Language Issues for Inheritance

CS 5010 Program Design Paradigms

"Bootcamp"

Lesson 12.3

Key Points for Lesson 12.3

- Object systems in different languages may differ in many ways.
- Among these ways are:
  - inheritance of methods and fields
  - can methods or fields be final?
  - visibility of methods and fields
- Multiple inheritance raises many new issues
- Types raise even more issues.

Some dimensions of the design space for object-oriented languges

- Object systems in different languages may differ in many ways.
- In the next few slides, we will look at some of the ways in which these languages may differ from each other. We will first review the mechanisms in the Racket object system, and then see what other possibilities there are.

Methods in the subclass

- **`define/public`**
  - ordinary method definition
- **`define/override`**
  - overrides method in superclass
- **`(send obj m ...)`**
  - ordinary message send
- **`(super` *`method-name args`* `...)`**
  - allows a subclass to invoke a method of its superclass.

These declarations may appear in a subclass; they allow the subclass to refer to values in the superclass, or to override them.

Methods in the superclass

- invoke a hook method:
  - **`(send this m ...)`**
  - if **`m`** is defined in the subclass, that is the definition that will be invoked
- **`(abstract`** ***`name`*****`)`** declares *name* to be a method that must be defined in each subclass.
  - Use **`define/override`** to define the method in the subclass
  - Can use this to make the superclass satisfy an interface, or to force each subclass to define a hook method.

Fields in the subclass

- **`(init-field` *`name`*`)`**
  - field to be supplied at initialization
- **`(init-field [`*`name default-value`*`])`**
  - field that may be supplied at initialization; otherwise gets the default value
- **`(field [`*`name initial-value`*`])`**
  - field whose initial value is determined by initial-value
- **`(inherit-field` *`name`*`)`**
  - field that is inherited from the superclass. Undefined until the superclass is initialized with **`super-new`**.

Fields in the superclass

Here are some of the ways fields of the superclass can be initialized.

- **init-fields**:
  - if field is not an init-field of the subclass, then field is initialized from the (new Sub% ...) expression.
- fields:
  - initialize it w/ **`(super-new [`*`field value`*`]..)`**
  - then inherit it back down into the subclass [see idiom for this on next slide]
  - this works if it's static (but our images weren't)
- Racket does not have something like **`abstract-field`**
  - Use a hook method, like **`(get-image)`**

3 ways to share fields in Racket

Initialize in superclass, inherit into subclass:

```
(define super%
 (class …
   (init-field x)
   …x…))


(define sub1%
 (class super% …
   (super-new)
   (inherit-field x)
   …x…))
```

Initialize in subclass, access via hook method:

```
(define super%
 (class …
   (init-field x)
   (abstract get-x)
   …(send this get-x)…))


(define sub1%
 (class super% …
   (super-new)
   (init-field x)
   (define/override
     (get-x) x)
   …x…))
```

Initialize in superclass:

```
(define super%
 (class …
   (init-field x)
    …x…))


(define sub1%
 (class super% …
   (super-new
    [x “this is sub1”])
   …x…))
```

From Racket to other OO Languages

- This completes our survey of the Racket class system.
- Let's take a look at some design dimensions on which languages may differ.
- For each design decision, you should be able to identify how it is handled:
  - in Racket
  - in your favorite OO language

Observe that most so-called "Object-oriented languages" are in fact "class-oriented"– Much of the complexity in the language stems from decisions involving classes. There are some true O-O languages, which do not involve classes. You should go research some of these.

Language Design: Inheritance of Methods

- Must we specify that we are overriding?
- Can we specify that some methods are not overridable ("final")?
- Can we specify that some methods are required to be supplied by each subclass ("abstract")?
  - either because it's a required hook or required to implement an interface.

Language Design Issues: Inheritance of Methods

- If the language has types, must a method have the same type (contract) as the method it overrides?
  - In Racket, we don't have types, like Java does, so this isn't an issue.
  - We need only worry about the new method having the same number of arguments.
  - Other languages have complex rules about this
  - This course is about program design, not language design, so all this is beyond the scope of this course.
  - Some hints below, under "Interfaces and Contracts"

Language Design Issues: Visibility of Methods

- In Racket, methods defined using **define/public** are visible throughout the current module (in Racket, this is the same as the current file.)
- It might be nice to restrict the visibility of methods, so that we could have methods that were local inside a class, just like we can define local functions inside a function.
- Object-oriented languages often have elaborate visibility rules, like **public**, **private**, or **protected** in Java. These are beyond the scope of this course.
- You should be familiar with the visibility and scoping rules of the language you are working with, and be aware of the best practices in that language.

Language Design Issues: Inheritance of Fields

- The same kinds of design issues come up for fields as well as methods. On the next slide, we list some of the ones we've looked at.
- Every object-oriented language must make decisions about each of these.
- A language may make different decisions for fields than it does for methods.

Language Design Issues: Inheritance of Fields

- What fields of the superclass are visible in the subclass?
- How do we make fields of the subclass visible in the superclass?
- Can we specify that some fields are not overridable ("final")?
- Can we specify that some fields are required to be supplied by each subclass ("abstract")?

Language Design Issues: Interfaces

- Can we specify that a class must implement multiple interfaces?
  - Generally: yes.
  - An interface is just a list of methods that the class must provide definitions for, so that's ok so long as there isn't an overlap between the interfaces.
- If our language has types (contracts), what is the relation between the type in the interface and the type in the class?

Interfaces and Contracts

```
in interface:
    on-tick : -> Shape<%>
in a class Rect% implementing
Shape<%>:
    on-tick : -> Rect%
```

This is ok, since **Rect%** implements **Shape<%>**

Interfaces and Contracts

```
in interface:
   adjoin-right : Shape<%> -> Shape<%>
   RETURNS: a shape like this one, but
    with the given shape added on the
    right

in a class Rect% implementing Shape<%>:
   adjoin-right : Rect% -> Rect%
```

Probably not OK: does the interface specify that the given shape is of the same class as this one?

Understanding situations like this was an important research issue in programming languages for a long while. There's still no consensus on how it should be treated.

Interfaces and Inheritance

- Should we allow inheritance between interfaces?
  - This would allow us to construct an interface incrementally.
  - Racket allows this
  - But what about types? All the same issues arise.

Language Design Issue: Multiple Inheritance

- Having a class implement multiple interfaces is ok.
- But what about inheriting from multiple superclasses?
- Some languges allow this. Most (including Racket and Java) do not.
- What goes wrong?

Problems with multiple inheritance (1)

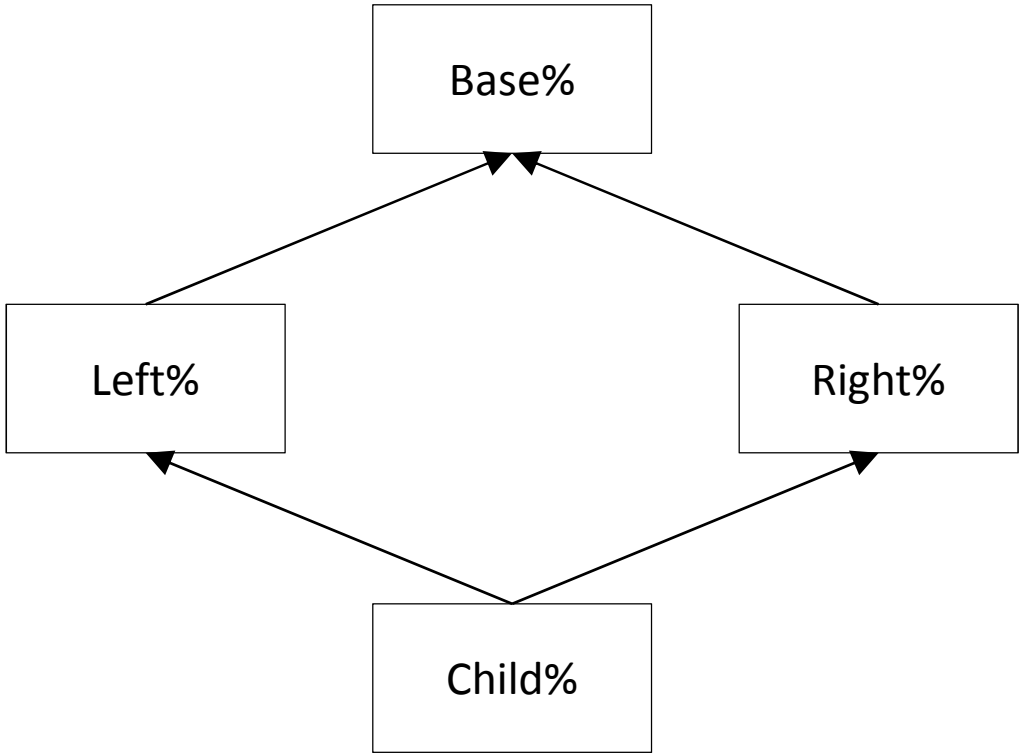
- If a method is defined in multiple superclasses, which definition do you choose?
- Here are some possible designs:
  - Have the language fix the policy (e.g. inherit methods from superclasses from left to right)
  - Have the class specify the policy for all the methods in the class.
  - Choose at each call site
- Different languages handle it differently

Problems with Multiple Inheritance (2)

- Your superclasses may also have superclasses.
- You might inherit from the same super-superclass twice!
- If you inherit from a superclass twice, how many objects of the superclass do you build?

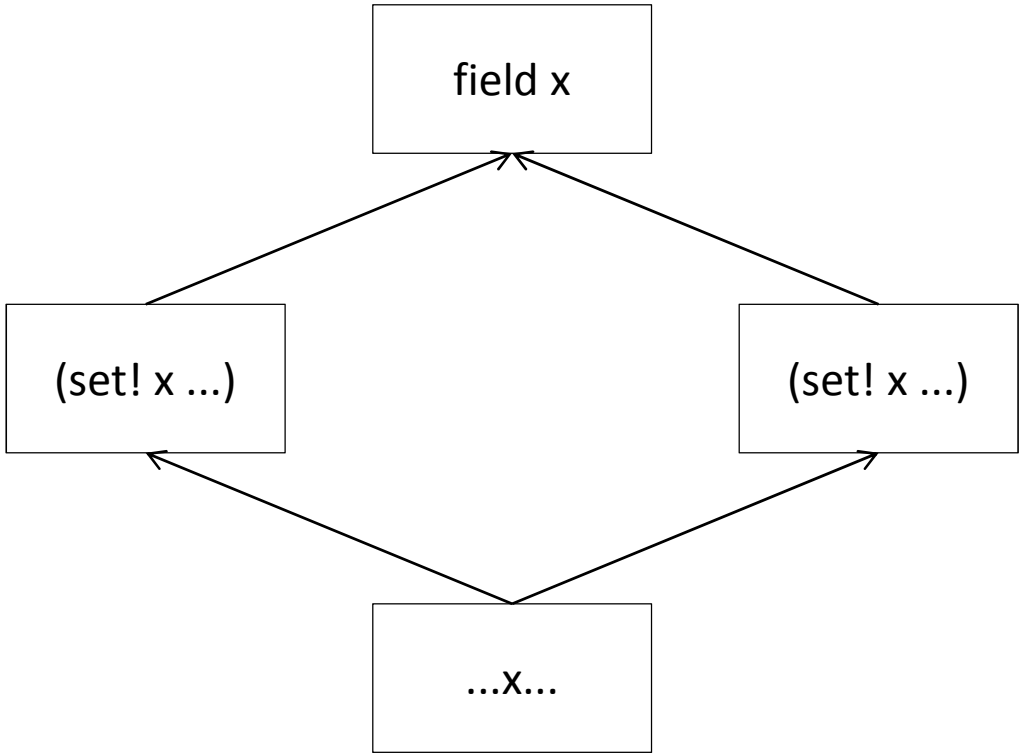
Problems with Multiple Inheritance (3)

- Here's an inheritance diagram that illustrates this:



Problems with Multiple Inheritance (4)

- How many copies of Base%'s fields should you have?



How many copies of Base%'s field should you have in a Child% object?

- Let's say that the class Base% defines a field x, which is inherited into Left% and Right%.
- Left% uses the field x for its own purposes; it doesn't know that it will eventually become a superclass of Child%.
- Similarly, Right% uses the field x for its own purposes, which may be different than the way Right% uses it.
- Child% uses methods from both Left% and Right%. Those methods both use an x field, but they may use it incompatibly.
- So it seems like we need two copies of Base%, so Left% and Right% can each use the field x without interfering with each other.

But what if Child% tries to inherit the field x?

- It could be inheriting one of the x's or the other.
- It could inherit both, and every reference to x would have to specify which one it was referring to
- Or it could be that Child%'s intention was to have Left% and Right% *share* information through x.

Different languages with multiple inheritance make different choices. Each choice makes some programs easy to design and other programs harder to design. There's no one right answer.

But single inheritance is non-modular

- Single inheritance is non-modular: it forces you to choose exactly one superclass, but unrestricted multiple inheritance leads to nasty problems.
- Some languages have restricted forms of multiple inheritance that avoid these problems.
  - *mixins* essentially allow a class to have a superclass that is a parameter rather than being fixed. This solution goes back to the 1980's. (Racket has these)
  - *traits* are generalized interfaces that are allowed to define concrete methods, but not fields. Scala is an up-and-coming language that does this.
- Our goal is not to teach you about these language features, but just to make you aware that these choices exist.