Lecture 2: Data Definitions: Unions

Design of classes that represent a disjoint union of sets of data. Extending unions to represent self-referential data.

Related files:
Unions.java AncestorsData.java

Overview

In the last lecture, we saw how to represent primitive forms of data, and simple compound data: the equivalent of atomic data and structs in DrRacket. In this lecture, we continue with more complex forms of data, namely unions of data and recursive data.

2.1 Unions: Traveling on the T in Boston

Suppose we want to represent train stations for the subway and for the commuter lines. Each station has a *name* and the *name of the line* that serves it.

> We'll see in a few days how to represent stations like Downtown Crossing, with multiple train lines available. Much later, we may talk about the challenges of how to represent stations like Ruggles, that are both subway and commuter rail stations.

A subway station also has a *price* it costs to get on the train. (This is a simplification: For the commuter rail the price also depends on the exit station, but for now we'll assume that all commuter rail customers are traveling between their entry station and South Station, so the prices depend only on where they board.) Additionally, a station on the commuter line may be *skipped* by the express trains, so we need to record this information.

Examples:

- *Harvard* station on the *Red* line costs *$1.25* to enter
- *Kenmore* station on the *Green* line costs *$1.25* to enter
- *Riverside* station on the *Green* line costs *$2.50* to enter
- *Back Bay* station on the *Framingham* line is an *express* stop
- *West Newton* stop on the *Framingham* line is *not an express* stop

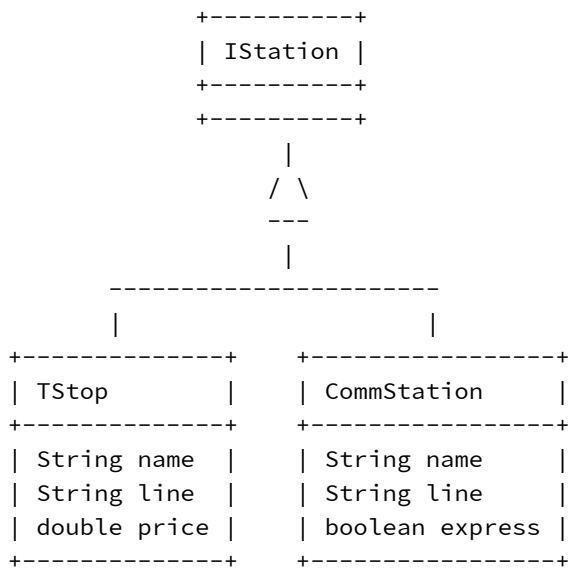
- *Wellesley Hills* on the *Worcester* line is *not an express* stop

> ***Do Now!***
>
> Represent this information in Racket definitions.

One way to represent this information is as follows:

```
;; IStation is one of
;; -- T Stop
;; -- Commuter Station

;; T Stop is (make-tstop String String Number)
(define-struct tstop (name line price))

;; Commuter Station is (make-commstation String String Boolean)
(define-struct commstation (name line express))

(define harvard (make-tstop "Harvard" "red" 1.25))
(define kenmore (make-tstop "Kenmore" "green" 1.25))
(define riverside (make-tstop "Riverside" "green" 2.50))

(define backbay (make-commstation "Back Bay" "Framingham" true))
(define wnewton (make-commstation "West Newton" "Framingham" false))
(define wellhills (make-commstation "Wellesley Hills" "Worcester" false))
```

These data definitions can also be represented by the following class diagram:



This diagram demonstrates two new concepts: first, it gives a common name to the data type that represents both kinds of station by defining an *interface* `IStation`. Second, it shows two classes that *implement* this interface, as indicated by the hollow-triangle arrow on the line connecting the classes to the interface. Colloquially, we say that "`TStop` is a `IStation`, and so is `CommStation`." This is a key distinction: Contrast this with the v-shaped arrowheads in the diagram in Lecture 1, which were used to indicate that `Book` "has-a" `Author` field. Here, we are

indicating that `TStop` "is-a" `IStation`. "Has-a" and "is-a" relationships are important organizing concepts for our data definitions, so our diagrams help us track this information. Interfaces are how we will inform Java about our union data types that (until now) had been described only in comments, and declaring that a class implements an interface is how we inform Java which classes are part of our union.

The data definitions then become:

```
// to represent a train station
interface IStation {
}

// to represent a subway station
class TStop implements IStation {
  String name;
  String line;
  double price;

  TStop(String name, String line, double price) {
    this.name = name;
    this.line = line;
    this.price = price;
  }
}

// to represent a stop on a commuter line
class CommStation implements IStation {
  String name;
  String line;
  boolean express;

  CommStation(String name, String line, boolean express) {
    this.name = name;
    this.line = line;
    this.express = express;
  }
}
```

The syntax for defining interfaces is to use the `interface` keyword, followed by the name of the interface (which by convention will always start with an uppercase "I"), followed by braces. (You may have guessed from the fact that interfaces have a brace-delimited block that perhaps something can be written there, like the body of class definitions. We will see in Lecture 4 how to use this space.)

The syntax for declaring that a class implements an interface is to add the keyword `implements` just after the class name, followed by the name of the interface.

2.2 Defining Examples of Data

As before, we define examples of data in a corresponding class `ExamplesIStation`:

```
class ExamplesIStation{
  ExamplesIStation() {}

  /*
   Harvard station on the Red line costs $1.25 to enter
   Kenmore station on the Green line costs $1.25 to enter
   Riverside station on the Green line costs $2.50 to enter

   Back Bay station on the Framingham line is an express stop
   West Newton stop on the Framingham line is not an express stop
   Wellesely Hills on the Worcester line is not an express stop
  */

  IStation harvard = new TStop("Harvard", "red", 1.25);
  IStation kenmore = new TStop("Kenmore", "green", 1.25);
  IStation riverside = new TStop("Riverside", "green", 2.50);

  IStation backbay = new CommStation("Back Bay", "Framingham", true);
  IStation wnewton = new CommStation("West Newton", "Framingham", false);
  IStation wellhills = new CommStation("Wellesley Hills", "Worcester", false);
}
```

How do we use these examples?

2.3 Enter the Tester

Lab 1 introduces you the *Tester library*, which provides facilities for testing your code, much like you had in Fundies 1. However, unlike Fundies 1, the test forms are not part of the language; in Java, we've had to write a tester library for you, and its functionality will be explained over the next few days.

If you follow the instructions in Lab 1, and then run the program above, the *tester* will automatically create an instance of our `ExamplesIStation` class for us, and display the data as follows:

> This "pseudosyntax" can be read as follows: strings, numbers and booleans are printed as you would write them in Java, but each object is written as "new *ClassName*:*number*(*fields*)". The numbers are irrelevant for now, and can be ignored. The fields are given as "this.*fieldName* = *value*", and each value is itself printed as more pseudosyntax. Finally, the line saying "No test methods found" implies we're not taking full advantage of the Tester library yet; that will change after Lecture 3.

```
---------------------------------
Tests for the class: ExamplesIStation
Tester Prima v.1.5.2.1
---------------------------------
Tests defined in the class: ExamplesIStation:
```

```
---------------------------
ExamplesIStation:
---------------

 new ExamplesIStation:1(
  this.harvard =
   new TStop:2(
    this.name =  "Harvard"
    this.line =  "red"
    this.price = 1.25)
  this.kenmore =
   new TStop:3(
    this.name =  "Kenmore"
    this.line =  "green"
    this.price = 1.25)
  this.riverside =
   new TStop:4(
    this.name =  "Riverside"
    this.line =  "green"
    this.price = 2.5)
  this.backbay =
   new CommStation:5(
    this.name =  "Back Bay"
    this.line =  "Framingham"
    this.express = true)
  this.wnewton =
   new CommStation:6(
    this.name =  "West Newton"
    this.line =  "Framingham"
    this.express = false)
  this.wellhills =
   new CommStation:7(
    this.name =  "Wellesley Hills"
    this.line =  "Worcester"
    this.express = false))
---------------
No test methods found.
```

> **Exercise**
>
> What would need to be done to represents all stops on all bus lines, including the express bus lines?

2.4 Self-referential unions: Ancestor trees

Suppose we want to represent an ancestry tree for a person, naming the ancestors as far as we can remember, and using “unknown” for those nobody can remember or trace.

Here is the data definition in DrRacket:

```
;; An Ancestor tree (AT) is one of
;; -- 'unknown
;; -- Person

;; A Person is (make-person String AT AT)
(define-struct person (name mom dad))
```

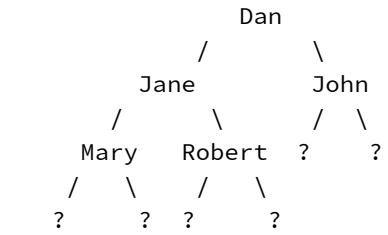
and here are a few examples:

```
(define mary (make-person "Mary" 'unknown 'unknown))
(define robert (make-person "Robert" 'unknown 'unknown))
(define john (make-person "John" 'unknown 'unknown))
(define jane (make-person "Jane" mary robert))
(define dan (make-person "Dan" jane john))
```

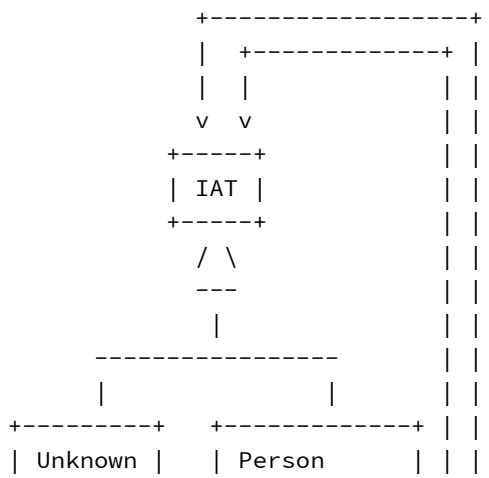
Two important skills you need to practice early are:

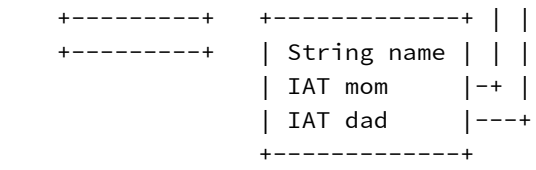
- design a representation of the given information as data
- interpret the given data as the information it represents

So, here, to understand what the data shown above represents, we may want to draw the ancestor tree, so we can tell easily how the different people are related. The given data represents the following ancestor tree:



(For brevity, we’re using the `?` for the “unknown” names.) The class diagram that represents this data definition looks like this:





We replaced `AT` with `IAT` since in Java the union type will be represented as an `interface`, and our naming convention tells us to start the names of interface types with the uppercase letter `I`.

Note the two kinds of arrows here: `Person` and `Unknown` both implement `IAT`, while `Person` also has two `IAT` fields.

The code below shows how this class diagram and our examples can be translated into a represetation as Java classes and interfaces (slightly misnamed as a *Java class hierarchy*):

```
// to represent an ancestor tree
interface IAT{ }

// to represent an unknown member of an ancestor tree
class Unknown implements IAT{
  Unknown() {}
}

// to represent a person with the person's ancestor tree
class Person implements IAT{
  String name;
  IAT mom;
  IAT dad;

  Person(String name, IAT mom, IAT dad) {
    this.name = name;
    this.mom = mom;
    this.dad = dad;
  }
}
```

Notice that we have defined a class `Unknown`, even though it does not yet contain any additional information (has no fields). Its purpose for now is simply to be *different* from `Person`, so we can tell them apart.

> *Do Now!*
>
> There is another subtle but crucial reason we need `Unknown`. What might that be? Hint: how might you build examples of ancestor trees without them?

As you may remember from last semester, we can define interesting behavior for unions, by defining functions that distinguish between the cases of the union and return different results in each case. In Lecture 4, we will see how this works in Java, which will involve enhancing the definitions of `Unknown`, `Person`, and `IAT`.

Examples of Ancestor Trees

As before, we define examples of the data in the class named `ExamplesAncestors`:

```
// examples and tests for the class hierarchy that represents
// ancestor trees
class ExamplesAncestors{
  ExamplesAncestors() {}

  IAT unknown = new Unknown();
  IAT mary = new Person("Mary", this.unknown, this.unknown);
  IAT robert = new Person("Robert", this.unknown, this.unknown);
  IAT john = new Person("John", this.unknown, this.unknown);

  IAT jane = new Person("Jane", this.mary, this.robert);

  IAT dan = new Person("Dan", this.jane, this.john);
}
```

> *Do Now!*
>
> Why do we declare the types of all of these identifiers to be `IAT`? Why not write `Person mary = new Person(...);` or `Unknown unknown = new Unknown();`?

At the time the program is checked for corectness by the compiler, the compiler knows that `mary` is of the type `IAT` (also known as its *compile-time type*; when the program runs, it will also know that after the data definition has been made, the *run-time type* of `mary` is `Person`.

We can see this when the Tester library displays our data:

```
ExamplesAncestors:
---------------

 new ExamplesAncestors:1(
  this.unknown =
   new Unknown:2()
  this.mary =
   new Person:3(
    this.name =  "Mary"
    this.mom = Unknown:2
    this.dad = Unknown:2)
  this.robert =
   new Person:4(
    this.name =  "Robert"
    this.mom = Unknown:2
    this.dad = Unknown:2)
  this.john =
   new Person:5(
```

```
    this.name =  "John"
    this.mom = Unknown:2
    this.dad = Unknown:2)
  this.jane =
   new Person:6(
    this.name =  "Jane"
    this.mom = Person:3
    this.dad = Person:4)
  this.dan =
   new Person:7(
    this.name =  "Dan"
    this.mom = Person:6
    this.dad = Person:5))
```

When designing a program, we must make sure that the *compile-time types* of all data items (identifiers and arguments) correspond to what the compiler can expect. (More about this later.)