

Types

A Type is a Set of Values

Consider the Asteroid statement:

```
let n:%integer = 3.
```

Here we constrain n to take on any value from the set of all integer values.

Types

Def: A type is a set of values.

Def: A primitive type is a type that is built into the language, e.g., integer, string.

Def: A constructed type is a user defined type, e.g., any type introduced by the user.
In Asteroid this is done through the 'structure' statement.

Example: Java, primitive type

<code>float q;</code>	}	q is of type float, only a value that is a member of the set of all floating point values can be assigned to q.
<u> </u>		
type float \Rightarrow set of all possible floating point values		

Types

Example: Java, constructed type

```
class Rectangle { int xdim; int ydim; };
```

```
Rectangle r = new Rectangle();
```



Now the variable r only accepts values that are members of type Rectangle;
☞ object instantiations of class Rectangle.

Types

Example: Asteroid, constructed type

```
structure Rectangle with  
  data xdim.  
  data ydim.  
end  
  
let r:%Rectangle = Rectangle(4,2).
```

an element of
type Rectangle.

Types

In statically typed languages arrays are also considered 'constructed types'

Example: C, constructed type

int a[3];

the variable a will accept values
which are arrays of 3 integers.

e.g.: `int a[3] = {1,2,3};`
`int a[3] = {7,24,9}`

That is, 'int a[3]' defines the set of all integer arrays of size three.

Subtypes

Def: a subtype is a subset of the elements of a type.

Example: Java

The notation $A < B$ means
A is a subtype of B.

Short is a subtype of int: $\text{short} < \text{int}$

Observations:

- (1) converting a value of a subtype to a values of the super-type is called widening type conversion. (safe)
- (2) converting a value of a supertype to a value of a subtype is called narrowing type conversion. (not safe)

Example: Java

$\text{float} < \text{double}$

Subtypes give rise to type hierarchies and type hierarchies allow for automatic type coercion – widening conversions!

Subtypes & Type Hierarchies

- In type hierarchies it is always safe to move from subtype to supertype – widening conversion (coercion)
 - E.g. short < int ✓
- Never the other way around
 - E.g. int < short ✗

char < short < int < float < double

Part of the Java type
hierarchy

Subtypes

- The Asteroid type hierarchy
 - `boolean < integer < real < string`
 - `list < string`
 - `tuple < string`
 - `none`
 - constructed types

Why do we use types?

- Types allow the computer/language system to assist the developer write better programs.
Type mismatches in a program usually indicate some sort of programming error.
 - Static type checking – check the types of all statements and expressions at compile time.
 - Dynamic type checking – check the types at runtime.

Type Equivalence

I. Name (nominal) Equivalence – two objects are of the same type if and only if they share the same type name.

Example: Rust

```
1 struct Type1 {x:i64, y:i64}
2 struct Type2 {x:i64, y:i64}
3
4 fn main () {
5     let x: Type1 = Type1{x:1,y:2};
6     let y: Type2 = x;
7     println!("{:?}",y);
8 }
```

Error; even though the types look the same, their names are different, therefore, Rust will not compile.

☞ Rust uses name equivalence

Type Equivalence

II. Structural Equivalence – two objects are of the same type if and only if they share the same type structure.

Example: Haskell

```
1  type Type1 = (Integer, Integer)
2  type Type2 = (Integer, Integer)
3
4  x :: Type1
5  y :: Type2
6
7  x = (1, 2)
8  y = x
```

Even though the type names are different, Haskell correctly recognizes this statement.

☞ Haskell uses structural equivalence.

Type Inference

- Type inference refers to the automatic detection of the data type of an expression in a programming language.
- To see how this might work let's work through an example.

Type Inference

- Assume we have the following statements in a programming language like Asteroid:

let x : integer = 3.

let y : integer = 2 * x.

- We want to make sure that all the assignments are legal.
- We will use the type notation '3.{integer}' indicating that this syntactic unit has the type integer.

Type Inference

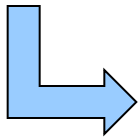
- We start at the primitives on the right side of the assignments of the first statement and then stepping through all the remaining statements

Type Inference

let x:%integer = 3.{integer}.
let y:%integer = 2 * x.



let x:%integer = 3.{integer}.
let y:%integer = 2 * x.



let x:%integer = 3. ✓
let y:%integer = 2.{integer} * x.{integer}.



let x:%integer = 3. ✓
let y:%integer = 2 * .{(integer,integer)→integer} x.



let x:%integer = 3. ✓
let y: %integer = 2 * .{(integer,integer)→integer} x. ✓



let x:%integer = 3. ✓
let y:%integer = 2 * x. ✓

Type Inference

- Let's try a program with a bug in it:

```
let x:%boolean = 3.
```


Type Inference

let x:%boolean = 3.{integer}.



let x:%boolean = 3.{integer}. ✗

Recall Asteroid's type hierarchy: `boolean < integer < real < string`

We are not allowed to assign a supertype to a subtype!

Types & Objects

- In any OO language class definitions create new types
- Objects are the values in those types
- In OO languages that support inheritance, inheritance creates a subtype-supertype relationship in the class hierarchy

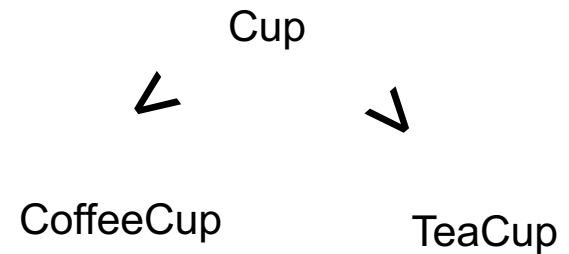
Types & Objects

Example: Java

```
class Cup { ... };  
class CoffeeCup extends Cup { ... };  
class TeaCup extends Cup { ... };
```

Which ones of the following statements are safe and which ones are not?

1. Cup x = new Cup();
2. Cup y = new CoffeeCup();
3. TeaCup z = new Cup();
4. TeaCup t = new TeaCup();
 Cup c = t;



Notation:
 $A < B$ means A is subtype of B

Note: Type coercion in type hierarchies gives rise to polymorphic programming in OO - objects can appear in different type contexts.

Object-Oriented Programming

- Most of the programming languages we use to today are inheritance based OO languages
- The main distinguishing feature between them is whether they support single or multiple inheritance.
 - C++ and Python support multiple inheritance
 - Java support single inheritance
- There are three main problems with inheritance based OO languages.

Problem #1

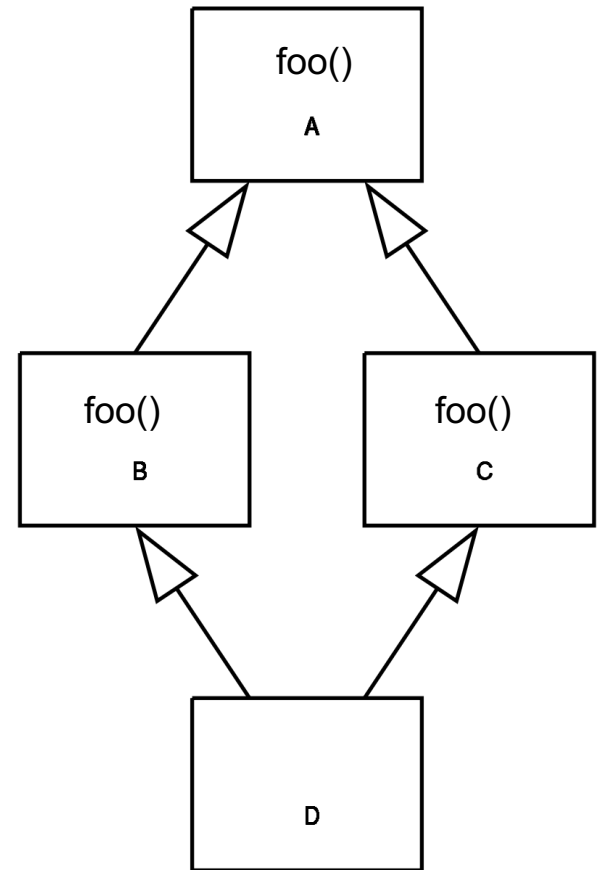
- *Bloated method inheritance* – that is, each child in an inheritance hierarchy will inherit ALL of the methods of its ancestors.
- This is true for both single and multiple inheritance.

Problem #2

- The *diamond problem* – sometimes referred to as the ‘deadly diamond of death’
- This occurs in languages with multiple inheritance

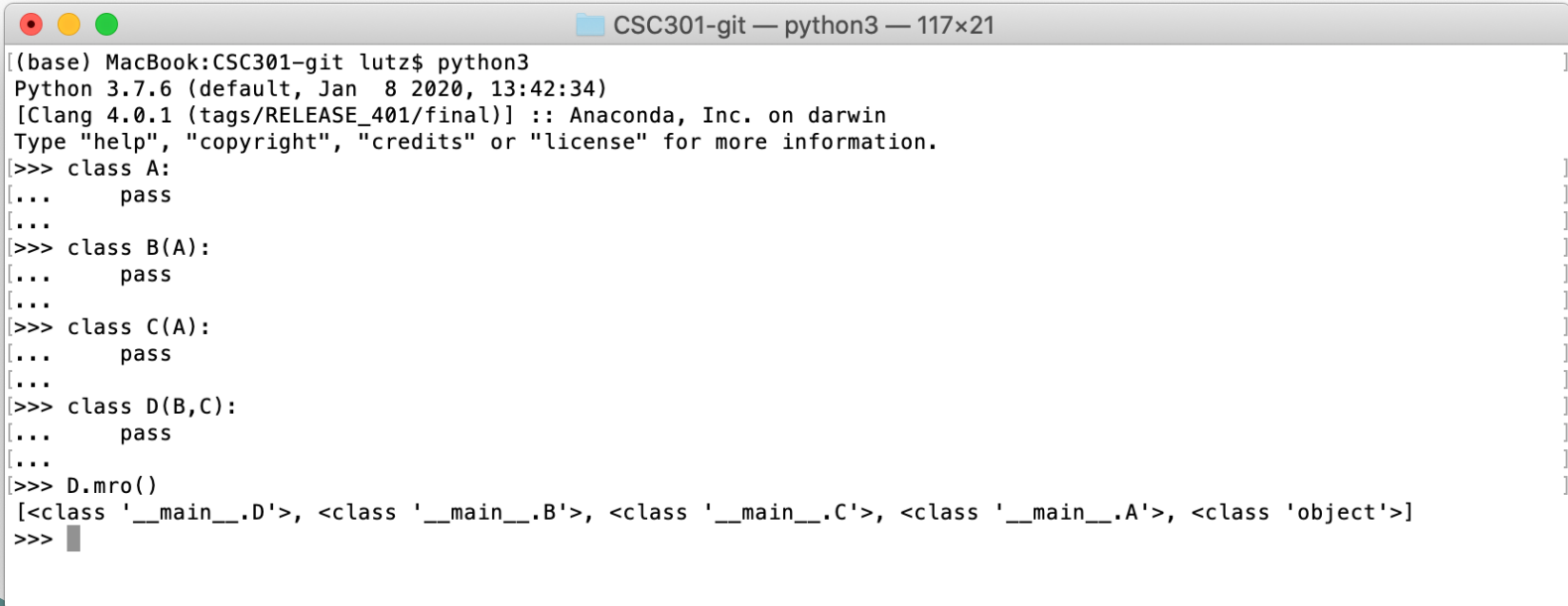
The Diamond Problem

- Briefly:
 - An ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C.
 - If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?
 - That is: D.foo() – which foo() should be called?
- This gets really problematic in deep inheritance structures.



The Diamond Problem

- Different languages deal with the diamond problem in different ways
 - C++ uses a fully qualified syntax
 - Python uses a class hierarchy linearization algorithm (C3 linearization or MRO) to resolve ambiguities



```
CSC301-git — python3 — 117x21
(base) MacBook:CSC301-git lutz$ python3
Python 3.7.6 (default, Jan  8 2020, 13:42:34)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> class C(A):
...     pass
...
>>> class D(B,C):
...     pass
...
>>> D.mro()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
>>>
```

MRO: Method Resolution Order

Problem #3

- A third problem that frequently arises in inheritance based OO languages are *rigid class structures*
 - This usually manifests itself in class hierarchies that are difficult to evolve in face of changing software requirements

Object-Based Programming

- A response to these problems is that recent languages no longer support inheritance and are object-based
- Of the three new big languages, Rust, Go, and Swift, only Swift supports a full OO model.
- Asteroid is object-based, that is, it supports objects but not inheritance.

Exercises

- Describe the type associated with the set of values $\{-1, -2, -3, -4, \dots\}$, call this type Q.
- Describe the type associated with the set of values $\{-2, -4, -6, -8, \dots\}$, call this type P.
- Is there a subtype-supertype relationship between these types? If so, what is it?
- Let x be a variable of type Q and y be a variable of type P, then is the assignment
$$x := y$$
a safe assignment? Why? Why not?

Hint: A type is a set of values!

Take Away

- Types are sets of values, typically with a common representation and common set of operations.
- Types in programming languages allows compilers and interpreters to check for consistency in your programs.
- Inconsistencies usually show up as type mismatches.
- Type equivalence between constructed types can be established in one of two ways, name equivalence or structural equivalence.
- Class hierarchies in OO languages give rise to subtype-supertype relationships due to inheritance.

Assignments

- Assignment #2 – See BrightSpace