(Re)Learning Object Oriented Programming

with Go

Jaime Silvela

August, 2017

Agenda

- Origin of OO
- Some pitfalls of OO
- Recommendations
- OO with Go

Origins of OO

Non-OO programming

- Data
  - Simple: ints, floats, strings
  - Structs aka. records
  - Arrays
  - Maps aka. hashes aka. associative arrays
- Functions

Case study 1: video game

- Alice is writing the main animation loop, Bob and Chuck are writing each a different foe.

```
bob_foe{…}
chuck_foe{…}
extra_state{…}

forever {
   clear_canvas()

   move_uniformly(&bob_foe)
   bob_display_on_canvas(&bob_foe, &canvas)
   move_relativistically(&chuck_foe, &extra_state)
   blah_blah(&extra_state)
   chuck_display(&chuck_foe, &canvas)
   …
```

Case study 1: video game

- Alice needs to learn how to use each type of foe.
- Implementation details are all over the place.
- We could inadvertently use the wrong function for a foe.
- If we had, say, 10 types of foe we’d have a mess on our hands.

Case study 2: a database connection library

- Don has written a database connection library. Emma needs to query a database.

```
dbConn {
   hostname: foo.bar,
   username: admin,
   password: 12345,
   ConnectionPool: …
}

main () {
   …
   query(“my query”, &dbConn)
   …
}
```

Case study 2: a database connection library

- Emma learns she can manipulate the Connection Pool to get faster queries.

```
dbConn {
    hostname: foo.bar,
    username: admin,
    password: 12345,
    ConnectionPool: …
}

main () {
    …
    emma_query(“my query”, &dbConn.ConnectionPool)
    …
}
```

Case study 2: a database connection library

- Don has found a much better way to build the Connection Pool. His new release is much faster … and breaks Emma’s code.

```
dbConn {
    hostname: foo.bar,
    username: admin,
    password: 12345,
    NewConnectionPool: …
}

main () {
    …
    emma_query(“my query”, &dbConn.AAAARGHH)
    …
}
```

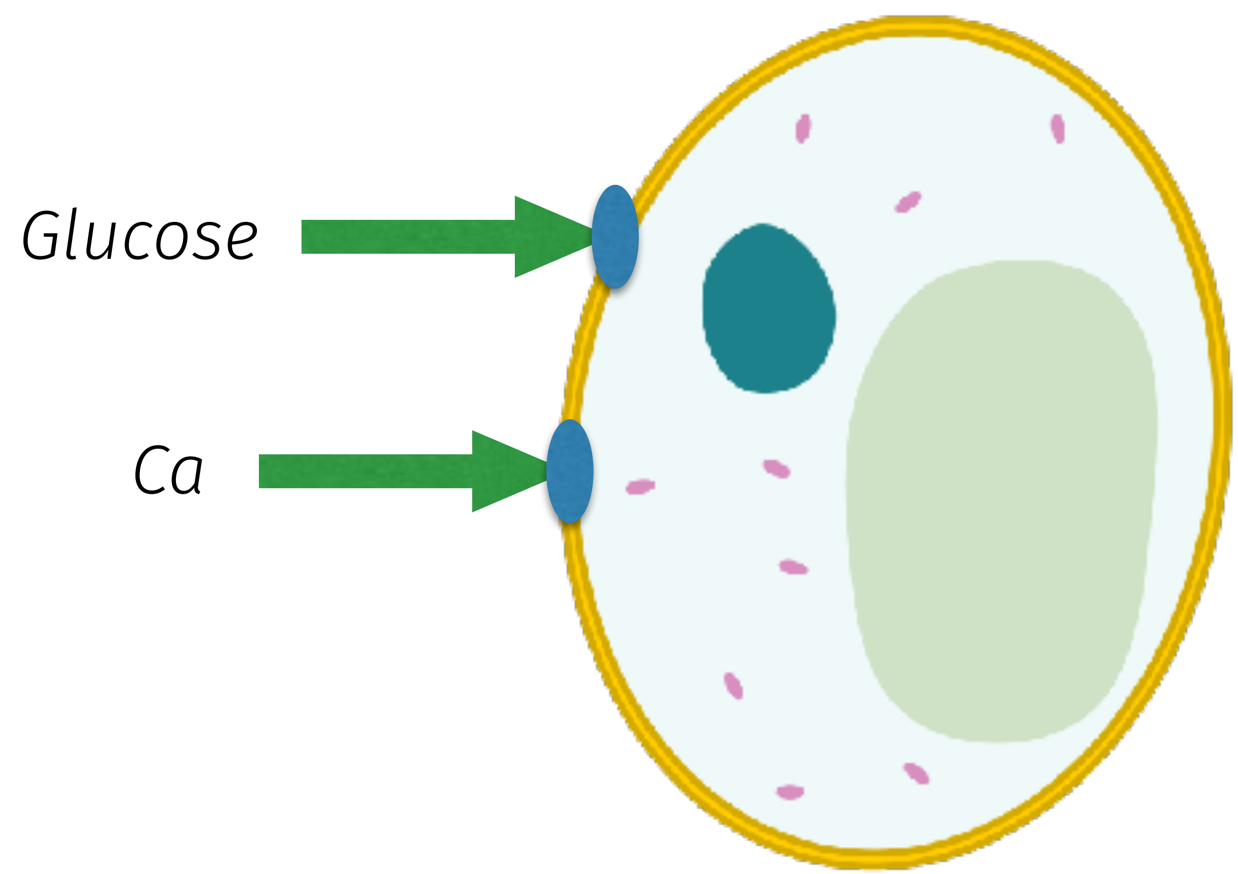
Case study 2: a database connection library

- The leaking of implementation details ends up causing unwanted coupling.
- Change becomes difficult gradually; codebases become fossilized.

The genesis of OO

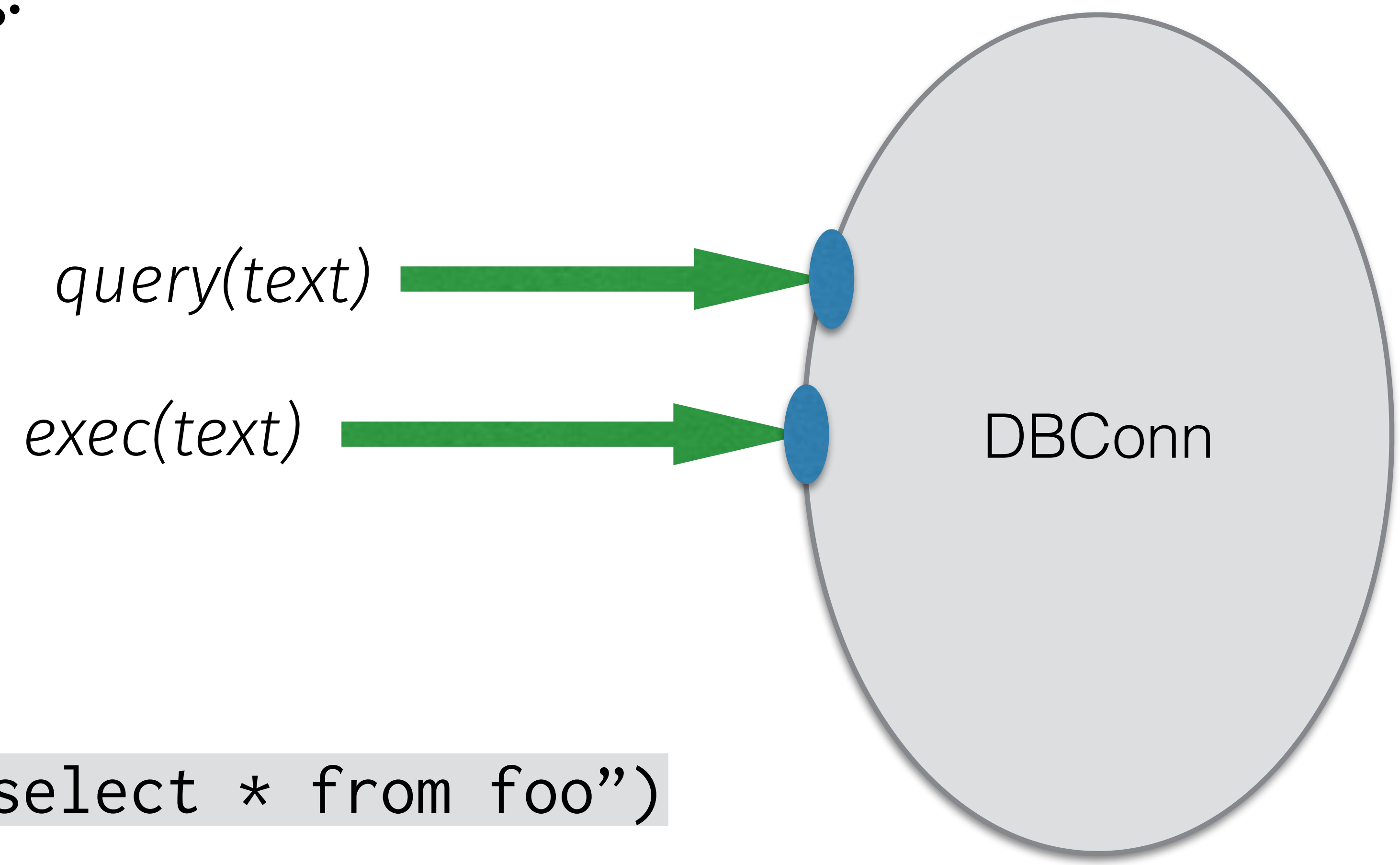
- Precursors: Simula, Sketchpad (1960s)
- Alan Kay, biological inspiration … Smalltalk (1970s)

Message Passing



Message Passing: DB connections

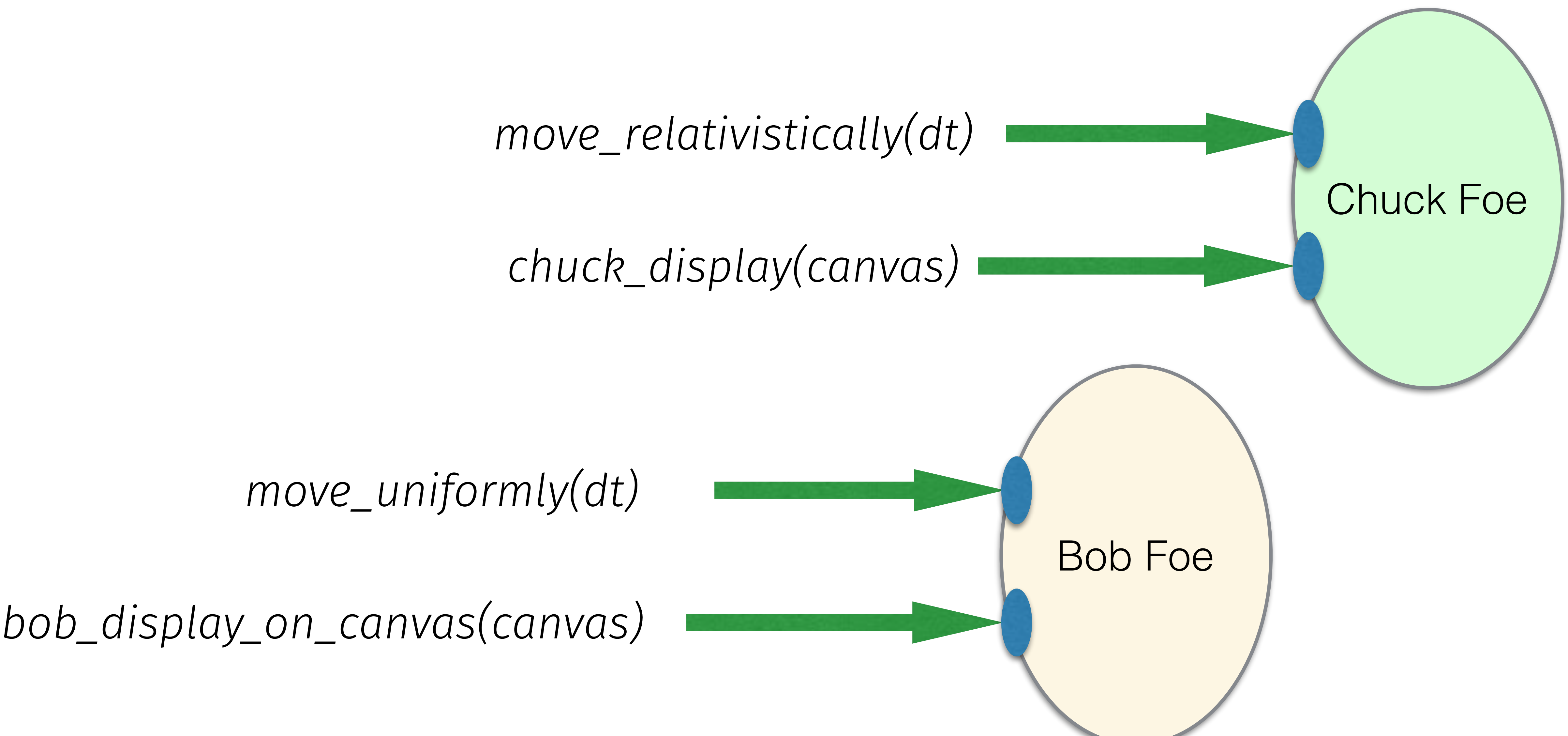
- Cells and messages … objects and methods.
- Information hiding.




```
DBConn12.query(“select * from foo”)
```

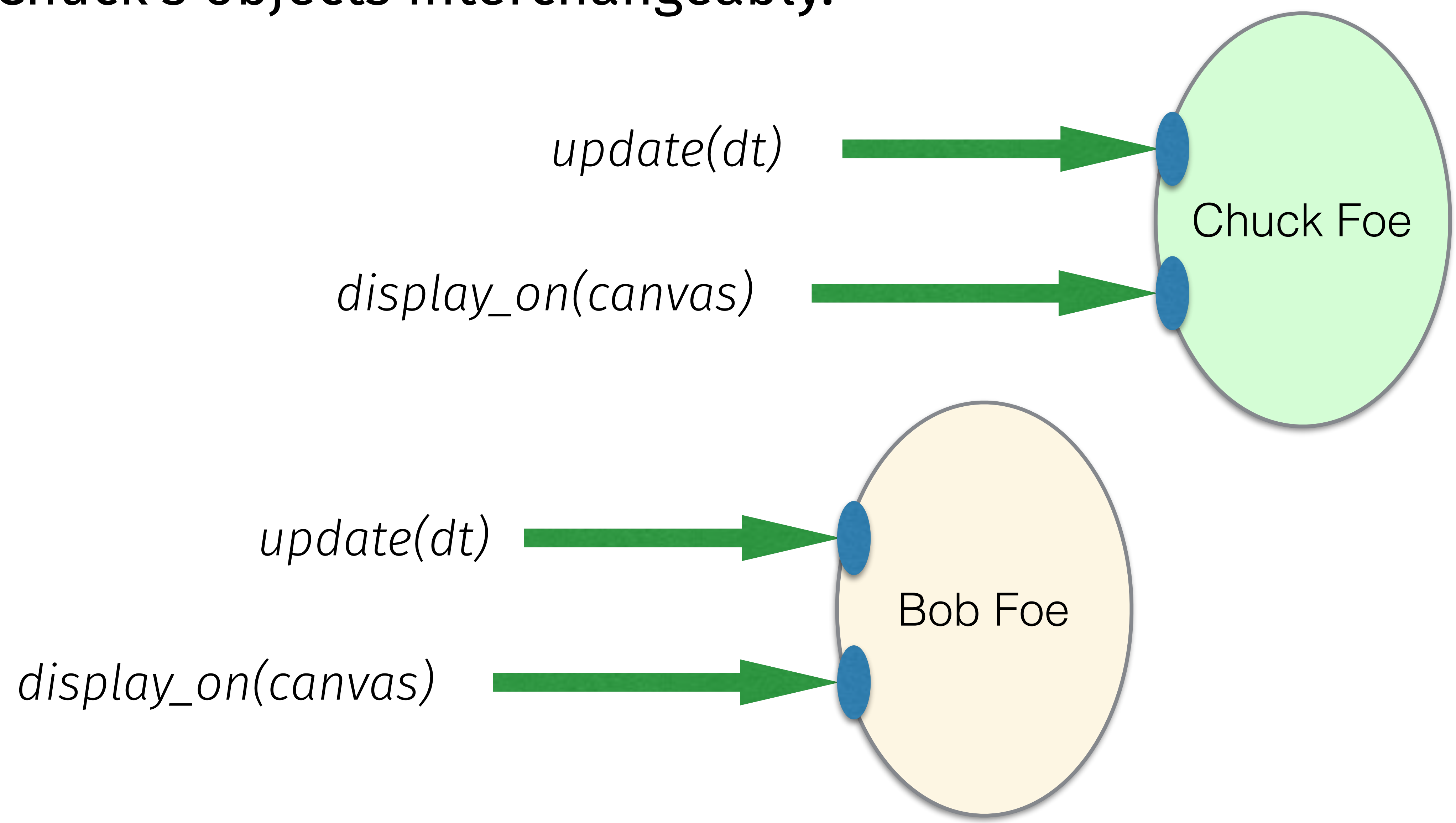
Message Passing: video game

- The `extra_state` Chuck uses is now out of sight. The objects Chuck and Bob wrote start to look similar ...



Message Passing: Polymorphism

- We unify the interface. An OO language can now treat Bob and Chuck's objects interchangeably.



Message Passing: video game loop

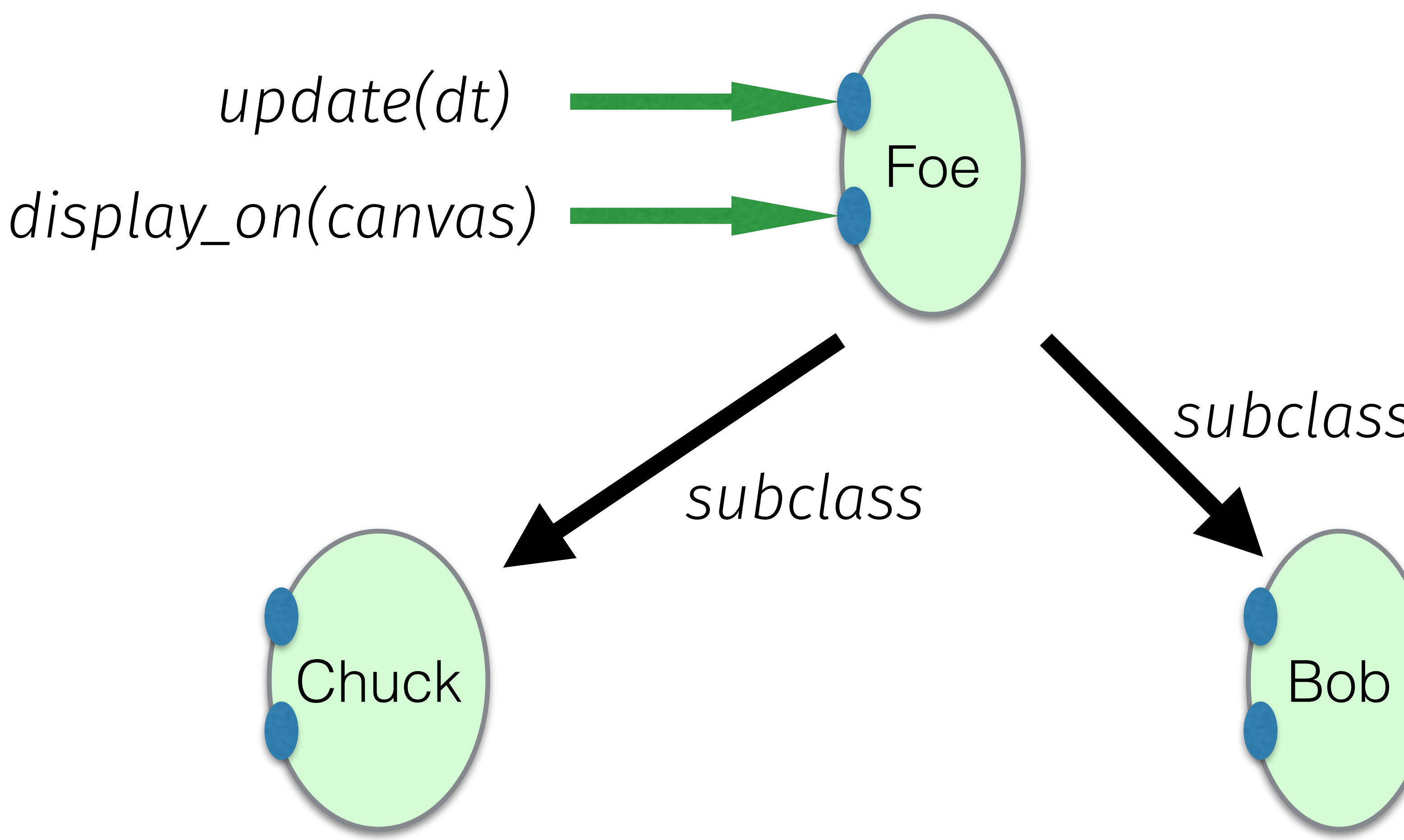
```
foes = [bob_foe, chuck_foe]

forever {
   clear_canvas()

   foreach foe in foes {
      foe.update(dt)
      foe.display_on(canvas)
   }
}
```

Message Passing: video game

- Alice doesn’t need to change her loop to support new foes.
- Conventional interfaces enable mixing and matching: Alice, Bob and Chuck can offer their modules to other parties.

Modern OO languages

- Object types are called “classes”, objects are “instances”.
- Polymorphism is generally conceived as class inheritance.



Some pitfalls with OO

Complexity

- Class inheritance has become an end, rather than the means for polymorphism. Bloated class hierarchies.
- Complexity in the languages:
  - virtual / abstract
  - private / protected / public
  - static / class methods vs. instance methods
  - copy-constructors, references

Dogma

- “Everything is an Object” mantra, even in places where it doesn’t work well.
- Design Patterns used where simple solutions would be much better.
- Programmers begin design with a taxonomy of classes.
- Don’t Repeat Yourself (DRY) used to justify abuse of inheritance.

Misuse of inheritance

```
class Point2 {
    public final double x;
    public final double y;

    Point2(double x, y) {
        this.x = x;
        this.y = y;
    }

    public double length() {
        return sqrt(x^2 + y^2);
    }
    …
```

```
class Point3 extends Point2 {
    public final double z;

    Point3(double x, y, z) {
        super(x, y);
        this.z = z;
    }

    public double length() {
        return sqrt(z^2 +
           super.length()^2
        );
    }
    …
```

Misuse of inheritance

- Only a programmer could think this was a good idea.

```
    // class Point2 contd.
    …
    public double sqrlen1() {
        return x^2 + y^2;
    }

    public double sqrlen2() {
        return length()^2;
    }
}
```

```
    // class Point3 contd.
    …
    // inherits sqrlen1
    // inherits sqrlen2
}
```

```
Point3 pt3 = new Point3(0, 0, 5);
pt3.sqrlen2();  // 25
pt3.sqrlen1();  // 0
```

Recommendations

Prefer composition to inheritance

- Stated in the seminal (and dangerous) book *Design Patterns* by Gamma et al. 1994.

```
class Point3c {
    public final Point2 base;
    public final double z;

    Point3c(double x, double y, double z) {
        this.base = new Point2(x, y);
        this.z = z;
    }

    public double length() {
        return sqrt(base.length()^2 + z^2);
    }
}
```

Prefer to inherit from virtual classes

- If we forgo inheritance of implementation, we use class inheritance only for polymorphism, as should be.
- The DRY principle sometimes used as justification to look at inheritance of implementation for “code re-use”. RESIST!

Liskov Substitution Principle

- Introduced by Barbara Liskov, 1987:

Let $\Phi(x)$ be a property provable about objects x of type T. Then $\Phi(y)$ should be true for objects y of type S where S is a subtype of T.

- Make sure your hierarchies actually obey an "is a" relationship. *(eg. a Point in 3D is not a Point in 2D)*

Interfaces > Inheritance

- **Java** introduced interfaces to avoid **C++** multiple inheritance.
- **Go** takes them to their natural conclusion.
- Interfaces specify the messages, without implementation.

```
type Foe interface {
    Update(dt time.Duration)
    DisplayOn(c Canvas)
}

func (c ChuckFoe) Update(dt time.Duration) {
    …
}
```

Back to the essence of OO

- OO should not be primarily about inheritance and code re-use.
- OO is a strategy to design the high level structure of a system.
- Information Hiding.
- Message Passing as a metaphor to focus on conventional interfaces.
- Read article (8 pages): *On the Criteria To Be Used in Decomposing Systems into Modules* by David L. Parnas, 1971.
- If you want to learn OO, learn **Go**.

OO with Go

Hey! Ho! Let’s Go!

OO with Go

- Interfaces everywhere.
- Composition everywhere.
- Duck typing, statically checked by the compiler.
- First-class functions. Not everything is an object.
- No subclasses, no classes, no virtual, static, protected.
- Anything can be a message receptor.

Duck typing

- Only dynamic languages could do this until Go.

```
package duck

type Duck interface {
   Quack()
   Walk()
}

func playWithDuck(duck Duck) { … }

func doSomeStuff() {
   mallard := otherPackage.GetMallard()
   // otherPackage does not “declare” Duck,
   // but Mallard has Quack() and Walk() methods
   playWithDuck(mallard) // This is fine
```

But I miss implementation inheritance!

- If I implemented Newtonian motion as the `Update()` for the top-level class Foe, Bob and Chuck could inherit it. That would be very DRY.

```
type MotionState struct {
   Position Vector3D
   Speed    Vector3D
}

type LawOfMotion func (*MotionState, time.Duration) *MotionState

type BobFoe struct {              func (b *BobFoe) Update(dt) {
   ammo   int                        // update b.ammo …
   mState *MotionState               b.mState = b.move(b.mState, dt)
   move   LawOfMotion             }
}
```

But I miss implementation inheritance!

- Still miss it?

```
// LawOfMotion: Newtonian, Relativistic, Brownian

// bob.MakeFoe(mover LawOfMotion)
// chuck.MakeFoe(move LawOfMotion, sh DefenseMechanism)
// emma.MakeFoe(move LawOfMotion, wp Weapon)

b := bob.MakeFoe(Newtonian)
c := chuck.MakeFoe(Relativistic, HideHeadInSand)
e := emma.MakeFoe(Newtonian, FriggingLaserBeam)
```

- OO purists would have implemented LawOfMotion as a class hierarchy, and used the *Strategy Pattern*. Ugh!

Actually I made up the term "object-oriented", and I can tell you I did not have C++ in mind.

The Computer Revolution hasn't happend yet — 1997 OOPSLA Keynote

Alan Kay, 1997