Object Oriented Programming

Polymorphism

Programación II
Facultad de Ingeniería
Universidad Austral

Polymorphic Methods

```
Student s = new Student();
Person p = (Person)s;
p.dance();
```

- Assuming Person has a dance() method, what should happen here?

Demo: revisit expressions from last time

- General problem: when we refer to an object via a parent type and both types implement a particular method: which method should it run?

Polymorphism: values and variables can have more than one type

```
int eval(Expression e) {

}
```

can be Literal, Mult or Plus

Polymorphic Concepts I

- **Static polymorphism**
  - Decide at compile-time
  - Since we don't know what the true type of the object will be, we just run the method based on its static type

```
Student s = new Student();
Person p = (Person) s;
p.dance();
```

- Compiler says “p is of type Person”
- So p.dance() should do the default dance() action in Person

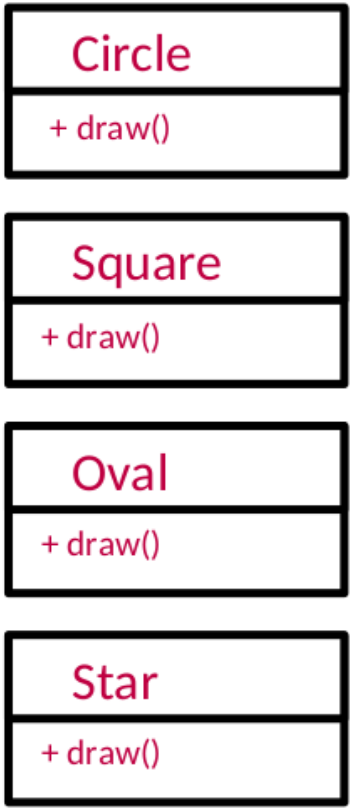
Polymorphic Concepts II

- **Dynamic polymorphism**
  - Run the method in the child
  - Must be done at run-time since that's when we know the child's type
  - Also known as ‘dynamic dispatch’

```
Student s = new Student();
Person p = (Person) s;
p.dance();
```

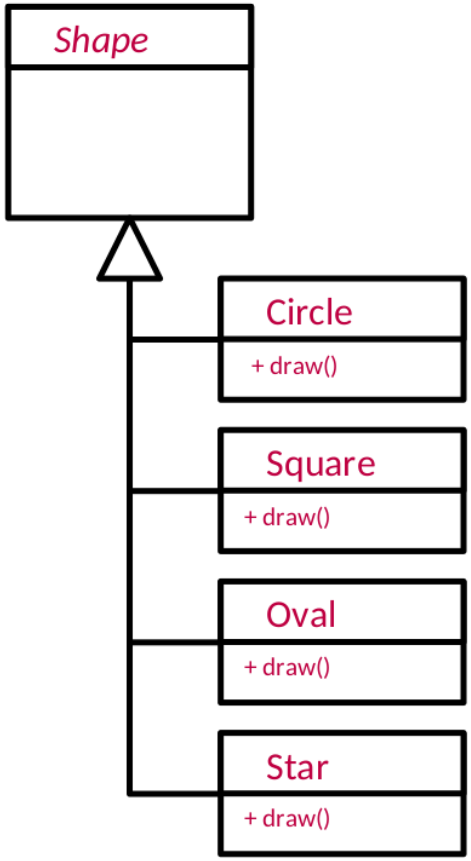
- Compiler looks in memory and finds that the object is really a Student
- So p.dance() runs the dance() action in Student

The Canonical Example I



- **A drawing program that can draw circles, squares, ovals and stars.**
- **It would presumably keep a list of all the drawing objects.**
- **Option 1**
  - Keep a list of Circle objects, a list of
  - Square objects,…
  - Iterate over each list drawing each object in turn
  - What has to change if we want to add a new shape?

The Canonical Example I

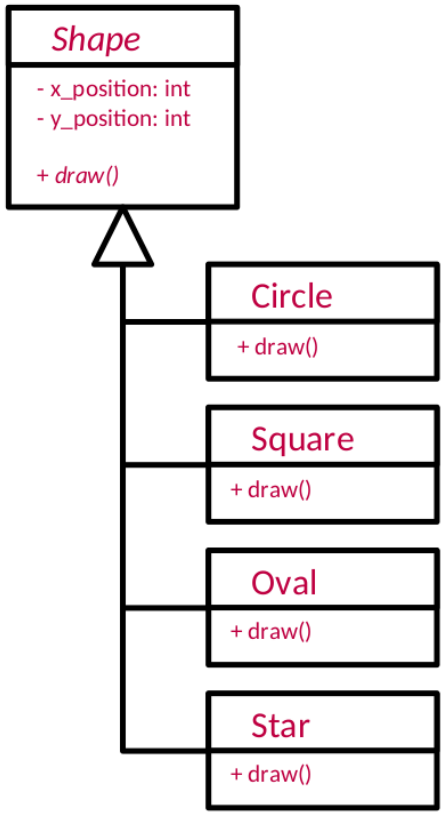


- Option 2
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

```
for every Shape s in myShapeList
    if (s is really a Circle)
        Circle c = (Circle)s;
        c.draw();
    else if (s is really a Square)
        Square sq = (Square)s;
        sq.draw();
    else if...
```

  - What if we want to add a new shape?

The Canonical Example I



- Option 3 (Polymorphic)
  - Keep a single list of Shape references
  - Let the compiler figure out what to do with each Shape reference

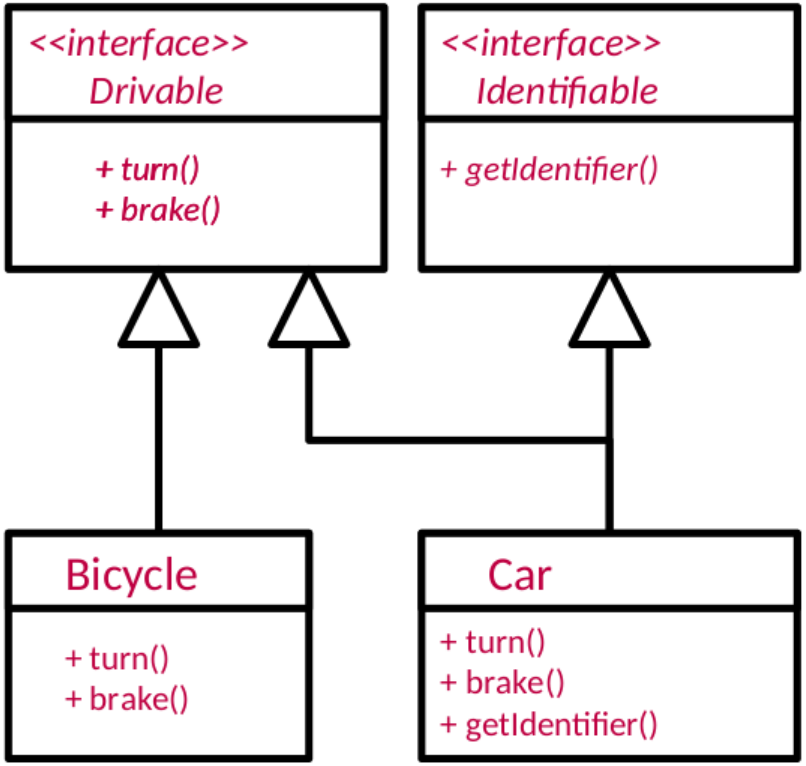
```
For every Shape s in myShapeList
  s.draw();
```

- What if we want to add a new shape?

Interfaces

- **Classes can have at most one parent. Period.**
- **But special 'classes' that are totally abstract can do multiple inheritance – call these interfaces**

Interfaces




```
interface Drivable {
  public void turn();
  public void brake();
}

interface Identifiable {
  public void getIdentifier();
}

class Bicycle implements Drivable {
  public void turn() {...}
  public void brake() {... }
}

class Car implements Drivable, Identifiable {
  public void turn() {...}
  public void brake() {... }
  public void getIdentifier() {...}
}
```

adjective

This is type inheritance (not code inheritance)

Interfaces have a load of implicit modifiers

```
interface Foo {
    int x = 1;
    int y();
}
```

means

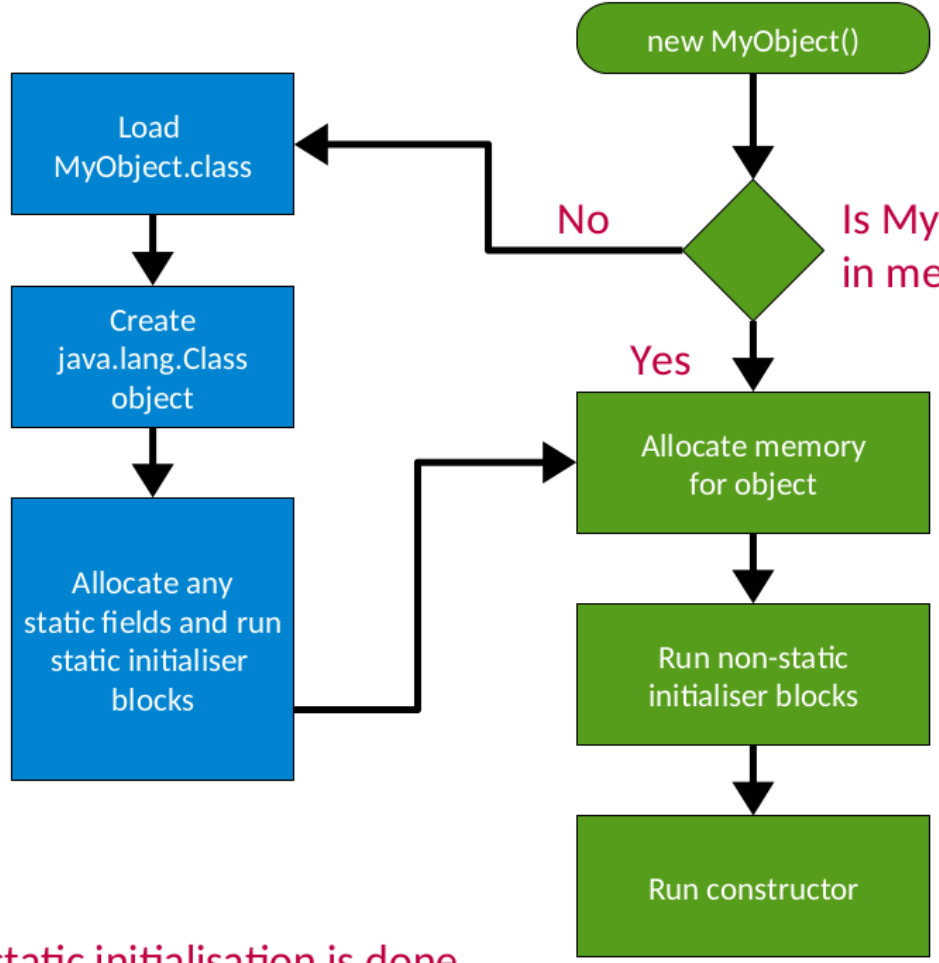
```
interface Foo {
    public static final int x = 1;
    public int y();
}
```

Interfaces can have default methods

```
interface Foo {
    int x = 1;
    int y();
    default int yPlusOne() {
        return y() + 1;
    }
}
```

- Allows you to add new functionality without breaking old code
- If you implement conflicting default methods you have to provide your own

Creating Objects in Java

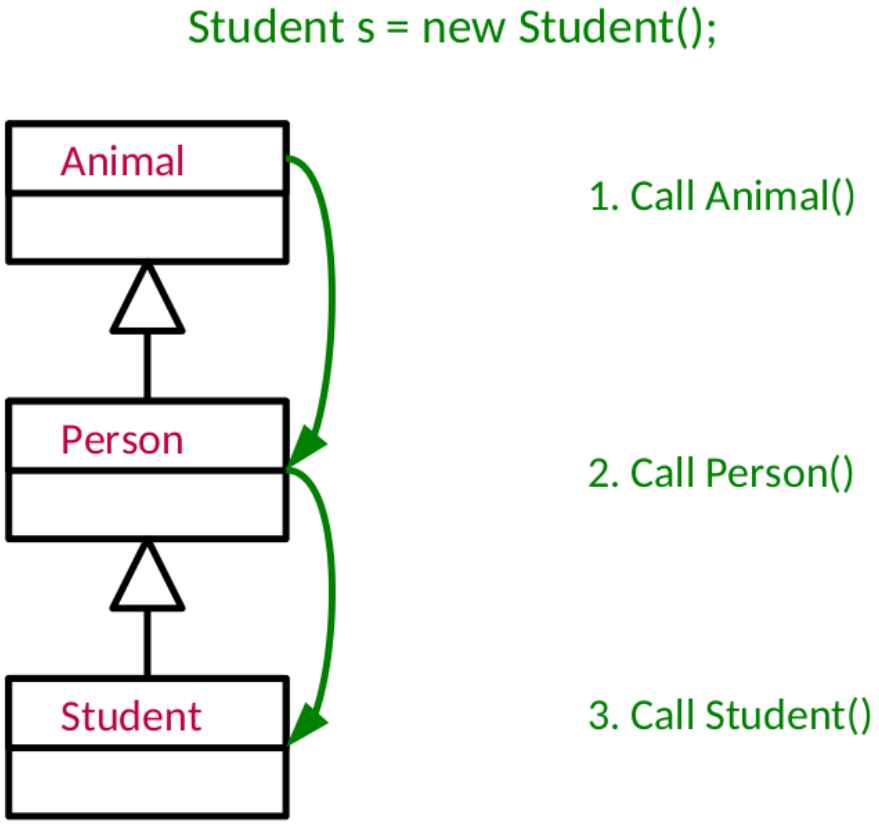


static initialisation is done in textual order

demo ObjectConstruction
demo InheritedConstruction

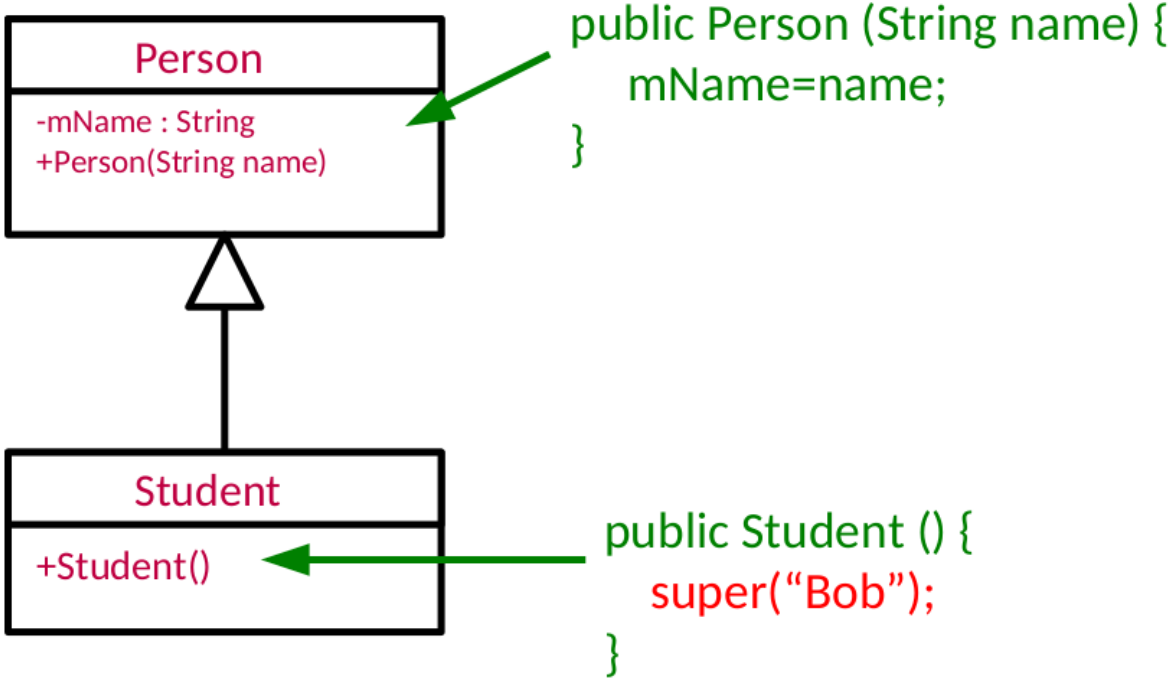
Constructor Chaining

- **When you construct an object of a type with parent classes, we call the constructors of all of the parents in sequence**



Chaining without Default Constructors

- What if your classes have explicit constructors that take arguments? You need to explicitly chain
- Use super in Java:



Object destruction and garbage collection

- **Non-Deterministic Destruction**
  - Deterministic destruction is easy to understand and seems simple enough. It turns out we humans are rubbish of keeping track of what needs deleting when
  - We either forget to delete (→ memory leak) or we delete multiple times (→ crash)
  - We can instead leave it to the system to figure out when to delete
    - “Garbage Collection”
    - The system somehow figures out when to delete and does it for us
    - In reality it needs to be cautious and sure it can delete. This leads to us not being able to predict exactly when something will be deleted!!
  - This is the Java approach!!

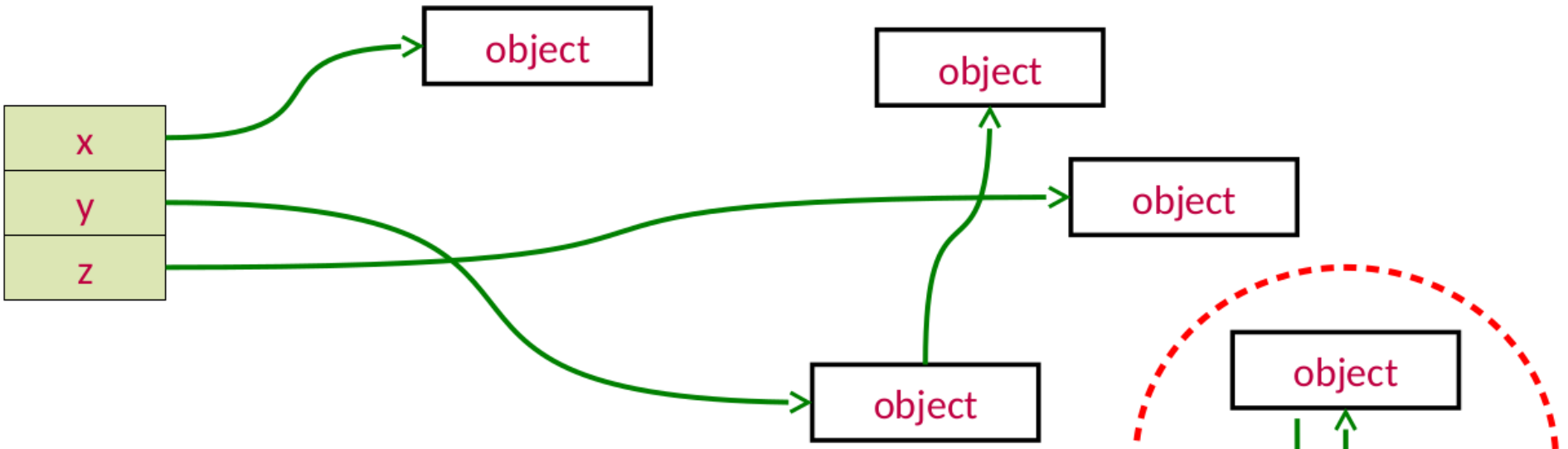
Garbage Collection

- **So how exactly does garbage collection work? How can a system know that something can be deleted?**
- **The garbage collector is a separate process that is constantly monitoring your program, looking for things to delete**
- **Running the garbage collector is obviously not free. If your program creates a lot objects, you will soon notice the collector running**
  - Can give noticeable pauses to your program!
  - But minimises memory leaks (it does not prevent them…)
- **Keywords:**
  - ‘Stop the world’ - pause the program when collecting garbage
  - ‘incremental’ - collect in multiple phases and let the program run in the gaps
  - ‘concurrent’ - no pauses in the program

Mark and sweep

- Start with a list of all references you can get to
- Follow all references recursively, marking each object
- Delete all objects that were not marked

Mark and sweep



Generational garbage collection: split objects into short-lived and long-lived and collect short-lived more frequently

Unreachable so deleted


Questions?