Lecture 3: Methods for simple classes

Design methods for simple classes of data.
Design methods for classes that contain objects of another class.
Wish lists.

Related files:
Book.java BookAuthor.java Book2.java

Overview

In the last two lectures, we've built classes to represent compound data: structures, nested structures, unions, and recursive unions of structures. However our data has been utterly inert, and we have not defined any computations that work with our data. In this lecture, we'll define *methods*, which are the object-oriented analogue of the functions we've seen before (though they are subtly and importantly different!), work with arithmetic expressions, and build up to methods that consume and produce additional objects.

3.1 Designing methods for simple classes

Related files:
Book.java

Suppose we're modeling books in a bookstore. Recall our definition of the `Book` class from Lecture 1. (We'll simplify it for now by representing the `author` field merely as a `String`; we'll reinstate the `Author` class below.) Let's remind ourselves how we might manipulate books in Racket, and then we'll see how to translate these concepts to Java.

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

;; Examples:
(define htdp (make-book "HtDP" "FFFK" 60))
(define beaches (make-book "Beaches" "PC" 20))
```

Occasionally, stores have sales: on a sale day, all books are discounted by some fixed percentage. We'll therefore want to define a function that will produce the book's sale price. Following the *Design Recipe* we produce the following function and tests:

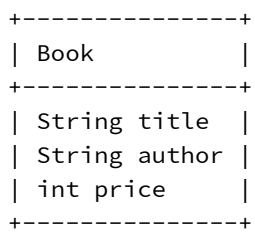
```
;; compute the sale price of a book for the given discount
;; sale-price: Book Num -> Num
(define (sale-price abook discount)
  (- (book-price abook) (/ (* (book-price abook) discount) 100)))

(check-expect (sale-price htdp 30) 42)
(check-expect (sale-price beaches 20) 16)
```

Note that this function is useless for other types of data - per our signature, this function requires that we provide a book.

In Java all computations relevant for one type of data are defined as *methods* in the class that represents the relevant data. (We say that the *methods* represent the behavior of the objects that are the members of this class.)

The class diagram and the Java class definitions for the class of books is straightforward:



> *Do Now!*
>
> Without looking at Lecture 1, translate this diagram into a Java class definition. Translate the examples of books into Java as well.

According to the design recipe, we need a purpose statement for our method, a signature for it, examples of using it, a template for it, then finally its implementation and tests. However, to write examples or tests for this method...we first need to know how to define and invoke methods! So in this one instance, we'll skip writing examples first and come back to them afterward. Once you're familiar with the syntax for defining and invoking methods, you should follow the recipe in order.

When reading the code below, notice the organization:

- The purpose statement for the class precedes the class definition, just as it would precede a struct definition in Racket.
- The class definition starts with the field declarations, followed by the constructor.
- After the constructor comes the *template* (described in more detail below).

> *Do Now!*
>
> Why do you think the template goes *here*, and not anywhere else in the class definition?

- After the template comes each *method definition* for the class.

Here is the Java code:

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

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

  /* TEMPLATE will go here */

  /* METHOD DEFINITIONS will go here. */
}
```

3.1.1 Signature and purpose

Every method definition consists of the following parts:

- A purpose statement, much as we wrote in Racket, except we will be very careful to use the pronoun **this**
- The type of the value returned from the function, known as the *return type*
- The method name, where the standard naming convention starts with a lowercase letter and uses "camelCase" to distinguish words within the name
- A parenthesized argument list, consisting of the type and name of each argument, separated by commas
- The method body, surrounded by braces; this is the code to execute when the method is invoked

How should we define the method `salePrice`? What should its signature be? We know it needs a `Book` and an `int` sale rate, so should it have two arguments? No! Unlike in Racket, where we have to explicitly pass a `Book` in as an argument to the `sale-price` *function*, in Java every *method* always has access to `this`, the object on which we want to *invoke* the method (in this case, `salePrice`). This object acts as an *implicit* argument for the method, and is sometimes called the *receiver object*. Within the body of the method we can access the fields of the receiver object as `this.title` or `this.price`, etc. (These field accesses are *selectors*, similar to `(book-title abook)` or `(book-price abook)` in Racket.) Accordingly, our signature will look like this:

```
// In Book, at the comment "METHOD DEFINITIONS will go here"
// Compute the sale price of this Book given using
// the given discount rate (as a percentage)
int salePrice(int discount) {
  ...
}
```

3.1.2 Template

What should the template look like, and where should it be defined? Recall that in Racket, we defined a template *once and for all* for each data type, rather than once per *function* that we define. Accordingly, we will define a template just once in each class. For now, the template is very simple, and contains only the fields of the current class. Later we will make this more elaborate:

```
// In Book, at the comment "TEMPLATE will go here"
/* TEMPLATE:
   Fields:
   ... this.title ...         -- String
   ... this.author ...        -- String
   ... this.price ...         -- int
*/
```

(A minor note on syntax: `/* ... */` denotes a *block comment* that can span multiple lines, as opposed to `// ...` which is a single-line comment.)

Just as the template for structs in Racket allowed us to tear the struct apart and access its fields, so too the template here allows us access to the fields of *this* object.

3.1.3 Method Body

Now that the preparatory work is done, defining the method is fairly straightforward. In Java, note that arithmetic expressions are *infix*, rather than prefix function calls as in Racket.

```
// In Book
// Compute the sale price of this Book given using
// the given discount rate (as a percentage)
int salePrice(int discount) {
  return this.price - (this.price * discount) / 100;
}
```

> We will revisit the distinction between expressions and statements more carefully in a few lectures.

This method body introduces a new keyword, `return`. Unlike Racket, where functions simply evaluated to “the expression in their body”, in Java we must explicitly state which value we want to return. Accordingly, these are known as *return statements*.

3.1.4 Tests

In order to test our method, we must learn how to *invoke* a method. We show the completed code for these tests below, which introduces several new features, and then explain the syntax in detail.

> You might notice that the constructor for the `ExamplesBooks` class is seemingly useless, as it contains no code and initializes no fields. In this case, you are permitted to omit the constructor entirely.

```
// examples and tests for books
class ExamplesBooks {
  ExamplesBooks() {}

  // examples of books
  Book htdp = new Book("HtDP", "Felleisen et al.", 60);
  Book ror = new Book("Realm of Racket", "Bice et al.", 20);

  // test the method salePrice for the class Book
  boolean testSalePrice(Tester t) {
    return t.checkExpect(this.htdp.salePrice(30), 42)
        && t.checkExpect(this.ror.salePrice(20), 16);
  }
}
```

The syntax for invoking a method is to first write the receiver object, followed by a period, then the name of the method, then a parenthesized list of arguments to the method: for example, `this.htdp.salePrice(30)`. Here, the receiver object is `this.htdp`, the method name is `salePrice`, and the argument is `30`.

> ***Do Now!***
>
> There are three more method invocations in this code: what are they, what are the receiver objects, and what are the parameters?

Testing our `salePrice` method actually required writing another method! Specifically, we needed to add a test method to our examples class. After all, *running* a test means we have to execute some computation, and the only way to execute computations in Java is to define methods. Test methods are slightly special: the tester library looks for methods named `test...`, with a single `Tester` parameter, in classes named `Examples...`, and invokes those methods for us – this is how our tests will get run. The `Tester` object that is given to these test methods is what implements the check-expect functionality you have seen in Racket. Naturally enough, `checkExpect` is a *method* on this object: we are asking the tester library to run some code for us, and give us a report about the passing and failing tests.

The `checkExpect` method returns a `boolean`, indicating whether the particular test passed (`true`) or failed (`false`). Notice that we want to run *two* tests here. If we simply wrote the

following:

```
// In ExamplesBooks
boolean testSalePriceBROKEN(Tester t) {
  // BAD
  return t.checkExpect(this.htdp.salePrice(30), 42);
  return t.checkExpect(this.ror.salePrice(20), 16);
}
```

Java would execute the first test, and get back a `true` result because our test succeeds. It would then immediately `return` that value from the `testSalePrice` method – and never even execute the second test. If we want to execute multiple tests, we have to combine their results, and in particular we only want to succeed if all tests succeed. Accordingly, the `&&` operator is the logical *and* operator. Just like in Racket, it is short-circuiting: if the first clause evaluates to `false` (i.e. our first test fails), the second clause will not be evaluated (i.e. our second test will not be run).

Phew! Quite a lot of new content in only four lines of code!

3.2 Aside: Evaluation of arithmetic expressions

The body of the `salePrice` method contained the following formula:

```
this.price - (this.price * discount) / 100
```

In Racket, every function expression was wrapped in parentheses and evaluation proceeded predictably left-to-right and inside-out. How does the Java code above evaluate?

Java tries to mimic the conventions used in mathematical formulas by assigning *precedences* to operators: it defines an *order of operations*. Specifically, multiplication and division take precedence over addition or subtraction, and equal-precedence operators evaluate from left to right. So the expression above is evaluated as if the parentheses were written:

```
(this.price - ((this.price * discount) / 100))
```

In fact, because of that left-to-right ordering, we could actually drop *all* the parentheses from this particular expression, and it would still evaluate the same way:

```
this.price - this.price * discount / 100
```

Savvy programmers add extra parentheses whenever the order of operations is not clear, or to emphasise the meaning of the formula. (Unlike Racket, adding parentheses does not mean "invoke this value as a function", since Java does not use parenthetical syntax. So adding parentheses for clarity is semantically just fine.)

Note, however, that some mathematically "valid" changes to this expression will produce *different* results than you might expect! For example, because multiplication is associative, we ought to be able to write any of the following and get the same answer:

```
this.price - (this.price * discount) / 100
```

```
this.price - this.price * (discount / 100)
this.price * (1 - discount / 100)
```

However, these will produce two different answers!

> *Do Now!*
>
> Try it and see. Why do you think this is happening?

Let's pick an example `price` of `50`, and a `discount` of `20`.

- In the first line, we first evaluate `50 * 20`, or `1000`, and divide by `100` to get `10`. We subtract `10` from `50` to get the correct answe of `40`.
- In the second line, we first evaluate `20 / 100`, which is *zero* — because we're working with integers. We then multiply `50` by zero, and subtract the result from `50`, leaving `50` — not much of a discount.
- The third line similarly evaluates the division to be zero, leading to the same answer of `50`.

We might argue that the problem is due to using integers and division together — why not just use `double` everywhere and avoid this problem? Unfortunately, `double`s don't obey the laws of algebra completely, either, and cannot be relied upon to check simple relations like equality: for example, `0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1` should equal `1.0`, but it will likely give you `0.99999999999999999` instead. This is still a rounding error; it's just a smaller, subtler one.

> There are *many* articles about floating-point analysis. A relatively gentle introduction can be found here.

The moral here, naturally, is to test your arithmetic thoroughly: be sure you understand the consequences of using `int`s instead of `double`s, and check the order of operations carefully.

3.3 Methods for classes with containment: Designing method templates

Related files:
BookAuthor.java

The example above introduced all the machinery and syntax we'll need to define methods, but we simplified our data definitions for brevity and clarity. Let's continue with the original example that included a `Book` and its `Author`:

```
// 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;
  }
}

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

  // examples of authors
  Author pat = new Author("Pat Conroy", 1948);
  Author dan = new Author("Dan Brown", 1962);

  // examples of books
  Book beaches = new Book("Beaches", this.pat, 20);
  Book prince = new Book("Prince of Tides", this.pat, 15);
  Book code = new Book("Da Vinci Code", this.dan, 20);
}
```

Suppose we would like to determine if the authors of two books are the same.

3.3.1 Signature and purpose

In Racket we might define the signature and purpose of a function to do this as follows:

```
;; are the two given books by the same author?
;; same-author? : Book Book -> Boolean
(define (same-author? book1 book2)...)
```

In Java, the first book becomes the implicit argument (`this`), the instance which invokes the method, and the second book will be the sole explicit argument for the method. Here is the

purpose and the header:

```
// In Book
// is this book written by the same author as the given book?
boolean sameAuthor(Book that) {
  ...
}
```

> *Do Now!*
>
> Complete the design of this method following the design recipe. What new things should be added to the template?

3.3.2 Examples

```
// In ExamplesBooksAuthors
// test the method sameAuthor in the class Book
boolean testSameBookAuthor(Tester t) {
  return t.checkExpect(this.beaches.sameAuthor(this.prince), true)
      && t.checkexpect(this.beaches.sameAuthor(this.code), false);
}
```

3.3.3 Template

Now we look at the template. The common template for all methods in the class `Book` now looks like this:

```
// In Book
/* TEMPLATE:
   Fields:
   ... this.title ...        -- String
   ... this.author ...       -- Author
   ... this.price ...        -- int

   Methods:
   ... this.salePrice(int) ...   -- int
   ... this.sameAuthor(Book) ... -- boolean

   Methods for fields:
   ... this.author.mmm(??) ...    -- ??
*/
```

Our template now consists of *three* sections. The first section, fields, is the same as before. Second, we add all methods defined for the `Book` class. That includes `this.salePrice(int)`, defined earlier, as well as `this.sameAuthor(Book)` we are defining now. So, for example, our method `this.sameAuthor` could invoke `this.salePrice` (though in this case, knowing a sale price is useless for checking authorship).

The third section is interesting: if any of the fields of this class is an instance of another class, we add to our template all methods defined for that other class, as they can be invoked on that field. Here we would add any methods defined for the `Author` class, since such methods can be invoked on `this.author`. We don't have any such methods...yet.

Using the templates we've constructed for each of our classes, we can also construct a template customized for our current method, by adding any information we can extract from the parameters of the method. In this case, such information includes anything that `that` book can provide: we might access its fields, and we might invoke methods on it.

So, our template for this method essentially expands to:

```
// In Book
boolean sameAuthor(Book that) {
  /* TEMPLATE: everything in the template for Book, plus
     Fields of parameters:
     ... that.title ...        -- String
     ... that.author ...       -- Author
     ... that.price ...        -- int

     Methods on parameters:
     ... that.salePrice(int) ...   -- int
     ... that.sameAuthor(Book) ... -- boolean
  */
  ...
}
```

3.3.4 Method Body

Annoyingly, when we try to finish the design of the method we see that we do not have enough information available. We need to know whether the authors are the same — but only the `Author` class can define the method that would compare two authors.

> ***Do Now!***
>
> What information isn't available to the `Book` class, that would be needed to compute whether two authors are the same? *Why* isn't that information available?

So, we make a wish list: we need to define the method `sameAuthor` in the `Author` class:

```
// In Author
// is this the same author as the given one?
boolean sameAuthor(Author that) {...}
```

We can now add this method to the `Author` template, and from there add it to the template for `sameAuthor` in `Book`:

```
// In Book's sameAuthor
/*
```

```
    Methods for fields:
    ... this.author.sameAuthor(Author) ...    -- boolean

    Methods on parameters
    ... that.author.sameAuthor(Author) ...    -- boolean
  */
```

Now writing the method body is trivial:

```
// In Book
// is this book written by the same author as the given book?
boolean sameAuthor(Book that) {
  return this.author.sameAuthor(that.author);
}
```

Of course, before we test this method, we need to finish designing any remaining methods in our wish list:

```
// 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;
  }

  /* TEMPLATE
     Fields:
     ... this.name ...    -- String
     ... this.yob ...     -- int

     Methods:
     ... this.sameAuthor(Author) ... -- boolean
  */

  // is this the same author as the given one?
  boolean sameAuthor(Author that) {
    return this.name.equals(that.name) &&
           this.yob == that.yob;
  }
}
```

with the added tests:

```
// test the method sameAuthor in the class Author
boolean testSameAuthor(Tester t) {
  return t.checkExpect(
          this.pat.sameAuthor(new Author("Pat Conroy", 1948)),
          true)
     && t.checkexpect(this.pat.sameAuthor(this.dan), false);
```

```
  }
```

(A note about testing for equality: always test whether two `String`s are equal by invoking their `equals` method; always check whether two `int`s are equal by using the `==` operator.)

This pattern of asking another object to "help", by invoking a method on that object which requires access to information that only it actually has access to, is called *delegation*, and we'll see a lot more of this pattern in the upcoming lectures.

3.4 Methods that produce objects

Related files:
Book2.java

The methods so far produced results of the primitive type. We now look at how to design methods that produce objects.

Suppose the bookstore wants to permanently decrease the price of all books by 20%. We need a method that produces a book with the price reduced as desired.

The purpose and signature will be:

```
// In Book
// produce a book like this one, but with the price reduced by 20%
Book reducePrice() {
  ...
}
```

Here are some examples:

```
// In ExamplesBooks
// examples of books
Book htdp = new Book("HtDP", "FFK", 60);
Book beaches = new Book("Beaches", "PC", 20);

// test the method reducePrice for the class Book
boolean testReducePrice(Tester t) {
  return t.checkExpect(this.htdp.reducePrice(),
                       new Book("HtDP", "FFK", 48))
     && t.checkExpect(this.beaches.reducePrice(),
                      new Book("Beaches", "PC", 16));
}
```

We see that the result is a `new` object and so the method body will contain `return new Book(...)`. The template is as follows:

```
// In Book
/* TEMPLATE:
   Fields:
   ... this.title ...         -- String
```

```
    ... this.author ...        -- String
    ... this.price ...         -- int

    Methods:
    ... this.salePrice(int) ... -- int
    ... this.reducePrice() ...  -- Book
*/
```

Conveniently, we can reuse the method defined earlier in the body of our method as follows:

```
// In Book
// produce a book like this one, but with the price reduced by 20%
Book reducePrice() {
  return new Book(this.title, this.author, this.salePrice(20));
}
```

Of course, we finish by running the tests. Notice that the test cases now compare the values of two objects, not just the values of data of the primitive types.