

Classes and Objects

COMP1100 — Introduction to Programming and Algorithms

Reading: *Big Java Ch.2 & 3*

Clem Baker-Finch

Australian National University

Semester 1, 2006

Types, Values and “Variables” in Java

As in Haskell, every **value** in Java has a **type**. For example:

- `42` is of type `Integer`
- `"Hello, sailor!"` is of type `String`

But the concept of a **variable** in Java is different from Haskell and Mathematics:

In Java, a variable is a **location** in the store.

Locations **contain** values.

- We give *names* to variables by **declaring** them:

```
String greeting;
```

and say that `greeting` has type `String`.

- We put values *into* variables by **assignment**:

```
greeting = "Hello, World!";
```

- Usually, we **initialise** variables when we declare them:

```
String greeting = "Hello, World!";
```

What does “Object-Oriented” Mean?

Three fundamental constructs:

- **Classes**
- **Objects**
- **Methods**

Object-oriented programs manipulate **objects**.

In a *pure* object-oriented language, *every value is an object*.

Java is not purely OO, but we'll try to pretend that it is. . .

What's a *Class*?

A **class** in Java is like a **data type** in Haskell.

Haskell has various types provided by the Prelude (e.g. `Int`, `Bool`, lists, tuples) and others provided in the libraries.

There are lots of *predefined* classes in the Java libraries (e.g. `Integer`, `Boolean`, about a dozen different kinds of lists, and so on.)

(Java doesn't have any specific tuple classes. In a sense, *every* class corresponds to a tuple.)

What's a *Class*? — Part 2

A **class** in Java is also like a **module** in Haskell.

In Haskell, a common use of modules is to group together a user-defined data type *with the functions that operate on that data type*.

For example, in Assignment 1 the `Pixel` module contained the data type representing pixels *and the functions that operate on pixels*. The `PPM` module contained the data type representing ppm images *and the functions that operate on those images*.

(Of course, we expect your Assignment 2 submissions to be structured this way, too!)

Java forces you to structure your programs that way!

A Java **class** groups together the data needed to represent things of that type *with the operations defined for values of that type*.

The class definition will include **instance fields** to store the data representing things of that type.

The operations are just functions.

In OO terminology, these functions are called **methods**.

Classes also contain **constructors**, which *directly correspond to the constructor functions for algebraic data types in Haskell*.

Java constructors create new **objects** that are *instances* of that class.

What's an *Object*?

If a class is like a type, an **object** (of a class) is like a **value** (of that type).

In **Haskell**, a value is just the ***data***.

In **Java**, an object *also comes with it's own copy of the **methods** of the class*.

In Haskell, the result of functions is always **new data**.

In Java, methods may **change the values stored in the instance fields**.

Another viewpoint:

A class is like a **template**. An object is like an **instance** of that template.

An Example

Java allows classes to be grouped together and organised in **packages**.

The standard Java libraries include a package called `java.awt` (`awt` stands for **abstract windowing toolkit**).

One of the classes in that package is `java.awt.Rectangle` which allows us to represent rectangles in 2D coordinate geometry.

Objects of type `Rectangle` allow us to *represent* rectangles in Java programs.

(Exercise: How would you represent rectangles in Haskell?)

java.awt.Rectangle **Fields:**

The position and the dimensions of the rectangle are specified:

```
int height;  
int width;  
int x;           // x coord of top left corner  
int y;           // y coord of top left corner
```

(Sadly, we're already seeing Java is not purely OO ...)

java.awt.Rectangle **Constructors:**

In fact, the class provides 7 different constructors!

The most natural is this one:

```
Rectangle(int x, int y, int width, int height)
```

which just sets the instance fields to the values passed in as arguments to the constructor.

Note: since this class is part of the Java libraries, we don't see the *implementation* of the constructors. That will have to wait until we build *our own* classes.

How do we *Use* Constructors?

The expression:

```
new Rectangle(5, 10, 20, 30)
```

constructs a new `Rectangle` object at (5,10) with width 20 and height 30.

Usually, the resulting object is stored in a variable:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

This statement declares a new variable `box` of type `Rectangle`, and stores the newly constructed `Rectangle` object in that variable.

java.awt.Rectangle **Methods:**

There are *dozens* of them!

(Again, since this is a library class, we don't see the implementation of the methods. Soon, we will look at building our own classes.)

Interlude — What's a *Method*?

Methods are just the operators (functions) that allow us to manipulate objects of this class (or equivalently, values of this type).

Methods come in two flavours:

Accessor Methods

Accessor methods return some information about the object *without changing it*.

Mutator Methods

Mutator methods *change the state (the instance fields) of the object*.

(Back to the example...)

Some `java.awt.Rectangle` **Accessor Methods**:

It is common to provide **accessor** methods to return the values in each of the instance fields:

```
double getX()  
double getY()  
double getHeight()  
double getWidth()
```

(Returning `double` rather than `int` is a decision I'll leave it to the implementers to explain.)

Some `java.awt.Rectangle` Mutator Methods:

To change the dimensions of the rectangle:

```
void setSize(int height, int width)
```

To move the rectangle a distance to the right and a distance down:

```
void translate(int x, int y)
```

Notice these methods do not return a result. They **change** the object.

This is a fundamental difference from the functional approach.

Calling Methods

Suppose we have constructed a new `Rectangle` object as before:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

Remember that the methods are associated with the **object**.

If we want to access the `x` field of `box` we use the `getX()` method.

The syntax is like this:

```
box.getX();
```

which is an expression that will return the double value `5.0`

For example:

```
Double area = box.getHeight() * box.getWidth();
```

Calling Methods — ctd

How do we call mutator methods? For example, how do we use

```
void translate(int x, int y)
```

to change box?

Notice (again) from the type declaration that `translate` doesn't return another `Rectangle`.

To *change* box using the `translate` method we use the **statement**:

```
box.translate(15, 25);
```

(See the example test program: `MoveTester.java`.)

Tricky ...

In Java, variables whose type is a *class* don't actually hold an *object* — they hold a **reference** to the object.

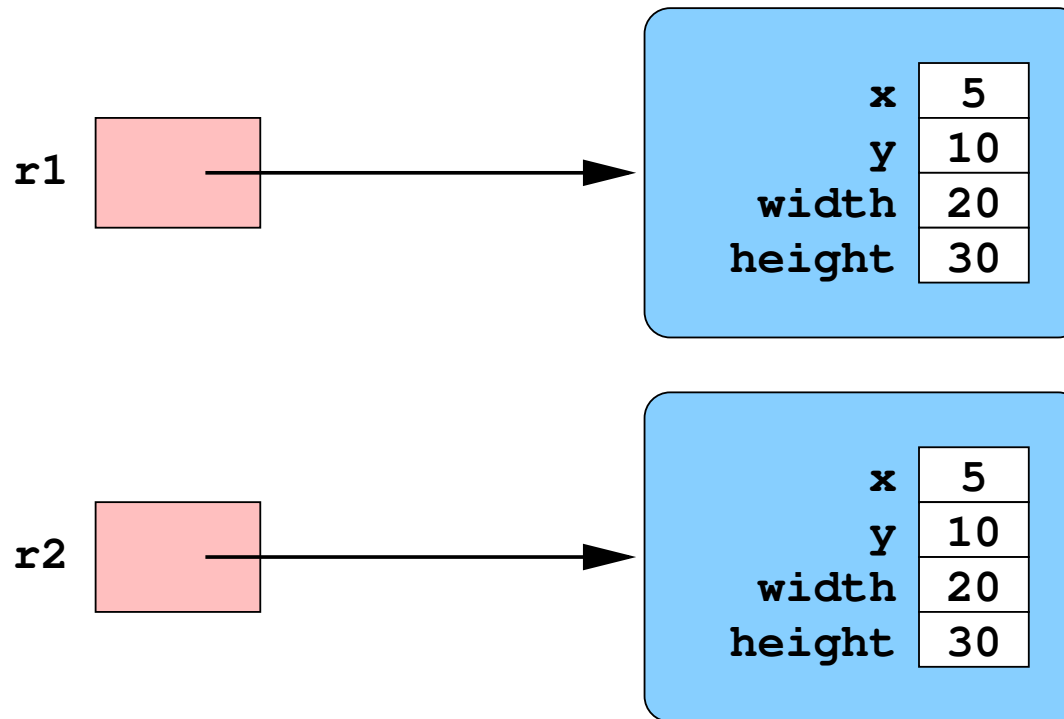
So What?

Consider:

```
r1 = new Rectangle (5, 10, 20, 30);  
r2 = new Rectangle (5, 10, 20, 30);
```

Is `r1 == r2` true or false?

A picture helps:



The rectangles are identical, but r1 and r2 refer to *separate objects*.

So `r1 == r2` evaluates to **false!**

This is called **reference equality**.

How can we test whether two objects are equal?

There is a standard method `equals(...)`

In class `Rectangle` we have:

```
boolean equals(Rectangle r)
```

Now we can ask:

```
r1.equals(r2)
```

which evaluates to ***true***.

Writing Our Own Classes

The Java libraries contain lots of useful classes.

But mostly they are *general purpose* in some way — they are a *programming toolkit*.

When designing programs, we begin by **identifying and representing the “things” — the abstract data types — that our program needs to manipulate.**

O-O Ideas Come From Data-Directed Design

which we looked at a few weeks ago . . .

Recall the Data Directed Design Process:

- 1. Understand the Problem**
- 2. Identify the (Classes of) Objects**
- 3. Identify the Basic Operations on Objects**
- 4. Choose Representations of the Objects**
- 5. Implement the Basic Operations on Objects**
- 6. Factor the Process into Manageable Parts**

In O-O languages, steps 2–6 correspond to designing and implementing classes.

1. Understand the Problem
- 2. Identify the (Classes of) Objects**
- 3. Identify the Basic Operations on Objects**
- 4. Choose Representations of the Objects**
- 5. Implement the Basic Operations on Objects**
6. Factor the Process into Manageable Parts

Example: Bank Account

Suppose that in the process of developing a particular system using the DDD method, one of the abstract data types identified in step 2 is a **bank account**.

(There are many different kinds of accounts, but let's keep it simple.)

Step 3: Identify the Basic Operations

- Deposit money
- Withdraw money
- Check the current balance

(We will leave the matter of *interest* to the lab exercises.)

Making a deposit *changes* a bank account, so it will be a **mutator method**.

The argument to the `deposit` method will be the amount of money being deposited.

Similarly `withdraw` is also a mutator method and its argument is the amount of money being withdrawn.

Checking the current balance doesn't change the account, so that will be an **accessor method**.

These observations give us the following skeleton for the BankAccount class:

```
class BankAccount {  
  
    void withdraw(Double amount) ...  
  
    void deposit (Double amount) ...  
  
    Double getBalance() ...  
  
}
```

Constructors

Setting up a new bank account is also a basic operation, so we also need to consider the `BankAccount` **constructors** at this point.

Suppose we decide that a bank account can be set up with an initial deposit of money, or (by default) with a balance of 0.

That suggests two constructors, one taking an argument being the initial amount, the other taking no argument.

```
BankAccount (Double initialBalance) ...
```

```
BankAccount () ...
```

Notice that constructors never have a result type. Notice that constructors always have the same name as the class.

The outline of the class now looks like:

```
class BankAccount {  
  
    BankAccount(Double initialBalance) ...  
    BankAccount() ...  
  
    void withdraw(Double amount) ...  
    void deposit (Double amount) ...  
  
    Double getBalance() ...  
}
```

We have now identified the basic operations on bank accounts and determined their types.

The next step is ...

Step 4: Choose a Representation

That is, *what **data** do we need to represent a bank account?*

For our simplified example, the only thing we need to keep track of is the **balance**, which we have already decided will be a Double.

In OO languages, the data representation appears as the **instance fields** of the class:

```
class BankAccount {  
    Double balance;  
    ...  
}
```

Finally ...

Step 5: Implement the Basic Operations

That is, fill in the bodies of the methods and constructors we have identified.

The **mutator** methods (`deposit`, `withdraw`) will *change* the balance field.

The **accessor** method (`getBalance`) will *not change* the balance field.

The **constructor** methods will *initialise* the balance field.

(`BankAccount.java`)

(`BankAccountTester.java`)