Lecture 1

We see another example of a Java interface. We use it to build a polymorphic data structure that is processed using dynamic method binding.

But the example helps to show a case where Java interfaces win out over inheritance.

In our example, there will be a class hierarchy. But it will not give us a superclass that we could use as the basis of our polymorphic data structure. By using a Java interface, we can ‘cut across’ the class hierarchy, and get the polymorphic data structure that we need.

The class hierarchy for some company

```
Object
   |
   +----Building
   |       |
   |       +----Shop
   |       |
   |       +----Warehouse
   |
   +----Employee
   |       |
   |       +----SalesStaff
   |       |
   |       +----WarehouseStaff
   |       |
   |       +----AdminStaff
   |
   +----Vehicle
   |       |
   |       +----CompanyCar
   |       |
   |       +----DeliveryVan
   |
   +----OfficeEquipment
           |
           +----Typewriter
           |
           +----PC
           |
           +----Photocopier
```


Taxable assets

- Suppose buildings, vehicles, PCs and photocopiers are valuable assets, and you have to pay some tax each year, depending on how valuable they are.
- Suppose the class definitions for `Building`, `Vehicle`, `PC` and `Photocopier` contain an instance method `getTaxableValue`

```
public class Building
{
   public double getTaxableValue()
   {  return floorSpace * landRate;
   }

   protected double floorSpace;
   protected double landRate;
   protected String address;
}
```



```
public class Vehicle
{  public double getTaxableValue()
   {  return purchasePrice - depreciation;
   }
   protected double purchasePrice;
   protected double depreciation;
   protected String registrationNum;
}

public class PC
{  public double getTaxableValue()
   {  return purchasePrice / age;
   }
   protected double purchasePrice;
   protected int age;
}

public class Photocopier
{  public double getTaxableValue()
   {  return SCRAP_VALUE;
   }
   private static final double
                              SCRAP_VALUE = 10;
}
```


Computing the value of the company’s assets

- We need a polymorphic data structure that can contain `Building`, `Vehicle`, `PC` and `Photocopier` objects

```
Set assets = new HashSet();
assets.add(new Shop(...));
assets.add(new CompanyCar(...));
assets.add(new Shop(...));
assets.add(new Photocopier(...));

double totalAssetValue = 0;
Iterator iter = assets.iterator();
while (iter.hasNext())
{  Object asset = iter.next();
   totalAssetValue +=
            asset.getTaxableValue();
}
```

Won’t compile. Why?

A redesigned but unnatural class hierarchy

One solution is to redesign the class hierarchy.

```
Object
   |
   +----TaxableThing
   |       |
   |       +----Building
   |       |       |
   |       |       +----Shop
   |       |       +----Warehouse
   |       |
   |       +----Vehicle
   |       |       |
   |       |       +----CompanyCar
   |       |       +----DeliveryVan
   |       |
   |       +----TaxableOfficeEquipment
   |               |
   |               +----PC
   |               +----Photocopier
   |
   +----NonTaxableThing
           |
           +----Employee
           |       |
           |       +----SalesStaff
           |       +----WarehouseStaff
           |       +----AdminStaff
           |
           +----NonTaxableOfficeEquipment
                   |
                   +----Typewriter
```

Using the new hierarchy

- Now `Building` `Vehicle`, `PC` and `Photocopier` have a superclass which we can use in casting:

```
Set assets = new HashSet();
assets.add(new Shop(...));
assets.add(new CompanyCar(...));
assets.add(new Shop(...));
assets.add(new Photocopier(...));

double totalAssetValue = 0;
Iterator iter = assets.iterator();
while (iter.hasNext())
{  TaxableThing asset =
       (TaxableThing) iter.next();
   totalAssetValue +=
            asset.getTaxableValue();
}
```

This works but why is it unsatisfactory?

Cutting across the class hierarchy

- Let's go back to the original hierarchy and define a Java interface:

```
public interface ITaxable
{
   public double getTaxableValue();
}
```

- Remember, Java interfaces act like specifications. This one specifies what it means to be a taxable object: you're a taxable object if you contain a `getTaxableValue` method

- Now, in `Building`, `Vehicle`, `PC`, and `Photocopier` declare that they implement the interface, i.e. they meet the specification; they are taxable things

Computing the value of the company's assets

- Now when we take items out of the polymorphic data structure, we cast them to `ITaxable`:

```
Set assets = new HashSet();
assets.add(new Shop(...));
assets.add(new CompanyCar(...));
assets.add(new Shop(...));
assets.add(new Photocopier(...));

double totalAssetValue = 0;
Iterator iter = assets.iterator();
while (iter.hasNext())
{  ITaxable asset =
       (ITaxable) iter.next();
   totalAssetValue +=
            asset.getTaxableValue();
}
```


When to use Java interfaces

- Define an interface when actual or potential classes
  - overlap in the services (methods) they offer
  - but don't share implementations for those methods (so inheritance doesn't help)
  - and aren't easily related by a class hierarchy.
- Use polymorphic interface variables in client code whenever you anticipate change:
  - data structures (changed for performance)
  - end-user interfaces (changed for usability)
  - platform-specific (changed for portability)
  - ill-understood subsystems (changed when understanding grows)

When to use class hierarchies

- Define class B to be a subclass of class A when
  - "Every B is a special kind of A"
  - B inherits from A, overrides very little of what it inherits, and extends A with additional things
- Superclass A might be abstract if
  - all subclasses must offer a method but the superclass can't provide a suitable default implementation
  - you don't need to instantiate objects from A, only from the subclasses
- Use polymorphic superclass variables in client code when you want to make the code more generic