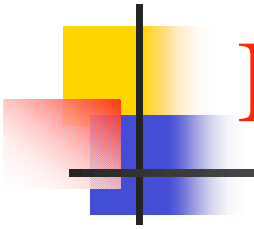
Adapting Our Design Recipe to Java

Corky Cartwright

Department of Computer Science

Rice University

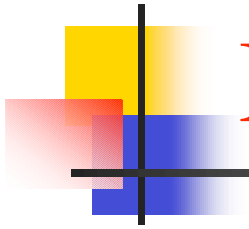


If and Other Statements

- Java is a statement based language rather than an expression language.
- `if` statements are used to express explicit conditional control in most OO languages including Java. Note: `if` statements are used much less frequently in well-written OO code than they are in functional or procedural code.
- An `if` statement has the following syntax:

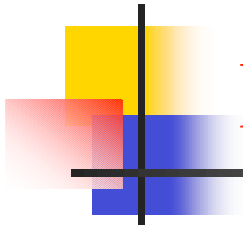
  `if (`*test*`)` *statement*

  `else` *statement*
- What other forms of statements have we used implicitly up to this point?
  - Variable definition: *type var* `=` *expr*`;`
  - Return: `return` *expr;*



Method Definition Revisited

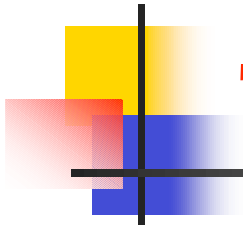
```
class Entry {
  /* fields */
  String name, address, phone;

  /** return true iff name matches keyName.*/
  Entry match(String keyName) {
    if (keyName.equals(name)) return true;
    else return false;
  }
  // Note: version without if was much cleaner
}
```



Reprise: the Design Recipe (Scheme)

How should I go about writing programs?

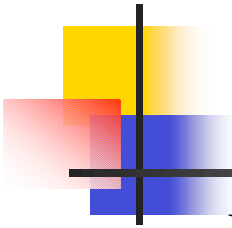
- Analyze problem, which includes:
  - defining any data types (and templates) that are not primitive;
  - determining what top-level (visible) functions must written.
- For each top-level function *f* to be written:
  1. State contract (type signature) and purpose of *f*.
  2. Give input-output examples for *f* written as tests
  3. Select and instantiate a template for the function body. Code the function by filling in the template
  4. Run the tests and confirm that they succeed.
- Writing a function may require help functions. Add these functions to the list of functions to be written. Use local? Perhaps.



The Design Recipe for Java

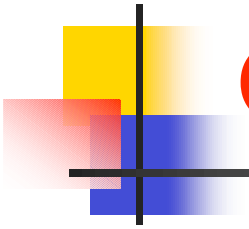
How should I go about writing programs?

- Analyze problem, which includes:
  - defining any classes ***C*** for data types that are not primitive;
  - determining what visible methods should appear in each class.
- For each visible method ***m*** in each class ***C*** :
  1. Write the header (type signature) and contract (purpose) for ***m***.
  2. Create a test class for ***C*** (or the set of tightly coupled classes including ***C*** if it does not already exist) and write a test method for ***m*** that checks it behavior on representative inputs.
  3. Select and instantiate a template for the method body.
  4. Code the method by filling in the template
  5. Run the tests and confirm that they succeed.
- Writing a method ***m*** may require help methods. Add these methods to the class ***C*** containing ***m***. Use `private`? Perhaps.



Java Data Types

- Primitive types: `int long short byte double float char boolean`
- Important primitive operations dicussed in monogaph and lab. Written in conventional infix/prefix notatin following C conventions
- Object types
  - Organized in a strict hierarchy with `Object` at the top.
  - Every class ***C*** except `Object` has an immediate *superclass*, which is the parent of ***C*** in the hierarchy.
  - A descendant in the class hierarchy is called a *subclass*. ***B*** is a subclass of ***A*** iff ***A*** is a superclass of ***B***.
  - Subclassing implies subtyping and vice-versa: if ***B*** is a subclass of ***A***, then ***B*** is a subtype of ***A***. If class ***B*** is a subtype of class ***A***, then ***B*** is a subclass of ***A***
  - An object ***o*** is an instance of only one class but belongs to a hierarchy of types.
  - Each subclass `C` *inherits* (includes) all of the members of its superclass.
  - Declared members of `C` augment the inherited members with *one exception*: if `C` declares a method `m` defined in the superclass, new definition *overrides* old.



OO style

- OO languages are designed to support writing programs in which *dynamic dispatch* is the principal control mechanism. Dynamic dispatch refers to the fact that in a method invocation

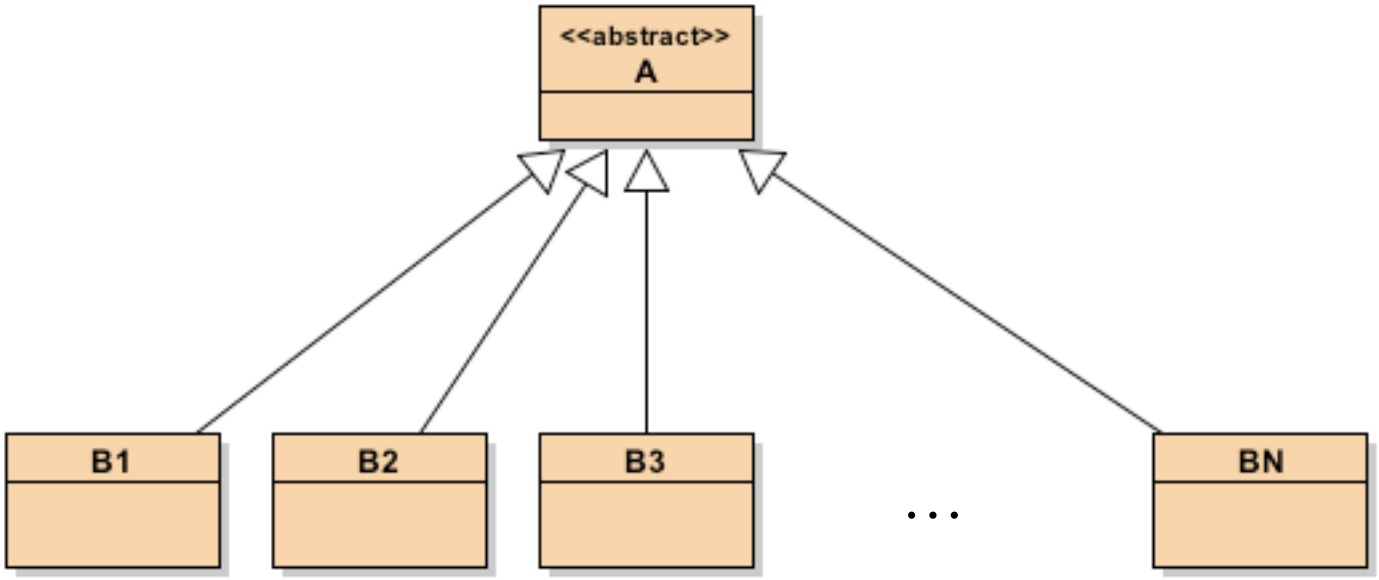
  `o.m()`

  the method code executed depends on the *class* of `m`. Recall that the method `m` is conceptually part of the object `o`. This idea is astonishingly powerful.
- The essence of OO design is representing data and computations in a form that leverages dynamic dispatch.

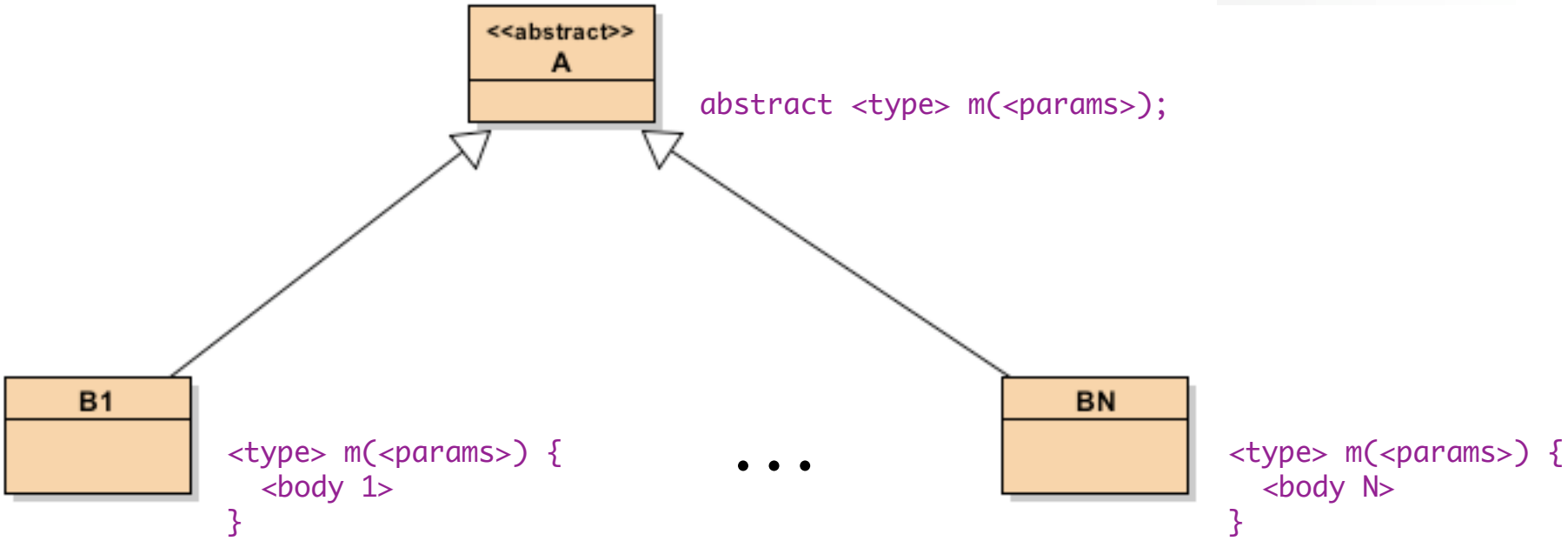
Union Pattern

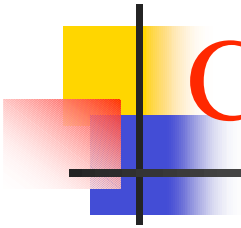
- The *union pattern* is used to represent different forms of *related* data with some common behavior.
- The pattern consists of an `abstract` class ***A*** together with a collection of *variant* subclasses B_1, ..., B_N extending ***A***. An `abstract` class cannot be instantiated using `new`. Note: if ***A*** is concrete then it is not the unition of B_1, ..., B_N because ***A*** has additional members that are instances of ***A***
- The collection of classes ***A***, B_1, ..., B_N is called a union hierarchy and ***A*** is called the *root* class of the hierarchy.
- The common behavior is codified by a set of methods in ***A***, which may be `abstract` Each such method ***m*** has an associated contract that that the implementation in each variant class must obey.

Class Diagram of Union Pattern



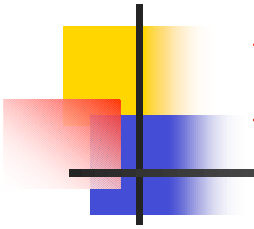
Defining a Method on a Union





City Directory Example

- Assume that we want to design the data for an online city phone book. In contrast to our `DeptDirectory` example, such a directory will contain several different kinds of listings: businesses, residences, and government agencies.
- The entry data for such a directory is represented by using the union pattern to identify the common behavior among the various kinds of listings.



Definition of `CityEntry`

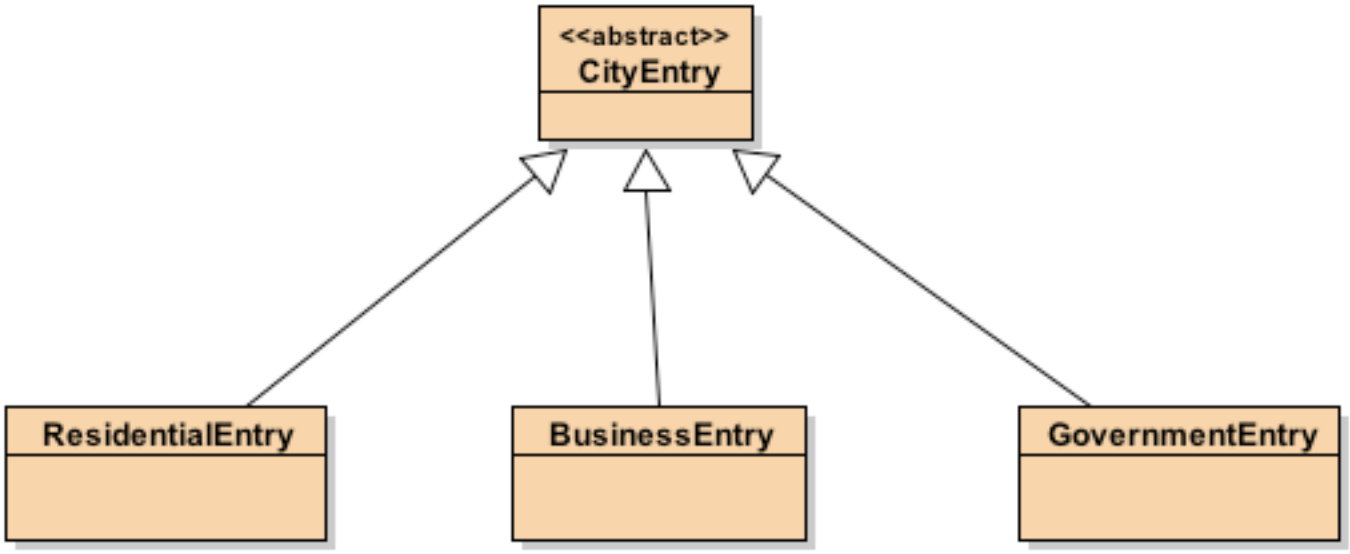
A `CityEntry` is either:

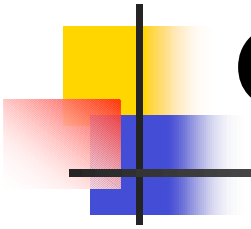
- a `ResidentialEntry(name, address, phone)`
- a `BusinessEntry(name, address, phone, city, state)`
- a `GovernmentEntry(name, address, phone, city, state, government)`

Examples:

```
ResidentialEntry("John Doe","3310 Underwood", "713-664-8809")
BusinessEntry("ToysRUs","2101 Old Spanish Trail",
  "713-664-1234","Houston", "TX")
GovernmentEntry("Federal Drug Administration",
  "800-666-9000", "Washington", "DC", "Federal")
```

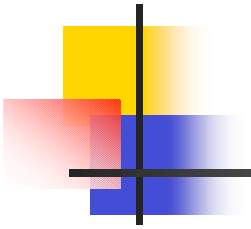
Class Diagram of `CityEntry` Union





Crude Code for `CityEntry`

```
abstract class CityEntry { }

class BusinessEntry extends CityEntry {
  String name, address, phone, city, state;
}

class GovernmentEntry extends CityEntry {
  String name, address, phone, city, state,
   government;
}

class ResidentialEntry extends CityEntry {
  String name, address, phone, city;
}
```



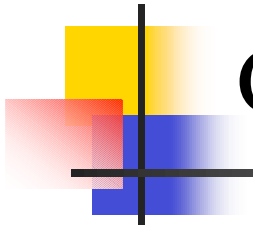
Defining Methods on Unions

- Assume that we want to define a method on a union. The method will typically require a separate implementation for each variant (subclass) of the union. But each implementation will satisfy the same (description of behavior).
- In Java, the method must not only be defined in each variant of the union, it must be declared as `abstract` in the root class of the union hierarchy. Otherwise, Java will not allow the method to be invoked on objects of the union type.

Defined Method for `CityEntry`

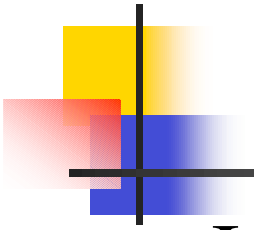
- Let's illustrate the definition of a plausible method for `CityEntry` :

```
abstract class CityEntry {

  /** Returns true if key is a prefix of name. */
  abstract boolean nameStartsWith(String key);

}
```



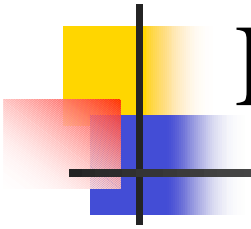
Crude Expanded Code for `CityEntry`

```
abstract class CityEntry {
  /** Returns true if key is a prefix of name. */
  abstract boolean nameStartsWith(String key);
}

class BusinessEntry extends CityEntry {
  String name, address, phone, city, state;
  boolean nameStartsWith(String key) { return name.startsWith(key); }
}
class GovernmentEntry extends CityEntry {
  String name, address, phone, city, state, government;
  boolean nameStartsWith(String key) { return name.startsWith(key); }
}
class ResidentialEntry extends CityEntry {
  String name, address, phone, city;
  boolean nameStartsWith(String key) { return name.startsWith(key); }
}
```



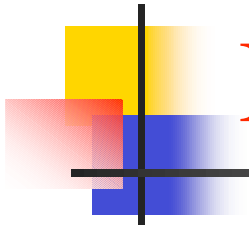
Member Hoisting

- In a union hierarchy, the same code may be repeated in every variant.
- A cardinal rule of software engineering is **never duplicate code**. We can eliminate code duplication in a union hierarchy by hoisting duplicated code (code that is *invariant* within the union) into the abstract class at the route of the hierarchy.



Revised Code for `CityEntry`

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

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

class BusinessEntry extends CityEntry {
  String state;
}
class GovernmentEntry extends CityEntry {
  String state, government;
}
class ResidentialEntry extends CityEntry {
  String city;
}
```



For Next Class

- Exams due Friday
- Optional Homework due Monday
- Reading: OO Design Notes, Ch 1.1 - 1.4.1.