Higher-Order Functions in Java

Functions as Objects?

- In Java, neither static methods nor instance methods are considered to be objects.

- How, then, could we implement something that behaves as a function taking functions as arguments, or one giving a function as a result?

Dynamic Creation

- In Java, the only things that can be created dynamically are:
  - Objects
  - Arrays
  - and arrays behave as if a special built-in object.

Function Objects

- We can define objects that **behave as** functions:
  - function world:

    f(x, y)

  - object world:

    f.apply(x, y)
- Define a **class** for these objects: Function1, Function2, … depending on the number of arguments.

Example: map

In rex:

- map(f, []) => [];
- map(f, [A | L]) => [f(A) | map(f, L)];

Example: OpenList map

In Java:

- static OpenList map(Function1 f, OpenList L)

```
{
if( L.isEmpty() )
  return nil;
else
  return cons(f.apply(L.first()),
                    map(f, L.rest());
}
```

Specific case: square each element

- class Function1

```
{
Object apply(Object x)
 {
 float num = ((Float)x).floatValue();
 return new Float(num*num);
 }
}
```

Slight problem:

- We'd need a whole *different* class for each function to be applied.
- We'd also need a different map for each of those functions.
- So little would be gained over ad hoc definitions of map-like functions.

Remedy:

- We need a way to define ***one*** map, yet use it for lots of different Function classes.

- A device that works for this is Java's **interface** concept

Java Interfaces

- An **interface** is like an abstract place-holder class.
- A single interface can stand for an arbitrary number of classes that have certain methods in common.
- The common methods are defined in the interface.
- The various special classes are said to **implement** the interface.

Java Keywords

- **interface**: used in place of "class" for an interface definition

- **implements**: added to class for an interface implementation

map arguments, the right way

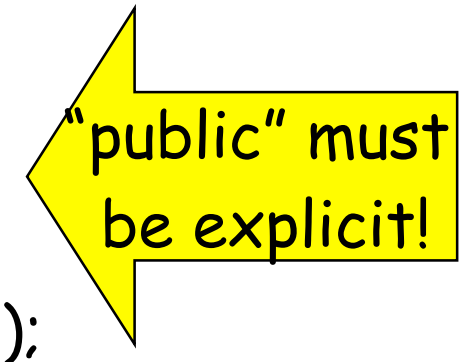
- interface Function1
  {
  Object apply(Object x);
  }

- An interface does not implement methods; it only declares them.

- All methods of an interface are **implicitly** public.

A map argument

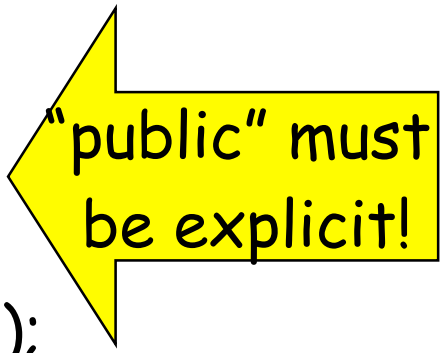
```
class Squarer implements Function1
    {
     public Object apply(Object x)
      {
      float num = ((Number)x).floatValue();
      return new Float(num*num);
      }
   }
```



Usage: map(new Squarer(), L)

Another map argument

```
class Cuber implements Function1
    {
     public Object apply(Object x)
      {
      float num = ((Number)x). floatValue();
      return new Float(num*num*num);
      }
   }
```

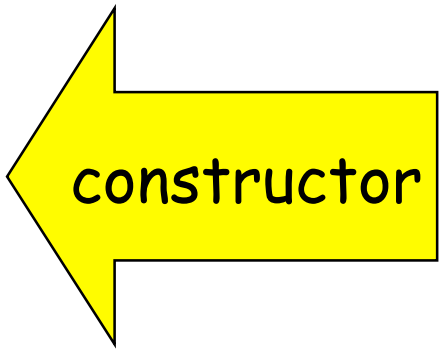


Usage: map(new Cuber(), L)

Yet another

```
class Scaler implements Function1
      {
      private float factor;
      Scaler(float _factor) { factor = _factor;}

     public Object apply(Object x)
        {
        float num = ((Number)x). floatValue();
        return new Float(factor*num);
        }
   }
```



Usage: map(new Scaler(2.5), L)

Number vs. Float and Integer

- **Number** is a class that generalizes **Float** and **Integer**.
- Technically, **Float** and **Integer** "inherit" from **Number**.
- This is often done when classes have a lot of methods in common.
- **Object** is the class that generalizes all other classes.

Restrictions on interfaces

- All methods are implicitly public.
- No static methods are allowed.
- No static constants are allowed.
- Implementations of interfaces can include other methods and constants not mentioned in the interface itself.
- A given class can implement multiple interfaces.

Exercise

- Similar to map, develop the java representation of reduce.

- In rex:

```
reduce(b, u, []) => u;
reduce(b, u, [A | L]) => b(A, reduce(b, u, L));
```