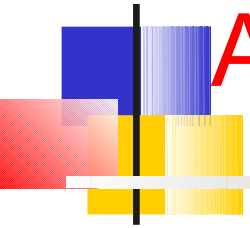
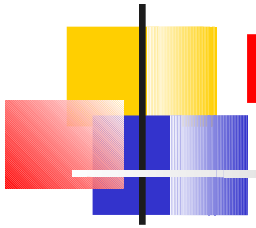


Adapting Our Design Recipe to Java

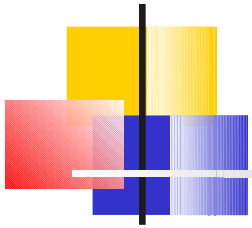
Corky Cartwright
Vivek Sarkar
Department of Computer Science
Rice University



Reprise: the Design Recipe (Scheme)

How should we go about writing programs?

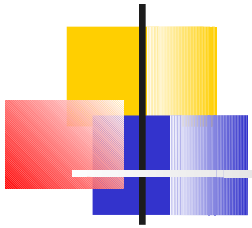
- Analyze problem, which includes:
  - defining any data types (and corresponding structural templates) that are not primitive;
  - determining what top-level (visible) functions must written.
- For each top-level function *f* to be written:
  1. State contract (type signature) and purpose of *f*.
  2. Give input-output examples for *f* written as tests
  3. Select and instantiate a template for the function body and primary argument. Define auxiliary functions for other arguments if needed.
  4. Code the function by filling in the template
  5. Run the tests and confirm that they succeed.



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` :
  - Write the header and contract (HTDP: purpose) for `m`.
  - 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.
  - Select and instantiate a template for the method body and primary argument (`this`). Define auxiliary helper methods as needed, and add them to the class `C` containing `m`.
  - Code the method by filling in the template
  - Run the tests and confirm that they succeed.

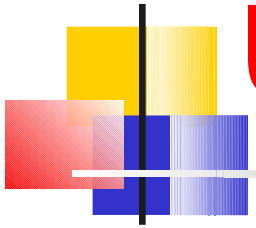


OO style

- 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 `o`. 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.

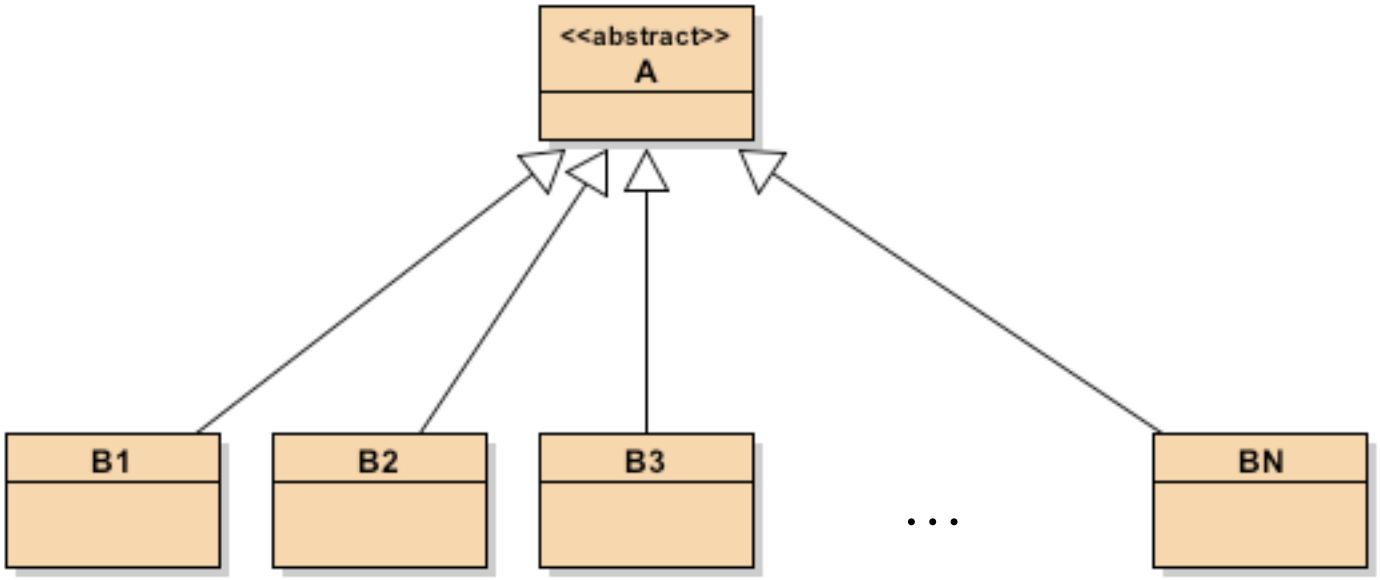


Union Pattern

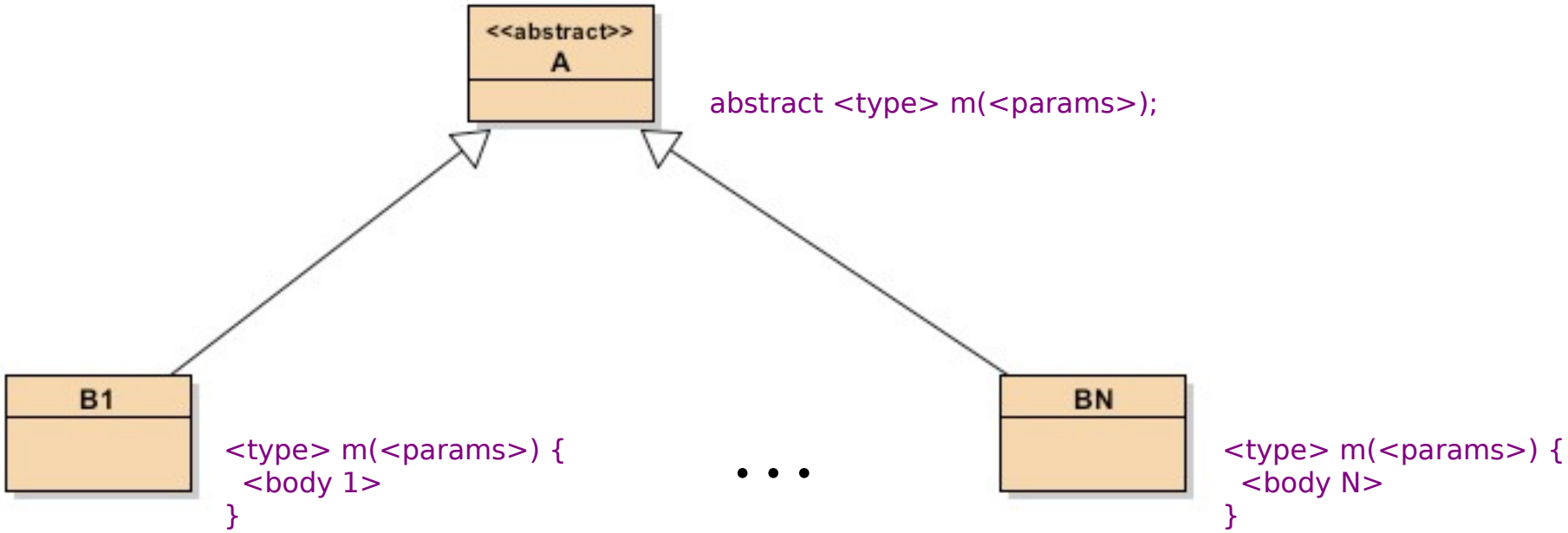
- The *union pattern* is used to represent different forms of *related* data with some common behavior.
- The pattern consists of an **`abstract`** class **`A`** together with a collection of *variant* subclasses $\mathbf{B_1}$, …, $\mathbf{B_N}$ extending **`A`**. An **`abstract`** class cannot be instantiated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

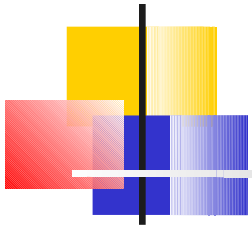
*In Java, the term *contract* corresponds to *purpose* in HTDP.

Class Diagram of Union Pattern



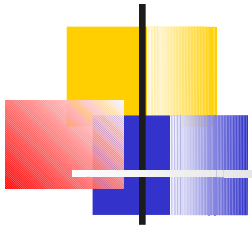
Defining a Method on a Union





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.



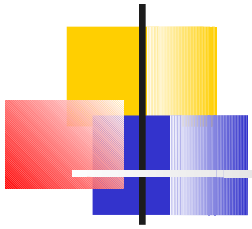
Definition of `CityEntry`

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:

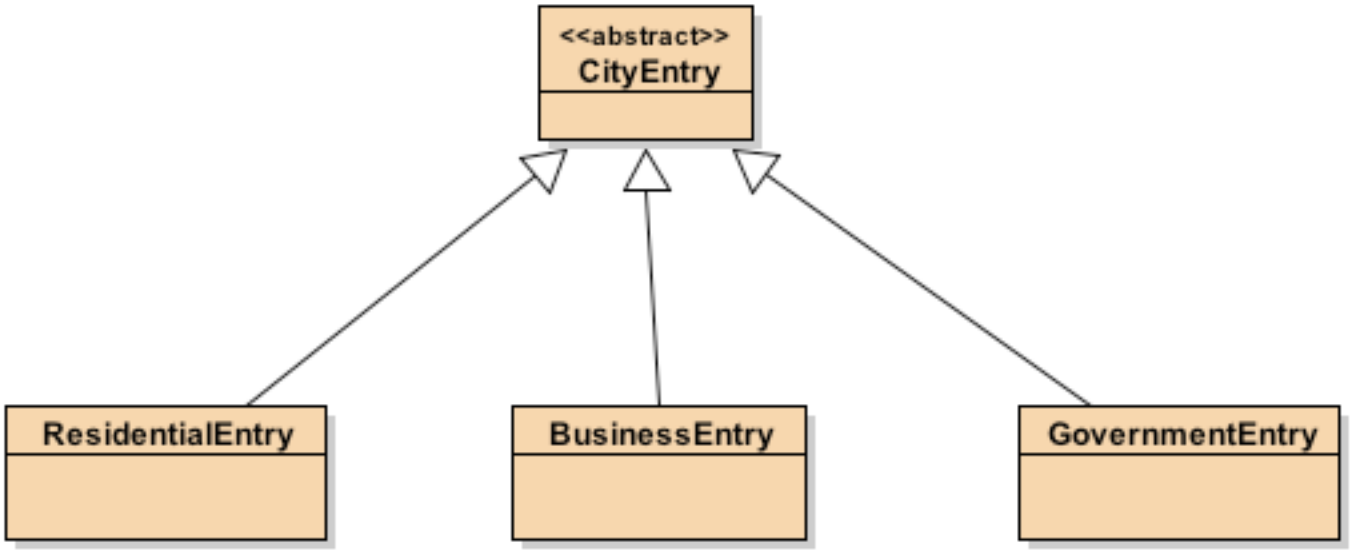
```
ResidentialEntry("John Doe","3310 Underwood", "713-664-8809")

BusinessEntry("ToysRUs","2101 Old Spanish Trail",
   "713-664-1234","Houston", "TX")

GovernmentEntry("Federal Drug Administration",
   "800-666-9000", "Washington", "DC", "Federal")
```

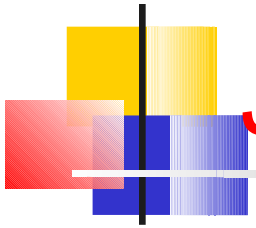


Initial Code for `CityEntry` (v1)

```
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;
}
```

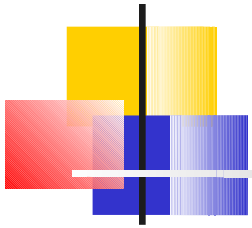
Class Diagram of `CityEntry` Union





Java Class Types

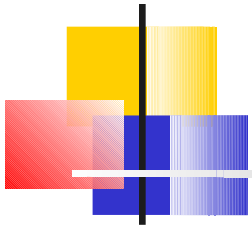
- Organized in a strict hierarchy with `Object` class at the top (root).
- Every class `C` except `Object` has a unique immediate *superclass* which is the parent of `C` in the hierarchy.
- A descendant in the class hierarchy is called a *subclass*. `B` is a subclass of `A` iff `A` is a superclass of `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 all its superclasses.
- Declared members of `C` augment the inherited members with *one exception*: if `C` declares a method `m` defined in the superclass, the new definition *overrides* the old.



Defining Methods on Unions

- Assume that we want to define a method on a union that requires a separate implementation for each variant (subclass) of the union. Each implementation will satisfy the same contract (description of behavior).

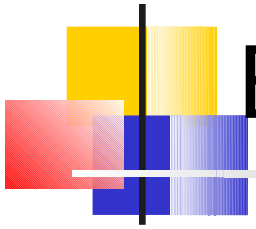
- In Java, the method must not only be defined in each variant of the union, it must be declared as `abstract` in the root class of the union hierarchy. Otherwise, Java will not allow the method to be invoked on objects of the union type.



Defined Method for `CityEntry`

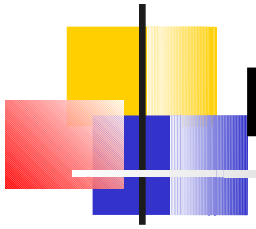
- Let's illustrate the definition of a plausible method for **`CityEntry`**:

```
abstract class CityEntry {

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

}
```



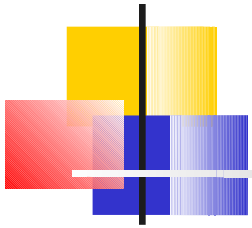
Extended Code for `CityEntry` (v2)

```
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;
  boolean nameStartsWith(String key) { return name.startsWith(key); }
}
```



Example: using the `CityEntry` code

```
CityEntry be = new BusinessEntry("ToysRUs", "2101 Old Spanish Trail",
                                 "713-664-1234","Houston", "TX");

CityEntry ge = new GovernmentEntry("Federal Drug Administration",
                       "800-666-9000", "Washington", "DC", "Federal");

CityEntry re = new ResidentialEntry("John Doe", "3310 Underwood",
                                    "713-664-8809");

boolean b = be.nameStartsWith("Toys"); // true

boolean g = ge.nameStartsWith("Drug"); // false

boolean r = re.nameStartsWith("J");    // true
```

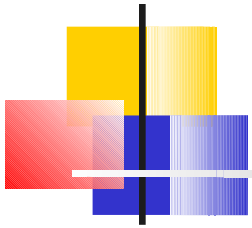


How would we have written `nameStartsWith` in Scheme?

```
;; CityEntry-nameStartsWith: CityEntry -> boolean
;; return true iff city entry C's name starts with key
(define (CityEntry-nameStartsWith C key)
  (cond
    [(BusinessEntry? C)
       (stringStartsWith (BusinessEntry-name C) key)]
    [(GovernmentEntry? C)
       (stringStartsWith (GovernmentEntry-name C) key)]
    [(ResidentialEntry? C)
       (stringStartsWith (ResidentialEntry-name C) key)]
  )
)
```

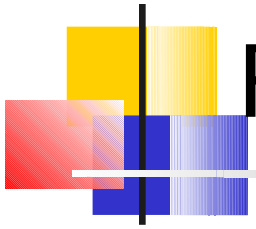
Assumes three distinct structs for `BusinessEntry`, `GovernmentEntry`, `ResidentialEntry`

NOTE: Interleaving of logic for different structs leads to "spaghetti" code



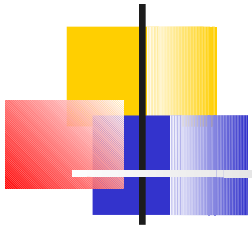
Member Hoisting

- 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.



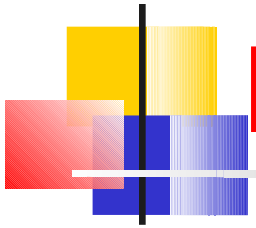
Revised Code for `CityEntry` (v3)

```
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 BusinessEntry extends CityEntry {
  String city, state;
}

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

class ResidentialEntry extends CityEntry { }
```



Usage of revised `CityEntry` code can stay unchanged

```
CityEntry be = new BusinessEntry("ToysRUs", "2101 Old Spanish Trail",
                                 "713-664-1234","Houston", "TX");

CityEntry ge = new GovernmentEntry("Federal Drug Administration",
                       "800-666-9000", "Washington", "DC", "Federal");

CityEntry re = new ResidentialEntry("John Doe", "3310 Underwood",
                                    "713-664-8809");

boolean b = be.nameStartsWith("Toys"); // true

boolean g = ge.nameStartsWith("Drug"); // false

boolean r = re.nameStartsWith("J");    // true
```



Reminders

- Exams due in class Friday
- Optional Homework due Monday
- Reading: OO Design Notes, Ch 1.1 - 1.4.2.
- No classes next week!