NTNU – Trondheim
Norwegian University of
Science and Technology
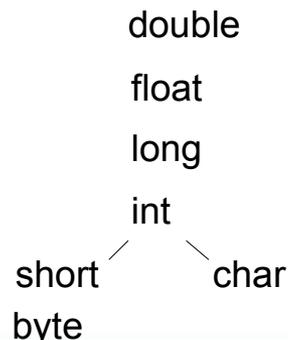
Derived and abstract data types

Where we were

- We've looked at static semantics for primitive types
    - and how it relates to type checking
- We've hinted at derived types
    - using a multidimensional array declaration as example
- We'll round of with a quick look at more complicated derived types, and what they do to the type system

NTNU – Trondheim
Norwegian University of
Science and Technology

Type expressions

- We've touched upon this

    int[2][3] consists of a basic type int, and (array-of-2 (array-of-3))

- Types can be constructed from basic types

- We've also mentioned that types can be converted into each other according to a hierarchy

    (short + char) converts into their least upper bound (int), we can write it that as a function, lub(short,char)

double

float

long

int

short   char

byte

NTNU – Trondheim
Norwegian University of
Science and Technology

Arrays can be many things

No idea how many entries here

↓

- Unbounded

  Java: `public static void main ( String args[] )`

- Fixed size

  C: `char mystring[256];`
  (Type of mystring contains its size)

- Ranges

  Ada: `array[2 .. 5] of integer`
  (Type contains offset for indexing)

- Multidimensional

  Fortran: `REAL, dimension(2,3) :: A`
  (Type contains shape, to check that B is (3,2) and C is (2,2) if you write C=matmul(A,B))

NTNU – Trondheim
Norwegian University of
Science and Technology

Records

- Records are collections of names and types
  - { id1:T1, id2:T2, … , idn:Tn}
- C calls them *struct*
- *Objects*, at their simplest, are just records extended with references (pointers) to the functions which manipulate their contents
  - ...and some calling syntax that doesn't let you invoke its methods without passing an instance to manipulate...

NTNU – Trondheim
Norwegian University of
Science and Technology

Type constructors

- When the program can define types, the compiler must be able to construct representations of them, to
  - Put in symbol tables
  - Consult for conversions
  - Etc.

- Generate a structure representing the type at its declaration

- Bind names to these structures when variables are determined to be of the matching type

NTNU – Trondheim
Norwegian University of
Science and Technology

Object types

- The type of a record is the cartesian product of all its component types

- Looking at objects as glorified records, their types are the same way, just add method signatures

```
Class Point {
    float x, y;
    float getX() { return x; }
    float getY() { return y; }
}
becomes
{ x: float, y: float, getX : fun(Point → float), getY : fun(Point→float) }
```

Wait a minute...

{ x: float, y: float,

    getX : fun(**Point** → float), getY : fun(**Point**→float)

}

      These are not in the argument lists of <u>float getX()</u> or <u>float getY()</u>

– Implementation generates one static lump of code for taking x-s out of Points

– The reason it acts differently with different instances is that the instance is passed as a hidden argument:

    Point p = new Point(3,2);

    p.getX()   ↔   Point.getX (p)

– Inside the method body, the hidden argument can go by the name "*this*"

NTNU – Trondheim
Norwegian University of
Science and Technology

Objects imply more

Class ColoredPoint extends Point {

    int color;

    int getColor() { return color; }

}

seen as an extended record works out to

{ x: float, y: float, color: int, getX : fun(Point → float), getY : fun(Point→float), getColor : fun(ColoredPoint → int) }

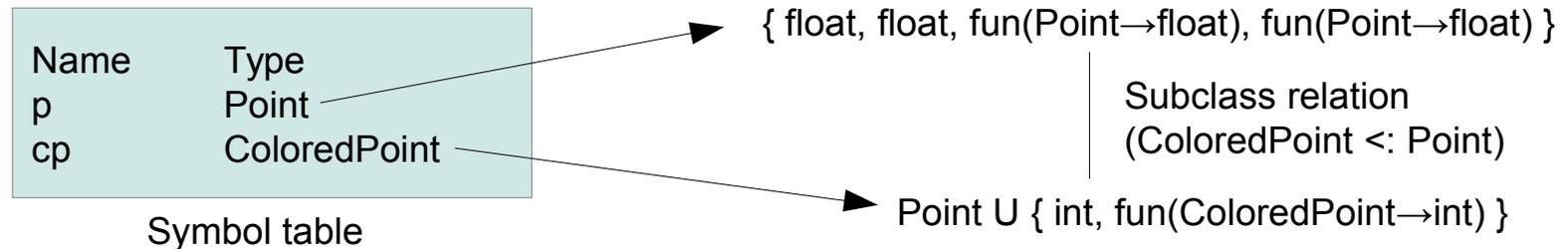That alone doesn't tell us what to do here:

ColoredPoint cp = new ColoredPoint (3.0, 4.0, red);

Point p;

p = cp;

Things to support

- Inheritance, subclassing, polymorphism, overloading, abstract classes, interfaces, *etc.* create a hierarchy of derived types

- Type-checking of comparisons, method calls, assignments, *etc.* must take this programmer-defined hierarchy into account

Name	Type
p	Point
cp	ColoredPoint

Symbol table

{ float, float, fun(Point→float), fun(Point→float) }

Subclass relation
(ColoredPoint <: Point)

Point U { int, fun(ColoredPoint→int) }

NTNU – Trondheim
Norwegian University of
Science and Technology

Assignment, revisited

- We had

  $$\frac{id : T\ |\text{-}\ E : T}{id : T\ |\text{-}\ id = E : T}$$

- Adding subclasses, it becomes

  $$\frac{id : T\ |\text{-}\ E : S\quad where\ S <: T}{id : T\ |\text{-}\ id = E : T}$$

Impact on static checking

- The type of an object reference isn't known at compile time
  - That's what a class hierarchy is for, different types can step in for each other
- Overridden methods appear with multiple implementations

  and *sligthly* different call signatures

- Resolution requires some policy on how much dynamic information to account for

NTNU – Trondheim
Norwegian University of
Science and Technology

An example with Java

```
Class Animal {
    String voice; void speak() {
        System.out.println ( voice );
    }
}
gives type { voice : String, speak : fun(Animal→void) }


Class Cat extends Animal {
    String voice = "meow";
}
gives type { voice : String, speak : fun(Animal→void) }
```

NTNU – Trondheim
Norwegian University of
Science and Technology

Same field name, inherited method

- Fields are statically resolved, not overridden
- Method calls are dynamically dispatched using the run-time type of the instance, and parameter list + return type

```
Class Animal {
     String voice;              Statically resolved
     void speak() { println ( voice ); }
}
Class Cat extends Animal {
     String voice = "meow";
}


(new Cat()).speak();           // ← this prints "null"
// Looking at the Cat type redirects the method call to Animal.speak
```

Overridden method

```
Class Animal {
    String voice;
    void speak() { println ( voice ); }
}
Class Cat extends Animal {
    String voice = "meow";        Statically resolved
    void speak() { println ( voice ); }
}


(new Cat()).speak();          // ← this prints "meow"
// At run time, look at instance and detect that speak : fun(→void) means
// fun ( Cat→void ) instead of fun(Animal→void)
```
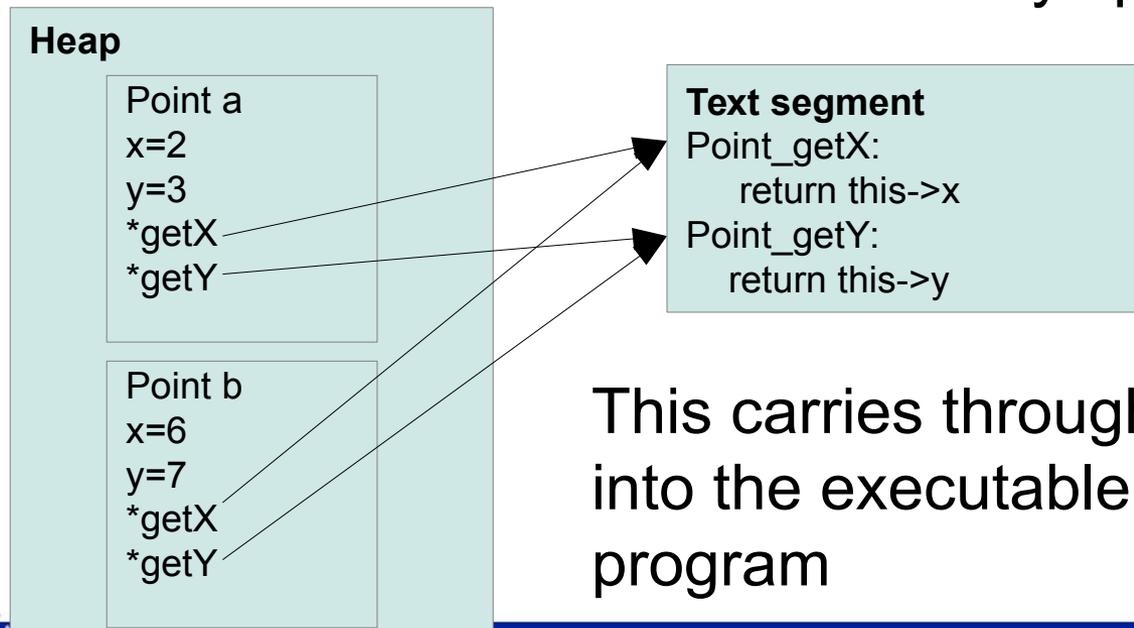
NTNU – Trondheim
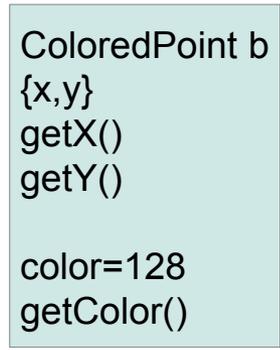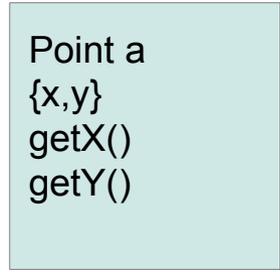Norwegian University of
Science and Technology

Objects as records

- One thing the compiler must deal with is how to lay out instances in memory

- The types of fields and methods indicate what instances must look like in a flat memory space

```
Heap
  Point a
  x=2
  y=3
  *getX
  *getY

  Point b
  x=6
  y=7
  *getX
  *getY
```

```
Text segment
Point_getX:
    return this->x
Point_getY:
    return this->y
```

This carries through into the executable program

NTNU – Trondheim
Norwegian University of
Science and Technology

Objects as abstract data types

– Checking, say, that
a.getX() == b.getY()
is correctly typed
requires a structure that
shows how both have
methods like that, and
that they return comparable
numbers

Point a
{x,y}
getX()
getY()

Point
<:
ColoredPoint

ColoredPoint b
{x,y}
getX()
getY()

color=128
getColor()

This may be discarded
after compilation
(well, depending on the language)

NTNU – Trondheim
Norwegian University of
Science and Technology

Summary of the week (in reverse)

- Representation of types
  - Objects make the compiler build a class hierarchy
  - Derived types make the compiler construct user-defined type descriptors
  - Basic types can be hard-coded in a predefined hierarchy

- Static type checking
  - Static type checking goes through all expressions to determine that the syntax attaches them to meaningful types
  - With an attribute grammar, it can be done during parsing

- *(Static semantics are a subset of natural semantics)*
  - *You just remove the states*

NTNU – Trondheim
Norwegian University of
Science and Technology

That's a lot to take in

- M'kay, I'm almost finished prattling on about types now

- All that remains is to take one short peek at how a simple single inheritance scheme and dynamic call dispatch can map into low-level code
  - After we've looked closer at low-level code

NTNU – Trondheim
Norwegian University of
Science and Technology

Is all this really necessary?

- Erm… I know we've been skipping along the borderline of the syllabus here.

- If you know your L and S attributions and can suggest how to represent simple things in symbol tables, it'll be just fine

NTNU – Trondheim
Norwegian University of
Science and Technology

What was it for, then?

- Most practical languages define a far richer notion of types than just skipping around the syntax tree and seeing that there are ints and bools in the right places

- Those are what people actually use

- Starting from a vague guess at how their compilers and run-time systems work, it's easier to make improved guesses over time
  - The ability to second-guess compilers and run-time systems is a coveted skill among software developers

NTNU – Trondheim
Norwegian University of
Science and Technology

Next time

- Three-address code
  - Which is an abstract cousin of assembly programming