CS 206

Lecture 17 – Types continued

In this class, let’s

- Think more about types

- Talk about (static) type safety in O-O languages

Aside: Reflection

- Discussion last time on what part of enum exists at run time
- Generally, languages like C throw away information about higher-level structure of the program
- Languages with “Reflection” retain this information. Programs can “see” (parts of) their higher-level selves at runtime.

Static checking

- Type checking is generally done right after parsing
- One expression/statement checked at a time
- So type checking rules are generally quite simple
- Type checking system specifies what the types are and how types can legally be combined in the fundamental constructs of the language

```
x = y; // type checks x and y
x = y op z; //checks if expression is legal, coerces types if necessary
x = f(a, b, c) // checks if types of actual & formal params are compatible
```

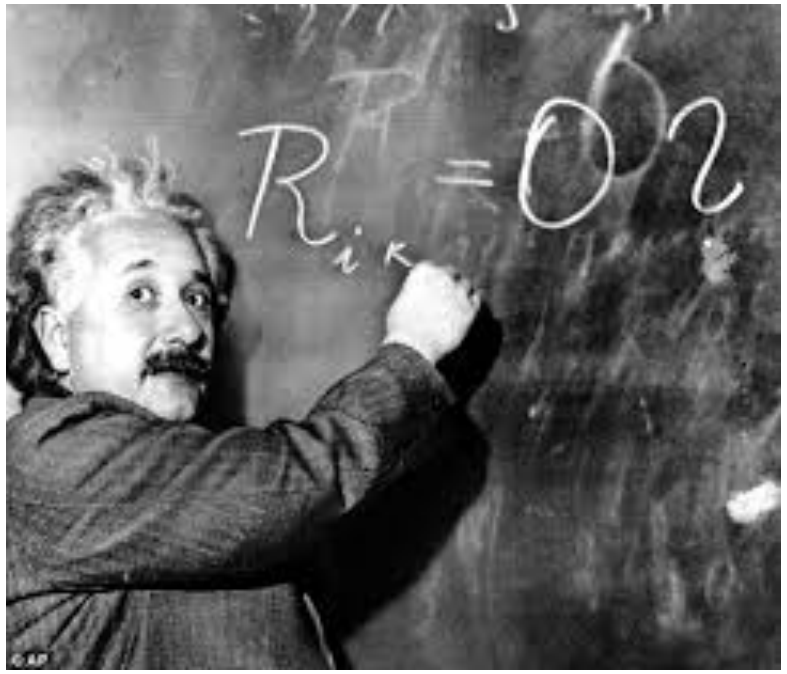
Finer grained typing

- Static typing raises the level of abstraction of the program and allows us to write programs at a higher level without worrying about (some) low-level errors
- Abstraction could be any level we want
- e.g. normally check type of check int, float, char, etc.
- Slightly finer-grained: unsigned int, not-null pointer (notice “polymorphism”: unsigned int is-a int, not-null pointer is-a pointer)
- Also note: is-a relationships are not symmetric

Finer grained typing

- Program-specific types: Person, Book, Shuttle, etc.
- PL / programmer needs to define the typing rules between these types (e.g. inheritance relationships, transformers)
- Some PLs allow for programmers to write some type checking rules themselves

What about physics dimensions?



```
float e, m, c;
…
e = m * c * c * c;
```

“Type checking” can easily tell this program is wrong

Many fine-grained types in programs: sizes, dates, colors, IDs, counts, positions, masks, ports, flags, states, file names, host names, addresses, properties, messages.

OO type safety

- Assuming static type checking, which statements will type-check at compile time?

(a) Book b = new Book()

(b) Book b = new Textbook()

(c) Textbook tb = new Book()

(d) Textbook tb = new Textbook()

Not all bugs can be caught at compile time

What if

```
Book b = null;
if (x == 1)
    b = new TextBook();
…
if (x == 1)
    Textbook tb = b;
```

How will you fix it?

Dynamic casts

- Dynamic casts are used when compiler cannot guarantee type-check at compile-type
- But programmer knows something more and guarantees it herself
- Runtime check needed in type-safe languages

```
Book b = null;
if (x == 1)
    b = new TextBook();
…
if (x == 1)
    Textbook tb = (TextBook) b;
```

Does this work?

```
class Animal { ... }
class Horse extends Animal { void neigh() { ... } }
class Elephant extends Animal { void trumpet() { ... } }
...
Animal a = b ? new Horse() : new Elephant(); // b is some condition
if (b)
    a.neigh();
```

The fix

```
class Animal { ... }
class Horse extends Animal { void neigh() { ... } }
class Elephant extends Animal { void trumpet() { ... } }
…
Animal a = b ? new Horse() : new Elephant(); // b is some condition
if (b)
    ((Horse) a).neigh();
```

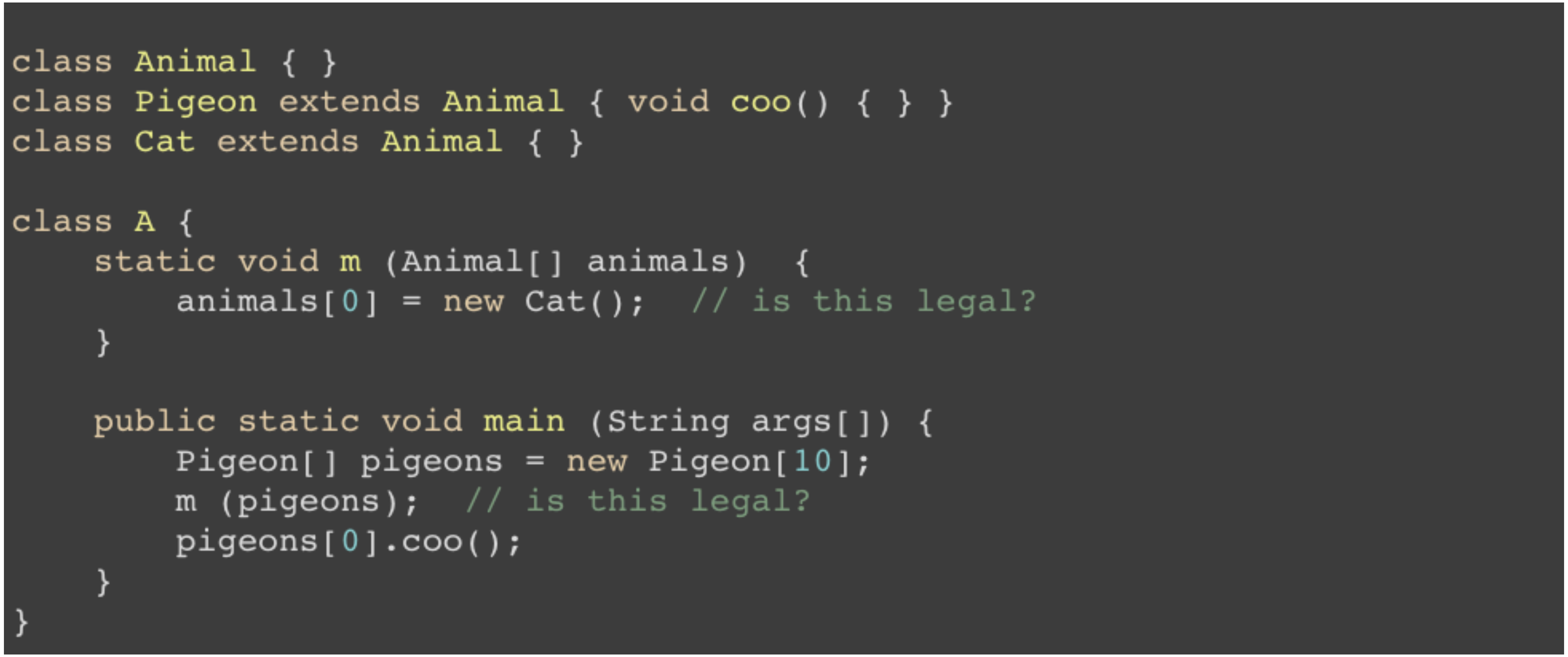
Casts in type-safe languages

- Checked casts: compiles, but type safety is maintained at run time
- Runtime error if runtime object not compatible
- Use sparingly! Loses the benefit of compile-time checking

Is-a relationships

- Cat is an animal; is an array of cats an array of animals?

Is-a relationships



```java
class Animal { }
class Pigeon extends Animal { void coo() { } }
class Cat extends Animal { }

class A {
    static void m (Animal[] animals)  {
        animals[0] = new Cat();  // is this legal?
    }

    public static void main (String args[]) {
        Pigeon[] pigeons = new Pigeon[10];
        m (pigeons);  // is this legal?
        pigeons[0].coo();
    }
}
```

Interfaces

- Declaration of an interface (no implementation)
- No fields, methods, etc.
- Like an abstract class
- Is-a hierarchy just like classes
- Classes can implement multiple interfaces

Interfaces

```
interface Readable { ... }
interface ReadableAndPrintable extends Readable { ... }
class Book implements Readable { ... }
class Textbook extends Book { ... }

// which of the following will compile?
Book b1 = new Readable();
Textbook tb = new Book();
Book b2 = new Textbook();
Readable r = new Book();
ReadableAndPrintable rp = new TextBook();
```