Object-Oriented Programming

Professor: Suman Saha

Abstract Data Types

PennState

Primitive types: values and operations on values

User-defined types: records, lists, …

Focus on values

ADT: defined by a set of operations on a type

Focus on operation

Stack is a type with `new, pop, push, empty ...`

Internal representation is less relevant

Classifying Operations

PennState

Creators: create new objects of type

Producers: create new objects from old ones

Mutators: change objects, e.g., `list.add(n)`

Observers: take objects of ADT and return objects with different type, e.g., `list.size()`

ADT Examples

PennState

int

Creators: numeric literals `1, 2, 3, ...`

Producers: arithmetic operations `+, -, *, /, ...`

Observers: comparison operators `==, !=, <, >`

Mutators: none (immutable)

ADT Examples

PennState

List

Creators: `ArrayList, LinkedList, …`

Producers: `Collections.unmodifiableList()`

Observers: `size(),get()`

Mutators: `add(),remove(), …`

ADT Examples

PennState

String

Creators: `String(), String(char[])`

Producers: `concat(),substring(),…`

Observers: `length(), charAt(),…`

Mutators: none (immutable)

OOP Terminology

PennState

Class: *a richer version of* ADT

Object (Instance): a variable of a class (a *value* of a *type*)

Field: variable in a class

Method: operation in a class

Object-Oriented Programming

PennState

Key elements:

- Encapsulation
- Subtyping
- Inheritance

Encapsulation (Information hiding)

PennState

- Group data and operations in one place (typically, in one class)
- Hide irrelevant details (using visibility modifiers, such as *public*, *private*, *protected*)

Subtyping

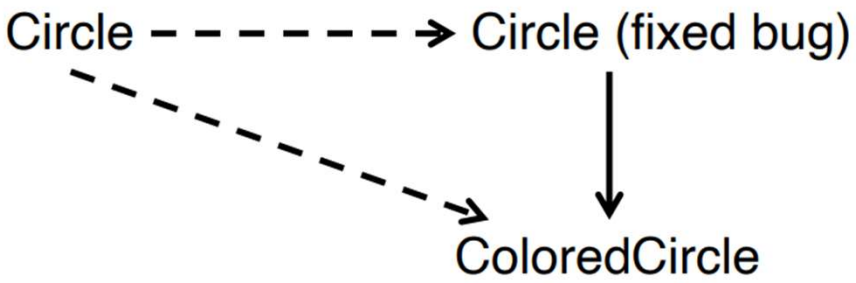
PennState

```
interface Shape {
    public double area();
    public int edges();
}
```

```
class Circle implements Shape {
  double radius;
  public double area() {return 3.14*radius*radius};
  public int edges() {return 1};
}
```

Inheritance

```
class ColoredCircle extends Circle {
  private Color color;
  ...
  Color getColor {return color};
  // methods area, edges are inherited from Circle
}
```



Overriding

PennState

```
class DoubleCircle extends Circle {
  pubic DoubleCircle(double r){super(r);}
  public int edges {return 2};
}
```

Subclass may redefine methods in super class

Object-Oriented Programming

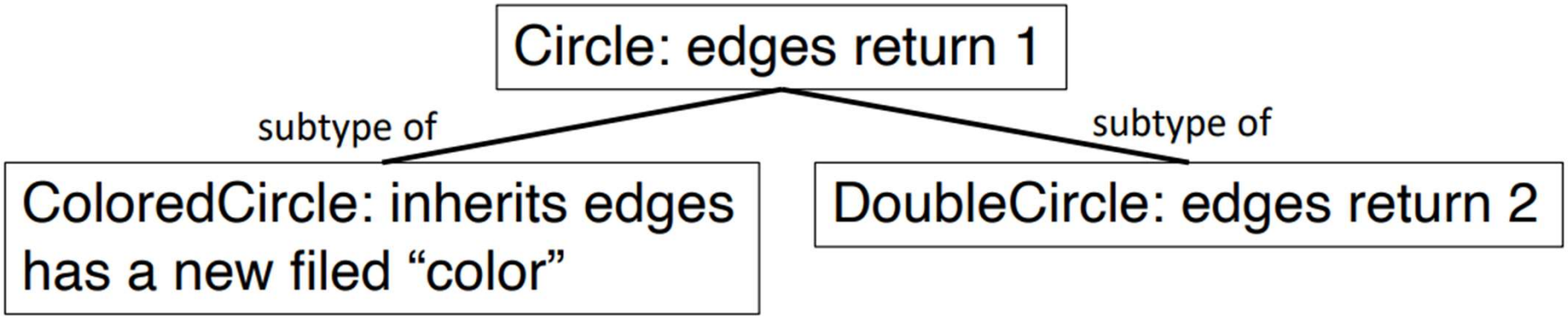
PennState

Key elements:

- Encapsulation
- Subtyping
- Inheritance

How are these features implemented?

Running Example



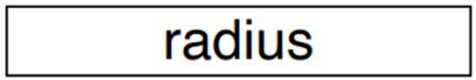
Memory Layout (Fields)

PennState

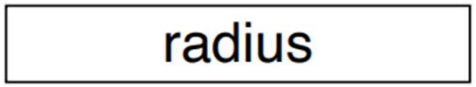
An object has

- Fields (and ones from super class)

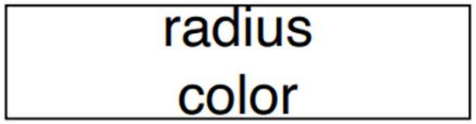
Circle object1:



DoubleCircle object:

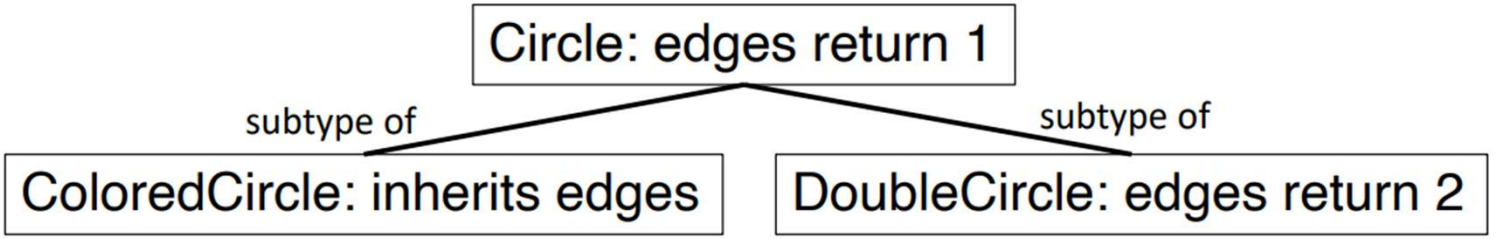


ColoredCircle object:




```
foo (Circle s) {
  s.radius;   // offset?
}
```

Memory Layout (Functions)

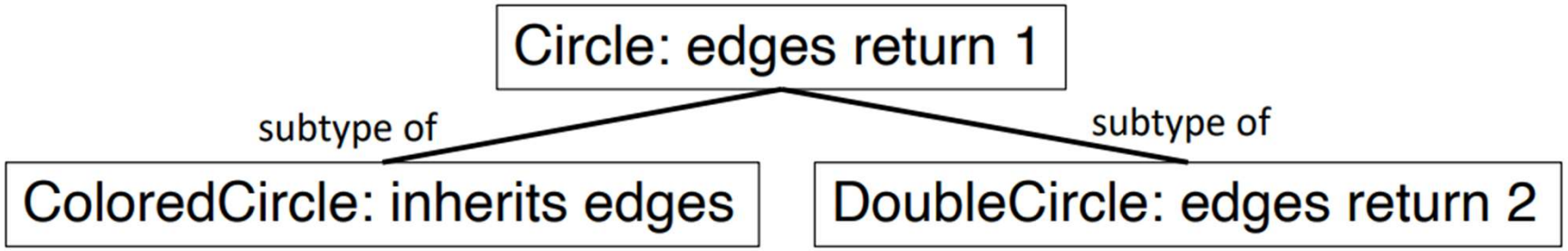



```
foo (Circle s) {
  s.edges();   // which implementation?
}
```

Static dispatch: s.edges() always returns 1

Dynamic dispatch: return value of s.edges() controlled by the type of s

Static Dispatch

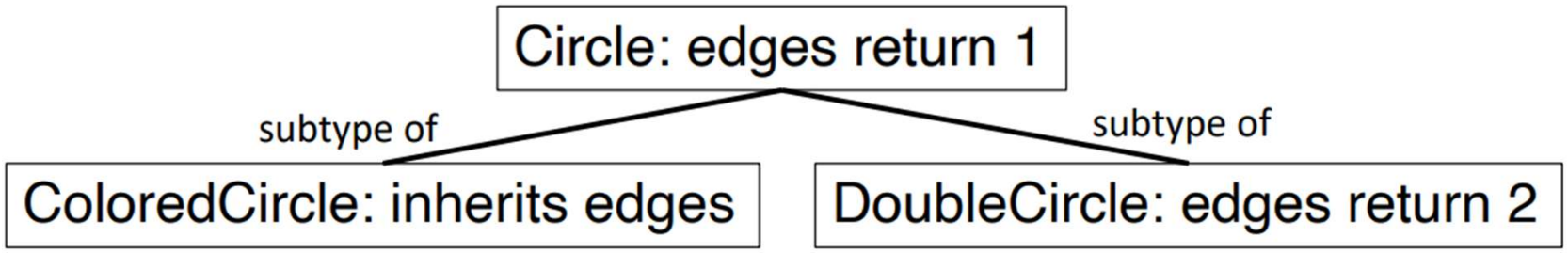



```
foo (Circle s) {
  s.edges();   // which implementation?
}
```

Dispatch to the implementation in class Circle

Hence, s.edges() always returns 1

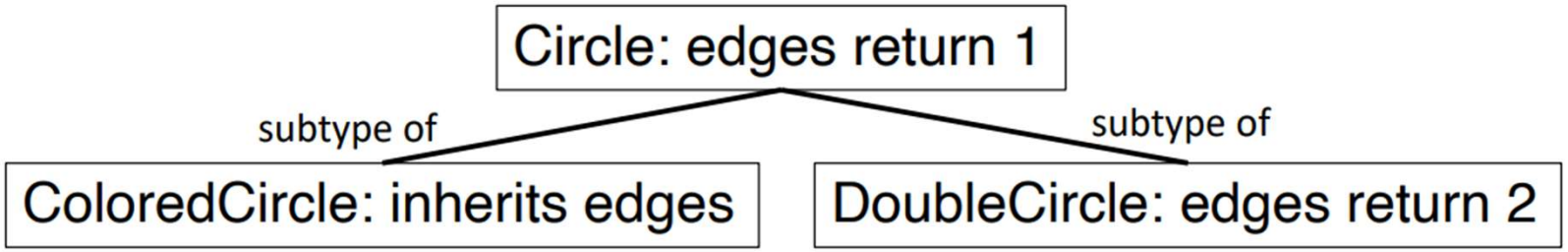
Implementation: Static




```
foo (Circle s) {
  s.edges();    // which implementation?
}
```

The compiler can always tell which implementation at compile time (e.g., the edges method in class Circle)

Dynamic Dispatch

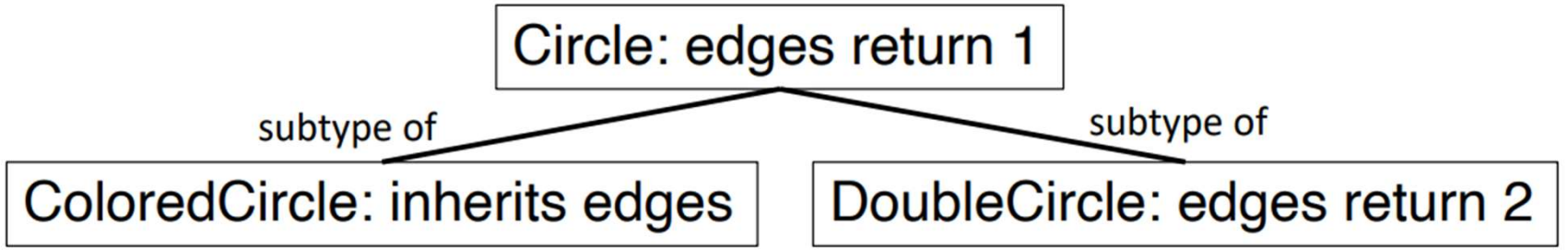



```
foo (Circle s) {
  s.edges();   // which implementation?
}
```

Dispatch to the implementation based on the type of the object s

Hence, s.edges() returns 2 when s is an object of class DoubleCircle

Implementation: Dynamic




```
foo (Circle s) {
  s.edges();    // which implementation?
}
```

The compiler does not know the type of s. How can it dispatch the method call to the correct implementation?

Trivial Memory Layout

PennState

An object has

- Fields (and ones from super class)
- Methods (and ones from super class)

Circle object1:

radius
edges: binary area: binary

Circle object2:

radius
edges: binary area: binary

DoubleCircle object:

radius
edges': binary area: binary

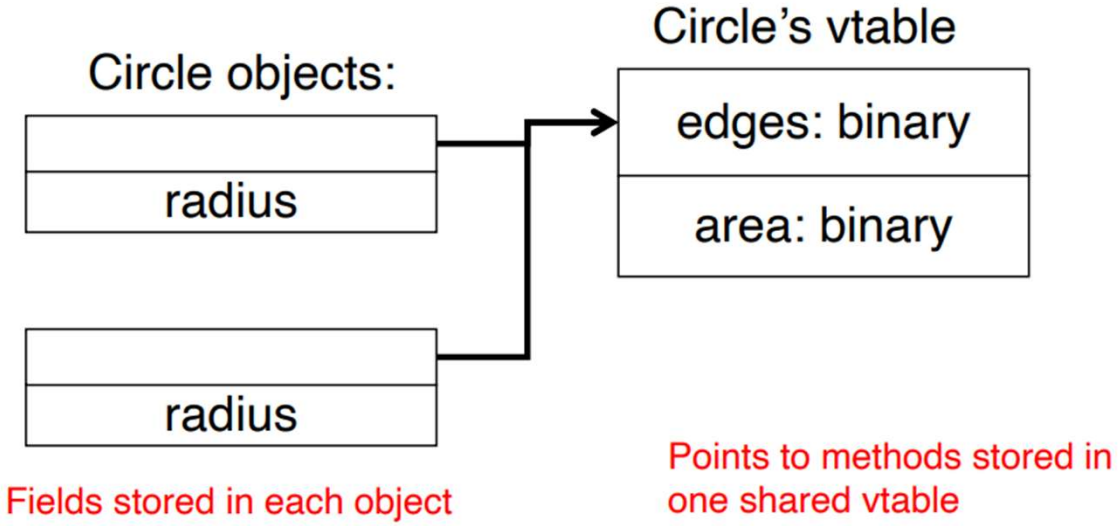
Issues: each object has a copy of impl. code (waste space)
polymorphic functions need to distinguish different layouts of classes (to find method offset)

Virtual Table (vtable)

- Tentative design

A (shared) table containing method binaries

To save memory: one table per ***Class***



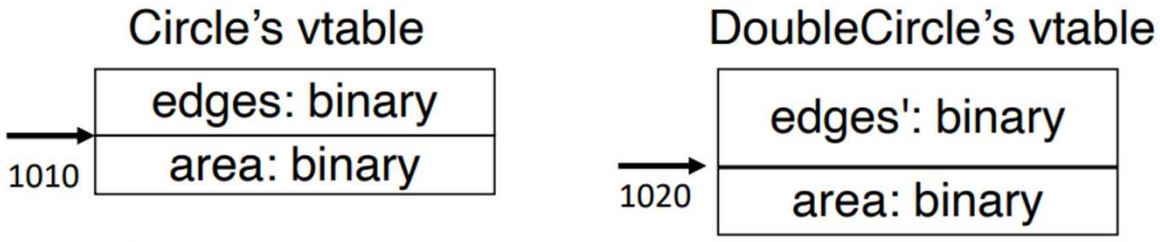
Virtual Table (vtable)

PennState

- Tentative design

A (shared) table containing methods

Override?



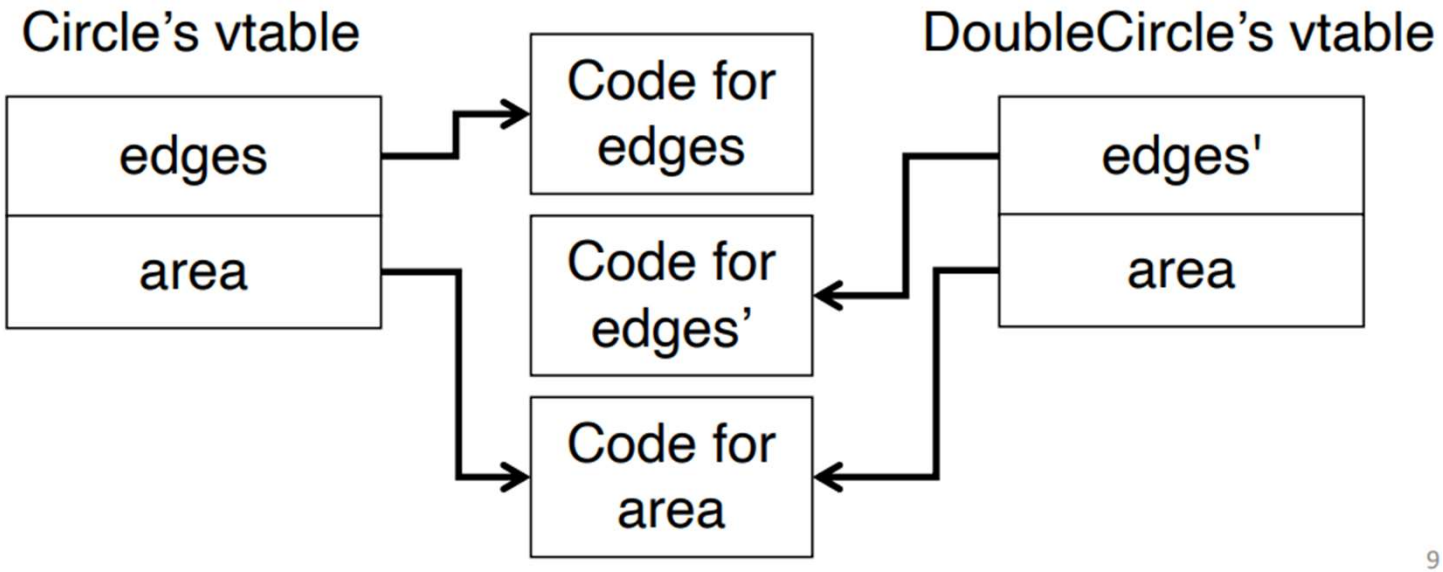

```
foo (Circle s) {
    s.area(); // code has different offsets
}
```

Issue: foo is compiled to different binaries with different offsets for different types of s

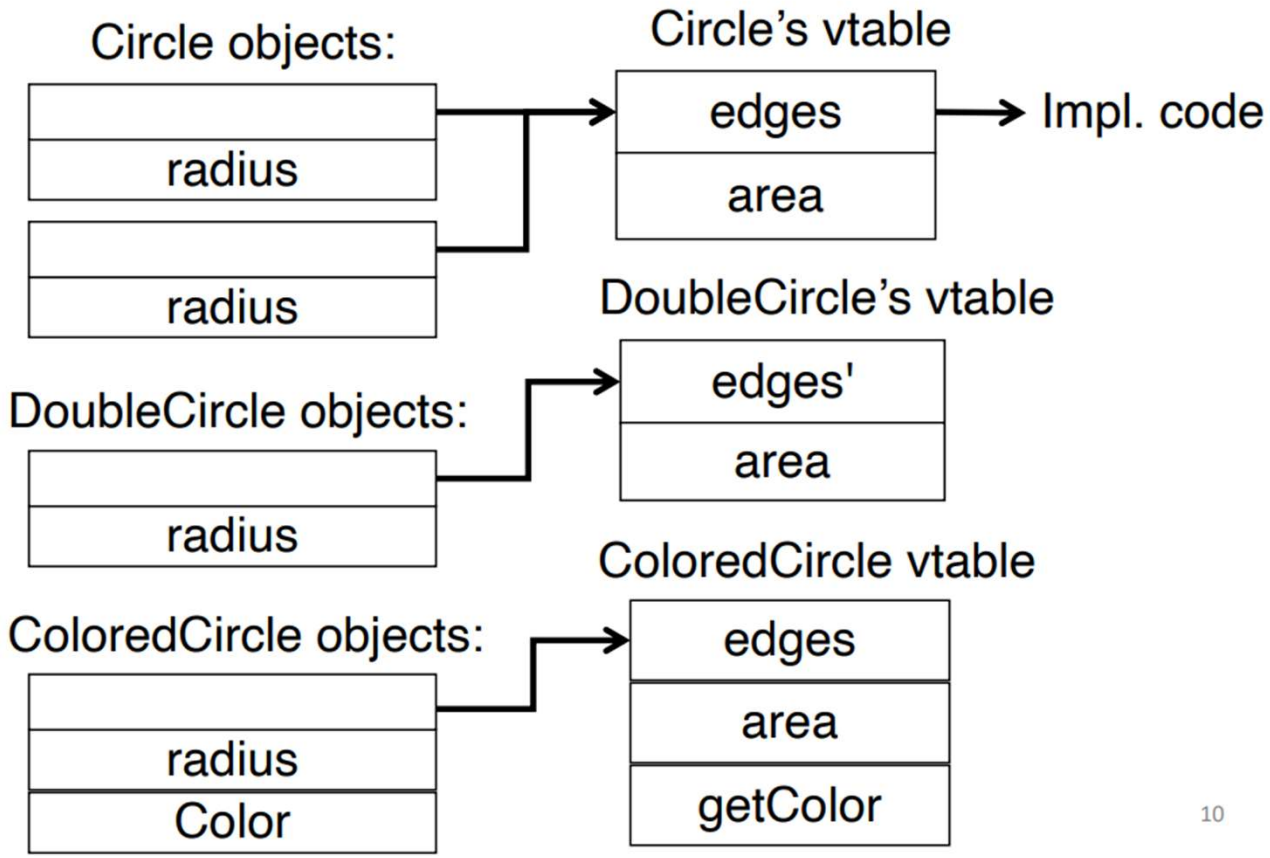
Virtual Table (vtable)

PennState

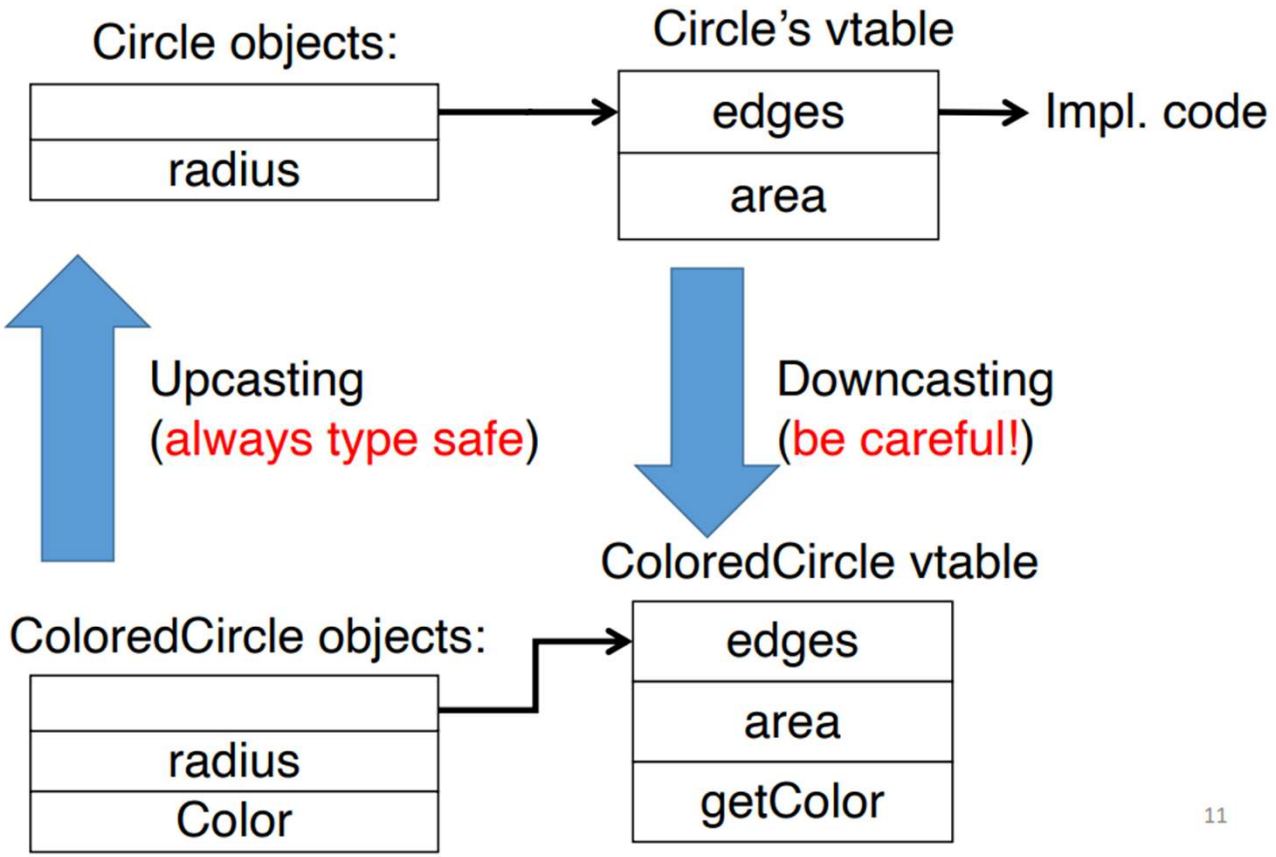
A (shared) table containing pointers to methods



Virtual Table (vtable)



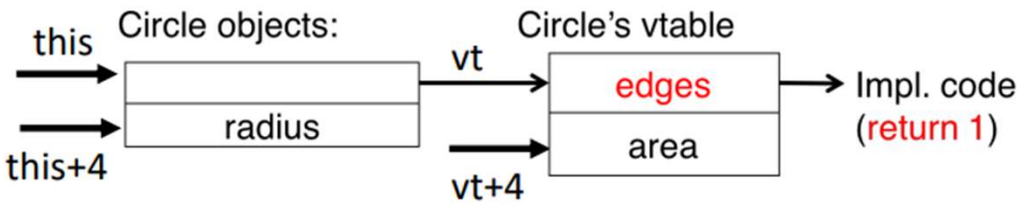
Downcasting/Upcasting



Member Lookup: Case 1

```
foo (Circle s) {
    s.radius;
    s.area();
    s.edges();
}
```

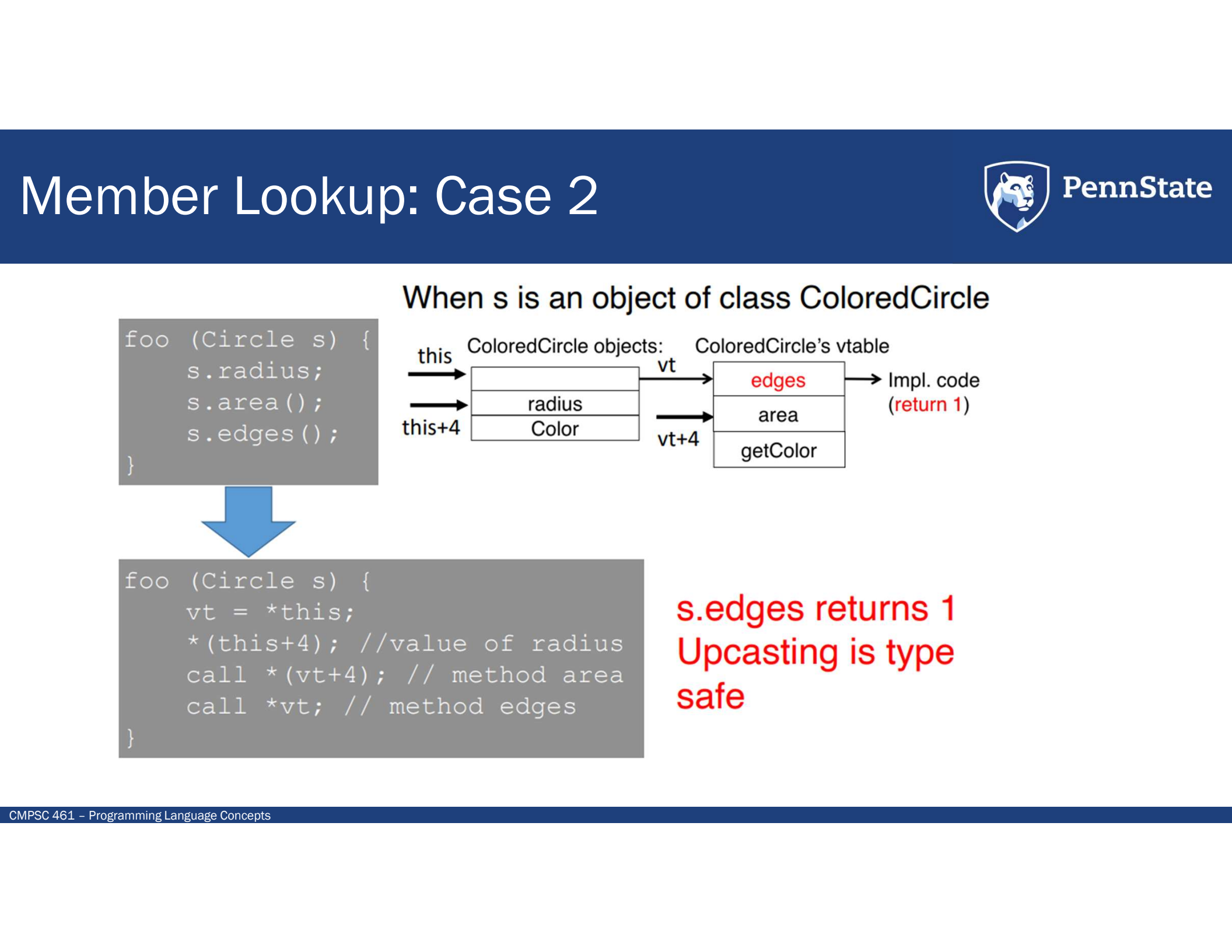
When s is an object of class Circle



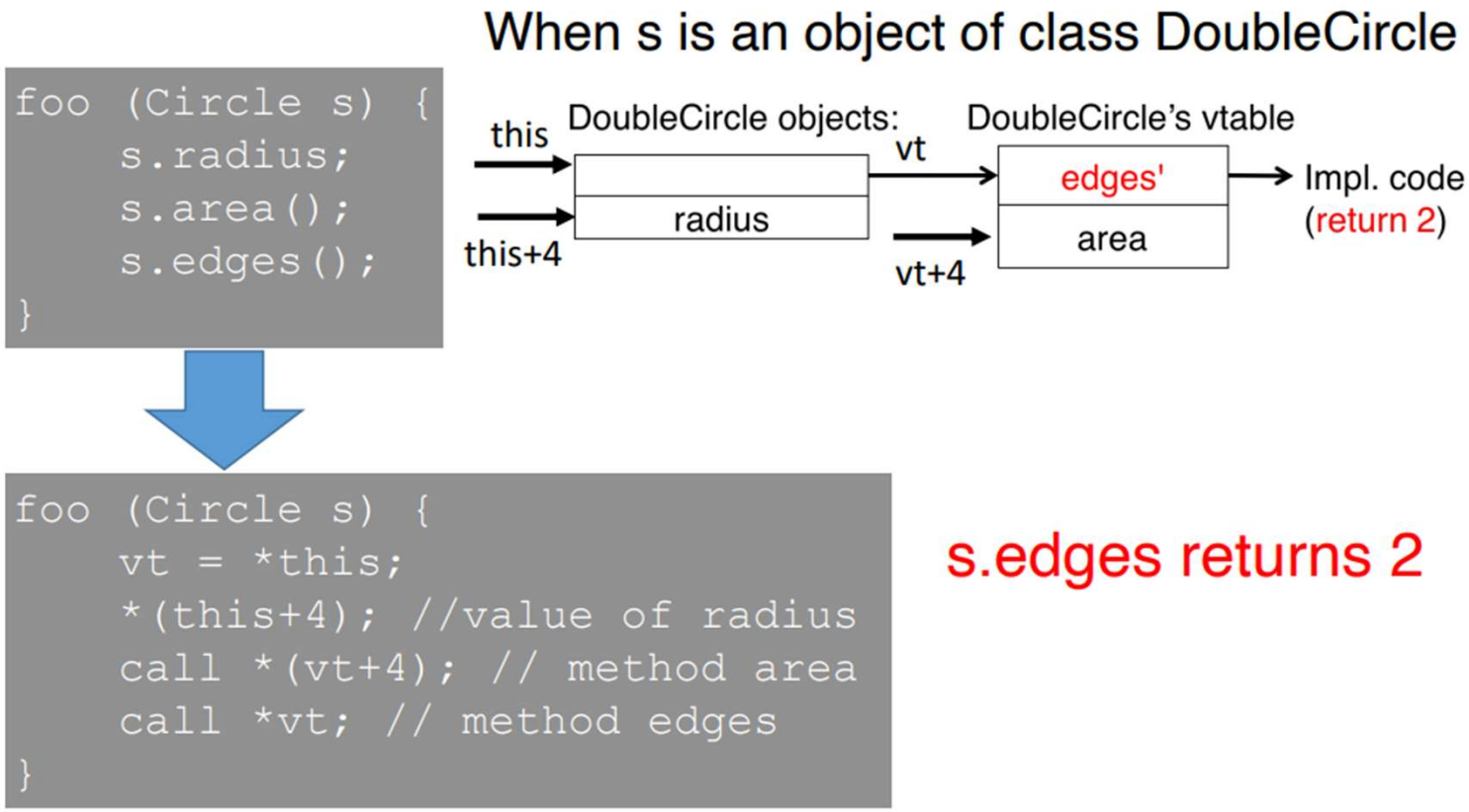

```
foo (Circle s) {
    vt = *this;
    *(this+4); //value of radius
    call *(vt+4); // method area
    call *vt; // method edges
}
```

s.edges returns 1

Member Lookup: Case 2



Member Lookup: Case 3



Dynamic Dispatch with VTables

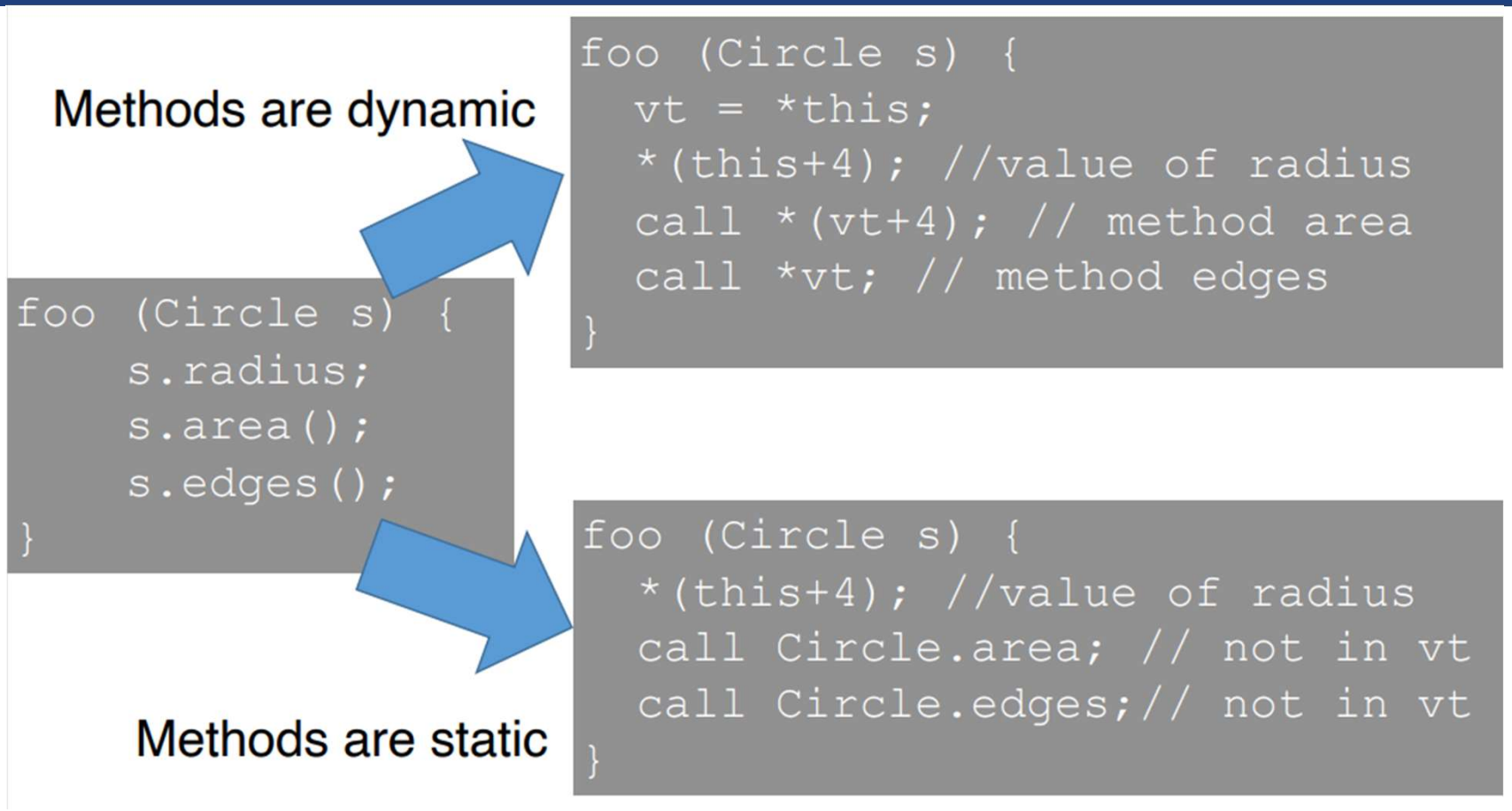
PennState

```
foo (Circle s) {
    s.radius;
    s.area();
    s.edges();
}
```

```
foo (Circle s) {
    vt = *this;
    *(this+4); //value of radius
    call *(vt+4); // method area
    call *vt; // method edges
}
```

One implementation for all subtypes!

Static vs. Dynamic Dispatching



Cost of Dynamic Dispatch

PennState

Dynamic dispatch has costs, but is better for extensibility

In C++: static by default, except the `virtual` methods

In Java: dynamic by default, expect the ones that cannot be overridden (e.g., `final` and `static` methods)

In Python: all methods use dynamic dispatch