)	AppleEatingFly:: eat(Apple a)	FruitEatingFly:: eat(Apple a)

Logic/constraint programming

```
abs(X,A) :- X >= 0, X = A.  /* a CLP(R) "relation */
abs(X,A) :- X < 0, -X = A.

?- abs(1, X).  /* CLP(R) query */
X = 1
*** Yes

?- abs(Y, 2).  /* Another query; notice that it goes */
Y = 2          /* in the opposite "direction".       */
Y = -2
*** Yes
```