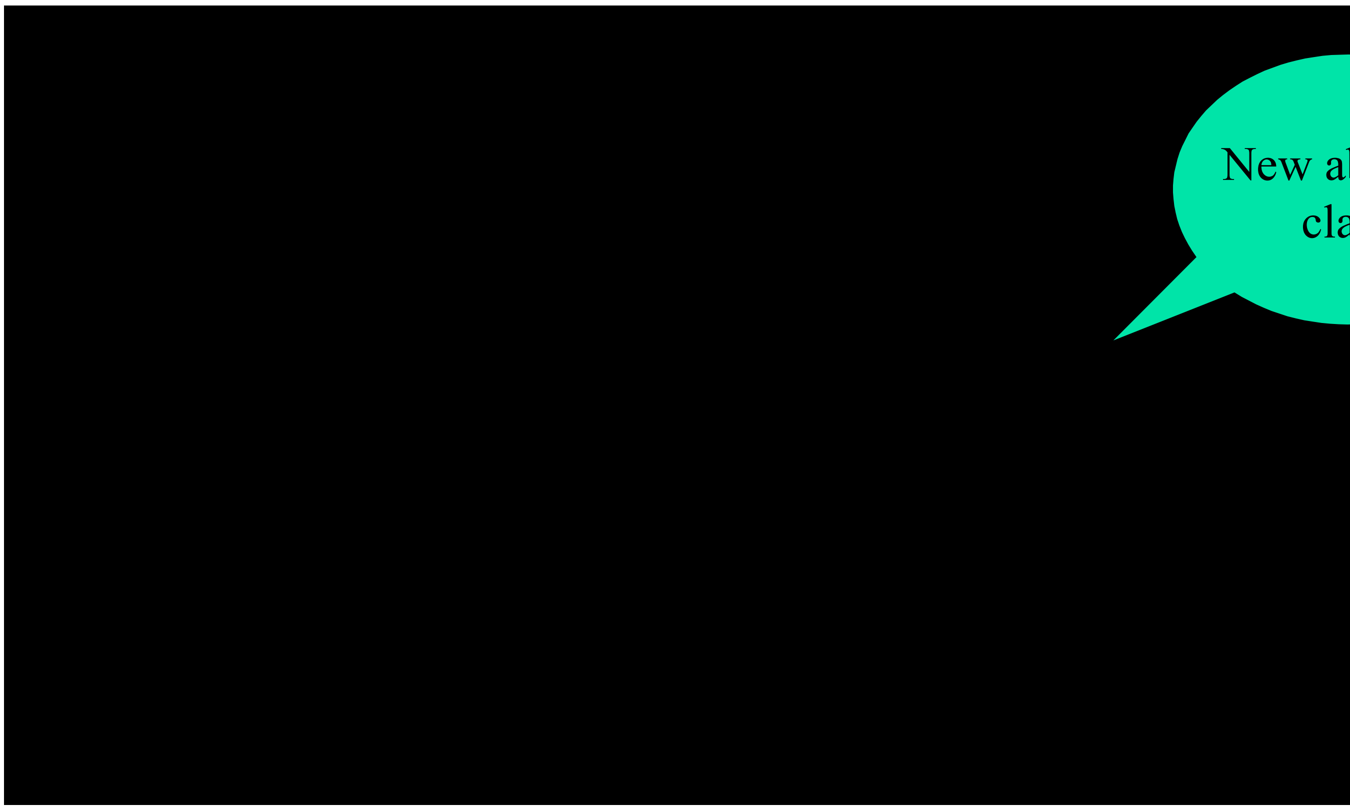
Defining Inductive Data in Java

Corky Cartwright
Department of Computer Science
Rice University

Partial Hoisting

- In a union hierarchy, the same code may be repeated in *some proper subset* of the variants.
- We can eliminate this code duplication by introducing a new abstract class that is a *superclass only of the variants that repeat the same code.*
- Partial hoisting modifies the form of the class diagram because it introduces a new abstract class below the *root* abstract class of the union.

Revised Class Hierarchy



Revised Code for `CityEntry`

```
abstract class CityEntry {
 /* common fields */
 String name, address, phone;

 /** Returns true if key is a prefix of name. */
 boolean nameStartsWith(String key) { return name.startsWith(key); }
}

class ResidentialEntry extends CityEntry { }

abstract class NonResidentialEntry extends CityEntry {
  String city, state;
}
class BusinessEntry extends NonResidentialEntry { }

class GovernmentEntry extends NonResidentialEntry {
  String government;
}
```

Defining `DeptDirectory` Data

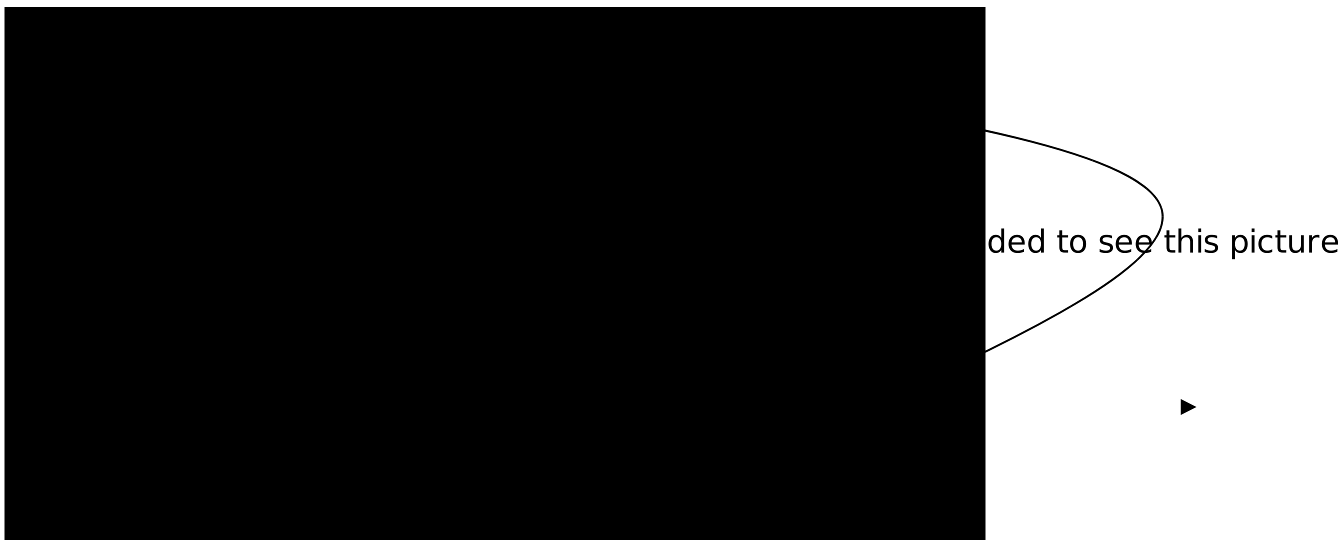
A `DeptDirectory` is either:

- `Empty()`, the empty `DeptDirectory`, or
- `NonEmpty(first,rest)`, a non-empty `DeptDirectory`, where `first` is an `Entry` and `rest` is a `DeptDirectory`

Examples:

```
Empty()
NonEmpty(Entry("Stephen","DH 3103","x
3846"), Empty())
```

Class Diagram for DeptDirectory



DeptDirectory

```
abstract class DeptDirectory {}

class Empty extends DeptDirectory {}

class Cons extends DeptDirectory {
  Entry first;
  DeptDirectory rest;
}
```

The Composite Pattern

The **`DeptDirectory`** class hierarchy is a special form of union called a *composite*. A union is composite when one or more variants contain fields of the root class type. In other words, variants in the union contain references to objects of union type. In the **`DeptDirectory`** example, the **`Cons`** variant contains a field **`rest`** of type **`DeptDirectory`**.

Searching a `DeptDirectory`

Given the name of a person, we want to find that person's phone number in our **`DeptDirectory`**. In other words, we want to define a method in the **`DeptDirectory`** class

```
findPhone(String keyName)
```

that searches "this" **`DeptDirectory`** for an **`Entry`** that matches the argument **`keyName`**.

The Interpreter Pattern

- To define a method **`m`** on a composite class like **`DeptDirectory`**, we follow the same process as we would in defining a method on a union class, with one new wrinkle. In the variants that refer to the composite class (have fields of composite class type), computing **`m`** for embedded self references will usually involve delegating the task of computing **m** to the (abstract) composite class.
- In our **`DeptDirectory`** example, the only embedded reference to **`DeptDirectory`** in variant subclasses is the **`rest`** field in **`Cons`**.

Template for coding `findPhone`

```
abstract class DeptDirectory {
  abstract String findPhone(String key);
}

class Empty extends DeptDirectory {
  String findPhone(String key) { … }
}

class Cons extends DeptDirectory {
  Entry first;
  DeptDirectory rest;

  String findPhone(String key) {
  … rest.findPhone(key) …
  }
}
```

Fill in this method body

Fill in this method body

findPhone

```
abstract class DeptDirectory {
  abstract String findPhone(String key);
}

class Empty extends DeptDirectory {
  String findPhone(String key) { return null; }
}
class Cons extends DeptDirectory {
  Entry first;
  DeptDirectory rest;

  String findPhone(String key) {
    if (key.equals(first.getName())) return first.getPhone();
    else return rest.findPhone(key);
  }
}
```

Another Composite Example: `Lists`

An `IntList` is either:

- `EmptyIntList()`, the empty list, or
- `ConsIntList(first,rest)`, a non-empty list, where `first` is an `int` and `rest` is an `IntList`.

Some examples include:

```
EmptyIntList()
ConsIntList(7, EmptyIntList())
ConsIntList(12, ConsIntList(7, EmptyIntList()))
```

IntList

```
abstract class IntList { }

class EmptyIntList extends IntList { }

class ConsIntList extends IntList {
  int first;
  IntList rest;
}
```

Defining Methods on `IntList`

Sort example:

```
abstract class IntList {
  abstract IntList sort() { }
}
class EmptyIntList extends IntList {
  IntList sort() { ... }
}
class ConsIntList extends IntList {
  int first;
  IntList rest;
  IntList sort() { ... }
}
```

`IntList sort` cont.

```
abstract class IntList {
  abstract IntList sort() { }
  abstract IntList insert(int i) { }
}
class EmptyIntList extends IntList {
  IntList sort() { return this; }
  IntList insert(int i) { return new ConsIntList(i, this); }
}
class ConsIntList extends IntList {
  int first;
  IntList rest;
  IntList sort() { return rest.insert(first); }
  IntList insert(int i) {
    if (i <= first) return new ConsIntList(i, this);
    else return new ConsIntList(first, rest.insert(i));
}
```

For Next Class

- HW6 (extra credit) Due Monday 11:59pm.
- HW7 due next Friday
- Reading: OO Design Notes, 1.5.