Lecture 1: Data Definitions in Java

Design of simple classes and classes that contain objects in another class.

Related files:
DataDefinitions.java

Overview

- Data vs. Information, and the need for structured data
- Saying what you mean: the benefits of static types
- Simple classes: `Book`, `Author`
- Containments: `Book` and `Author`
- Examples of data

Note: throughout these lecture notes, you'll see the following:

> *Do Now!*
>
> A **Do Now!** exercise is strongly recommended for you to do now, without reading ahead in the lecture notes. You should think through the question being asked, and then see if your understanding matches the concepts the lecture notes suggest.

> **Exercise**
>
> An **Exercise** is strongly recommended for you to do, now or when finished reading these notes. These exercises may suggest further applications of the concepts in the lecture notes.

1.1 Representing information by structured data

Information is everywhere: "it is 75 degrees Farenheit outside", "Providence, RI is 65 miles away from Boston, MA", "the book *How to Design Programs* was written by Matthias Felleisen", etc. As humans, we can read those sentences and make sense of them: we understand the *information* they are trying to convey, and can use that information in other settings. But how can we cause a computer to do the same?

Computers do not work with information: they work with *data*, which is a *representation* of the information relevant for the computation. Data is different from information in much the same way that the four characters "2510" are not the same thing as the number two-thousand-five-hundred-ten. The first step in designing a program to compute anything is to figure out an appropriate data representation of the relevant information.

1.2 Types of data

Data in DrRacket

So how can we represent information? In Fundies I, we learned about several kinds of data:

- Strings
- Numbers
- Booleans
- Symbols
- Images
- Structures containing other pieces of data
- Unions of other forms of data

When we defined data representations in DrRacket, we included a purpose statement to explain what information a given datum was supposed to represent:

```
;; A number representing the temperature in degrees Farenheit
(define outside-temp 75)
```

It was our responsibility to use `outside-temp` correctly: if we mistakenly tried to combine it with a string, say, DrRacket would complain that `75` is not a string. Such a bug is all too common when writing programs. But sometimes this kind of bug can be more subtle: suppose we have the following program:

```
;; A number representing the temperature in degrees Farenheit
(define outside-temp 75)

;; Some examples...
(define hot-day (+ outside-temp 50))
(define bad-day (+ "75 degrees" 50))
```

> ***Do Now!***
>
> If we type that whole program into DrRacket, what will happen?

Nothing! But if we run the program, the definition of `bad-day` will complain that

```
+: expects a number as 1st argument, given "75 degrees".
```

> *Do Now!*
>
> Why couldn't DrRacket warn us about our mistake as soon as we wrote it?

Unfortunately, DrRacket cannot use the purpose statement for the `+` operator to enforce that it only accepts numbers, because the purpose statement for `+` (or for anything, for that matter) is just a comment. Instead, we must wait until runtime to detect the error.

Data in Java

Look carefully at the purpose statement for `outside-temp` above: it says that `outside-temp` is a `Number` that represents a temperature. The first part of that statement describes the *type* of the data, while the second part describes what that number *represents*. In Java, when we define data, we will always include the type of the data *in the definition*, rather than just in comments. In this way, we can inform Java what type of data we mean, and Java can then check for these *type errors* even before we run our program.

Let's see how to define some simple forms of data, corresponding to the kinds of data we saw in Fundies I.

Booleans

Let's start with the simplest data: Boolean values. We can define an identifier to be Boolean like so:

```
boolean isDaytime = true;
boolean inNewYork = false;
```

In Java, Boolean values are written `true` and `false`, and the name of the type is `boolean`.

Strings

The next simplest form of data are strings. Defining strings is similar:

```
String courseName = "Fundies II";
```

(We'll explain later why `boolean` is lowercase but `String` is capitalized.)

Numbers

There are many sorts of numbers: integers, decimals, complex numbers, and more. In DrRacket, we called them all "Number", but in Java we must be more precise. Integers are written as follows:

```
int ten = 10;
```

Decimal numbers are written as follows:

```
double pointSix = 0.6;
```

We'll have a lot more to say about the difference between `int` and `double` later, especially as it regards writing test cases.

Symbols and Images

Java does not have DrRacket's notion of symbols. And in Java, images are not a built-in form of data; we will come back to images next week.

So much for simple forms of data. Our goal now is to learn how to represent more complex information in Java.

Classes of Data

The *Design Recipe for Data Definitions* says that if the information consists of several components, it should be represented by a structure. For example, here is the data definition for a book in a bookstore in DrRacket:

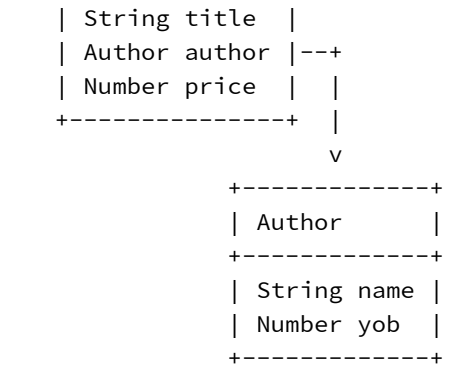
```
;; to represent a book in a bookstore
;; A Book is (make-book String Author Number)
(define-struct book (title author price))

;; to represent an author of a book (with the year of birth)
;; An Author is (make-author String Number)
(define-struct author (name yob))
```

The *Design Recipe for Data Definitions* tells us to follow up with concrete examples of data:

```
;; Examples:
(define pat (make-author "Pat Conroy" 1948))
(define beaches (make-book "Beaches" pat 20))
```

In Java, we don't define "struct"s to describe compound data; we define *classes*. (Throughout this course, we'll see the ways classes differ from structs in how they let us organize our code, and the abstraction mechanisms they provide.) A more visual way to describe these structure definitions is with a diagram:

```
+---------------+
| Book          |
+---------------+
```



We will often draw these *class diagrams* instead of writing out the corresponding Java code. Here, the Java definitions of these classes is:

```
// to represent a book in a bookstore
class Book {
  String title;
  Author author;
  int price;

  // the constructor
  Book(String title, Author author, int price) {
    this.title = title;
    this.author = author;
    this.price = price;
  }
}

// to represent a author of a book in a bookstore
class Author {
  String name;
  int yob;

  // the constructor
  Author(String name, int yob) {
    this.name = name;
    this.yob = yob;
  }
}
```

Notice the differences from the DrRacket code: instead of specifying the signature for the constructors `make-book` and `make-author` as comments, we declare the *fields* of our classes with their types. Additionally, while the `define-struct` form in DrRacket defined constructor functions for each struct for us, in Java we define *constructors* for the classes explicitly, though for the time being, they will all look the same: the name of the class, followed by a parenthesized list of argument types and names that (for now) mimic exactly the field definitions, then a block (enclosed in curly braces) of *initialization statements* `this.fieldname = argumentname;` that initialize the fields of our object to the provided values. For clarity and convenience, we choose to give the arguments the same names as the fields of the object, so it is obvious which argument corresponds to which field. The `this`

keyword used in these initialization statements indicates which object is being initialized: *this* one, and not some other one.

Naming convention: class names in Java always are written in `TitleCase`, and field names are always written in `camelCase`. Primitive type names, like `int` and `boolean`, are lowercase.

> ***Do Now!***
>
> Why do you think `String` is capitalized?

1.3 Examples of Data

While we have now defined the *classes* `Book` and `Author`, we haven't yet defined any actual books or authors. As a matter of terminology, we say we create *instances* of these classes, and these instances are known as *objects*.

Unlike DrRacket, in Java we are not allowed to just define identifiers whenever and wherever we like. All identifiers must be defined within classes: for now, that means we can only define them as fields of some class.

> ***Do Now!***
>
> Does it make sense to construct examples of e.g. the `Book` class as fields of the `Book` class itself?

This means we need to define a new class, whose purpose is to demonstrate examples of our data. Accordingly, we'll name the class `Examples`*Something* – either `Examples`*ProblemName* if we are working on a specific problem, or `Examples`*ClassName* if we are building examples specifically for one of our data classes. Here, for instance, we're building examples of `Book`s:

```
// examples and tests for the classes that represent
// books and authors
class ExamplesBooks{
  ExamplesBooks() {}

  Author pat = new Author("Pat Conroy", 1948);
  Book beaches = new Book("Beaches", this.pat, 20);
}
```

This introduces two new pieces of syntax: first, in addition to declaring fields we are also initializing them to their values. (Compare with the field definitions in `Book` and `Author`, which do not use these initializers.) Second, we are *constructing* new *instances* of the `Book` and `Author` classes. The general syntax `new ClassName(arg1, arg2, ..., argN)` is the Java equivalent of `(make-class-name arg1 arg2 ... argN)` in DrRacket.

Notice the use of `this` in the code above: it's used to indicate which `ExamplesBooks` object's `pat` field is being used to initialize `beaches` – *this* one. This is called *field access*, and in general its syntax is `someObject.aField`, meaning "obtain the value of the `someObject`, then obtain the value of its field named `aField`." (It is legal Java syntax to repeat this process, and write `someObject.aField.anotherField`, but it is bad design – don't be tempted to do it!)

> ***Do Now!***
>
> In terms of the design recipe from last semester, which part of the recipe would we be violating?

1.4 Anatomy of the Class Definition

A class definition consists of the *keyword* `class` followed by the name we choose for this class of data.

Next are the definitions of *fields* of the class, similar to the fields or components of *DrRacket* `structs`. In Java, the field definitions provide both the *type* of data that the field represents, and the *name* of the field, so we can refer to it. Contrast this with *DrRacket*, where the information about the types of data the fields represent was written only in comments. In some classes, such as `ExamplesBooks`, we also supply initial values for the fields. Recognize these by an `=` sign after the field name, followed by an expression that provides the desired value.

After the field definitions comes the *constructor*, which accepts a list of arguments and initializes the fields of the being-created instance to the supplied values.

Creating new instances of classes requires *invoking* the desired constructor. The syntax to do so is the keyword `new` followed by the name of the class for which we want to construct an instance, followed by a parenthesized list of values to give to the fields.

Every field definition and initializer statement in Java must end in a semicolon.

> More or less. We'll see later some of the subtleties here.

Note: order matters! Java executes programs in order, starting at the top of the file and working its way down. This means that when we construct objects, we must be careful of the order in which we initialize fields. Suppose we wrote our examples this way, defining `beaches` before defining `pat`:

```
class ExamplesBooks{
  ExamplesBooks() {}

  // BAD IDEA!
  Book beaches = new Book("Beaches", this.pat, 20);
  Author pat = new Author("Pat Conroy", 1948);
```

```
    // CRASH
    String beachesAuthorName = this.beaches.author.name;
  }
```

> *Do Now!*
>
> When we create an `ExamplesBooks` instance, what will the value of the `author` field of its `beaches` field be?

There are several problems with this example: First, it is poor design to access a field of a field, and all the more so a field of a field of a field! Second, it is silly to initialize the field `beachesAuthorName` from some smaller part of `beaches`; it would be simpler to initialize `beachesAuthorName` *first*, then `pat` based on it, then `beaches` based on that.

Since `this.pat` has not yet been defined and initialized, its value will be `null`. When the program attempts to initialize `beachesAuthorName` by accessing one small part nested in `this.beaches`, the program will end with a `NullPointerException`. Whenever you see this exception, stop and reconsider your program design: you almost certainly have not followed the design recipe, and need to define your data more carefully.

> **Exercise**
>
> Enhance the definitions of `Book` and `Author` above to include `Publisher` information. A `Publisher` should have fields representing their name (that is a `String`), their country of operation (that is also a `String`), and the year they opened for business (that is an `int`). Should `Book`s or `Author`s have `Publisher`s? Enhance the class diagram above to include your new inforamation. Define the new class. And enhance the `ExamplesBooks` class to include examples of the new data.