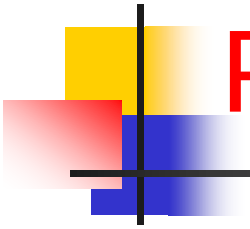
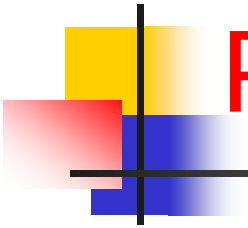
Polymorphism and Interfaces

Corky Cartwright
Department of Computer Science
Rice University



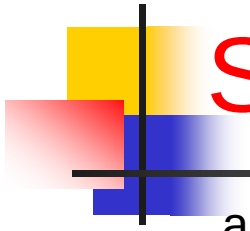
Polymorphism

- In Scheme, we defined a multitude a different kinds of lists: **list-of-numbers**, **list-of-symbols** , **list-of-list-of-numbers** , etc. before we concluded that we could abstract over the element type **T** in lists and write a single parametric definition for lists **(listOf T).**
- We can do the same thing in Java.
- Prior to Java 5.0, Java had no parameterized types other than arrays. We will subsequently study parameterized (generic) types in Java, but that is a more advanced topic that is not supported by DrJava language levels.



Polymorphism cont.

- Data definitions that are implicitly or explicitly parameterized by some component types are called *polymorphic* (*generic*) data definitions.
- We can convert our definition of **IntList** to implicitly polymorphic form by allowing the elements to be arbitrary objects. Let us call the resulting class **ObjectList**.
- But we cannot support methods like **sort** or **insert** on such a type because the **Object** has no natural ordering.
- Let's write a minimalist definition of **ObjectList**.



Singleton Composite **ObjectList**

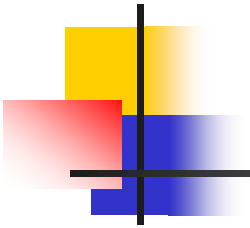
```
abstract class ObjectList {
  ObjectList cons(Object n) {
    return new ConsObjectList(n, this);
  }
}

class EmptyObjectList extends ObjectList {
  static EmptyObjectList ONLY = new EmptyObjectList();
  private EmptyObjectList() { }
}

class ConsObjectList extends ObjectList {
  Object first;
  ObjectList rest;
}
```

Defining Implicitly Polymorphic Methods on Lists

- We can easily add methods like concat and reverse to ObjectList
- To sort lists of objects, we need for the objects to support some notion of comparison. How can identify such objects as a type? Object does not work.
- Answer: we need a mechanism for talking about all objects that support the method:

```
int compareTo(Object other);
```

- How can we identify such a type? Java includes a special facility for defining such types called *interfaces*.

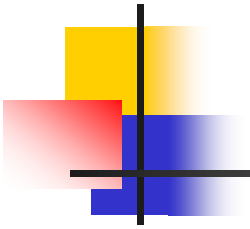


Java Interfaces

In Java, an **interface** is a language construct that resembles a "lightweight" abstract class (an abstract class with no concrete methods). An **interface** definition has the syntax

```
interface <name> {
<members>
}
```

which looks exactly like a class definition except for the use of the keyword **interface** instead of **class**. But the members of an **interface** are restricted to **abstract** methods and **static** fields.



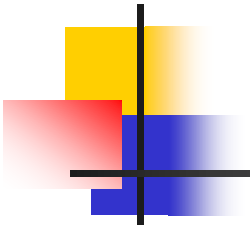
Examples

- The interface **Comparable**, which is built-in to Java (part of the core library **java.lang**) has the following definition

```
interface Comparable {
    int compareTo(Object other);
  }
```

The value returned by **compareTo** is negative, zero, or positive depending on whether **this** is less than **other**, equal to **other**, or greater than **other**.

- The built-in class **String** also implements the interface **CharSequence** which includes methods such as **int length()**. The built-in classes **StringBuffer** and **StringBuilder** (mutable strings) also implements this interface.

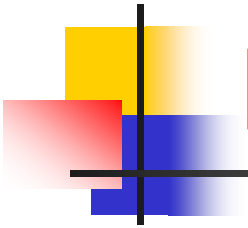


Key Properties of Interfaces

- A class can implement an *unlimited number* of interfaces.
- The super-interfaces of a class are declared as follows:

```
class <name> extends <name>
  implements <name1>, …, <namen> {
  <members>
}
```

- All of the members of an interface must be abstract method or static fields (which are uncommon and prohibited in DrJava language levels).



For Next Class

- Easy Homework due Friday
- Reading: OO Design Notes, Ch 1.9.-1.11.