Object-Oriented Programming

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

Object-Oriented Programming

Key elements:

- Encapsulation
- Subtyping
- Inheritance

Encapsulation (Information Hiding)

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

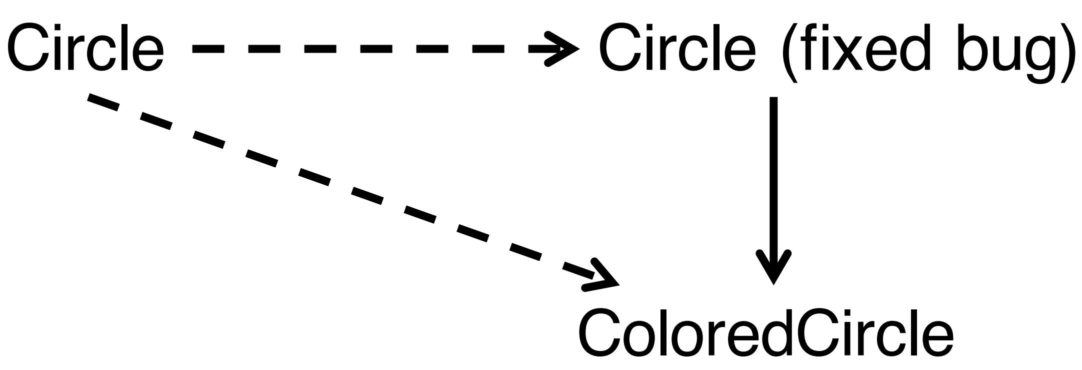
Subtyping

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

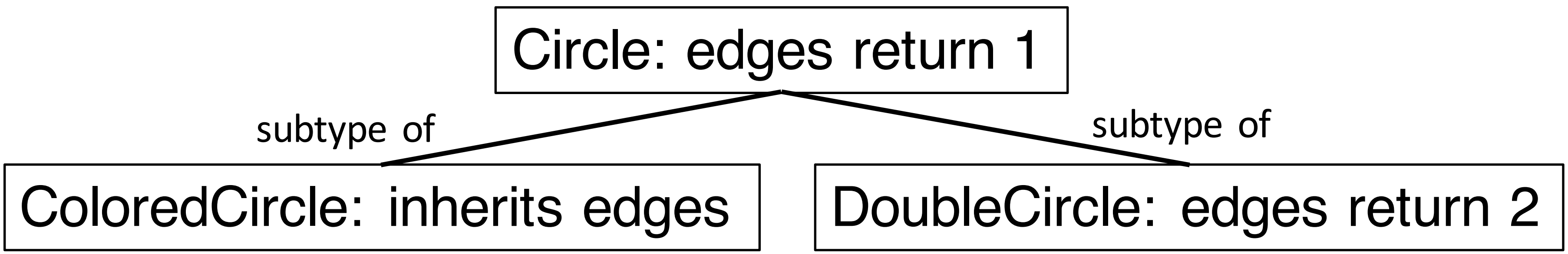
```
class Circle implements Shape {
  double radius;
  public double area() {return 3.14*radius*radius};
  public int edges() {return 1};
}
```

Inheritance

```
class ColoredCircle extends Circle {
  private Color color;
  pubic ColoredCircle(int r, Color c)
        {super(r);color=c;}
  Color getColor {return color};
}
```

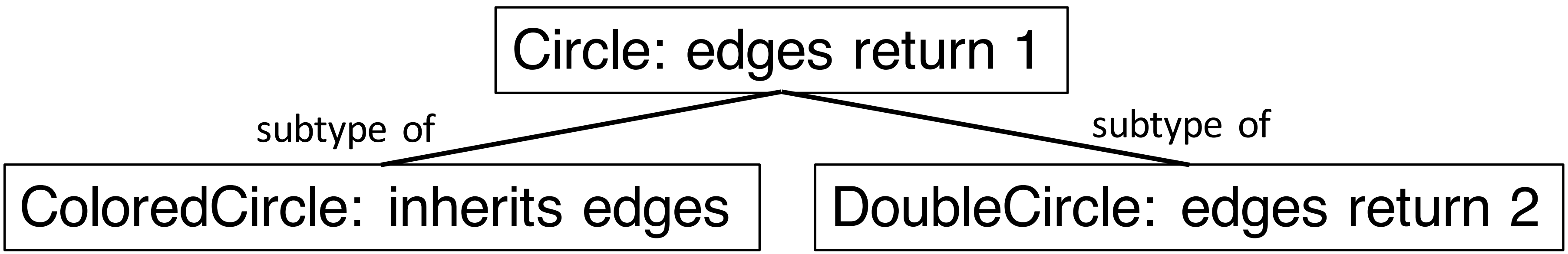


Method Binding




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

Static Dispatch

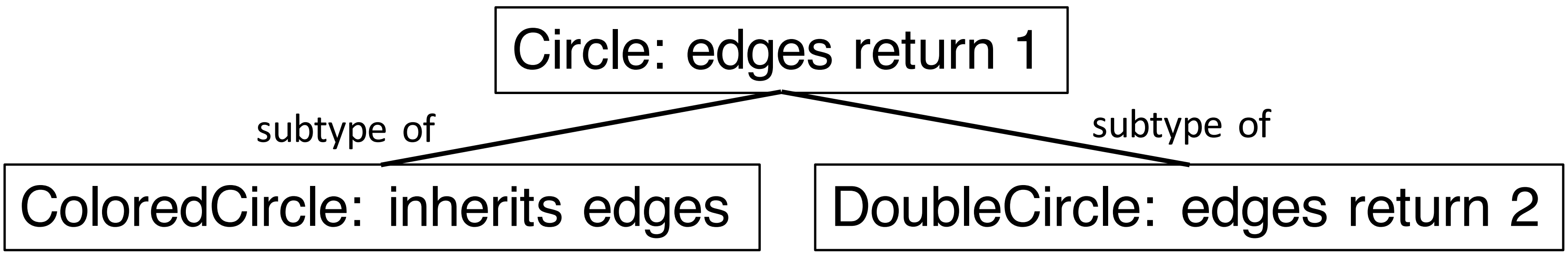



```
foo (Circle s) {
  s.edges();    // which implementation?
}
```

Dispatch to the implementation in class Circle

Hence, s.edges() always returns 1

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

Static vs Dynamic Dispatch

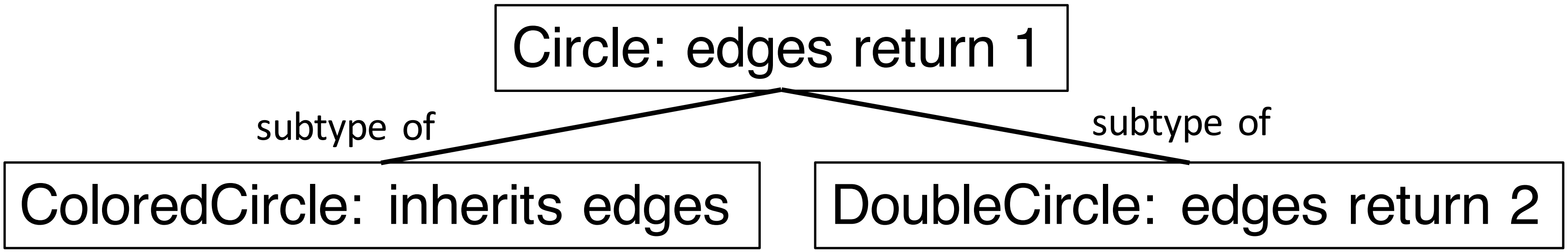
```
foo (Circle s) {
  s.edges();    // which implementation?
}
```

Static dispatch: s.edges() always returns 1

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

Static dispatch is easier to implement and more efficient
Dynamic dispatch less efficient, but provides better extensibility (central to object-oriented programming)

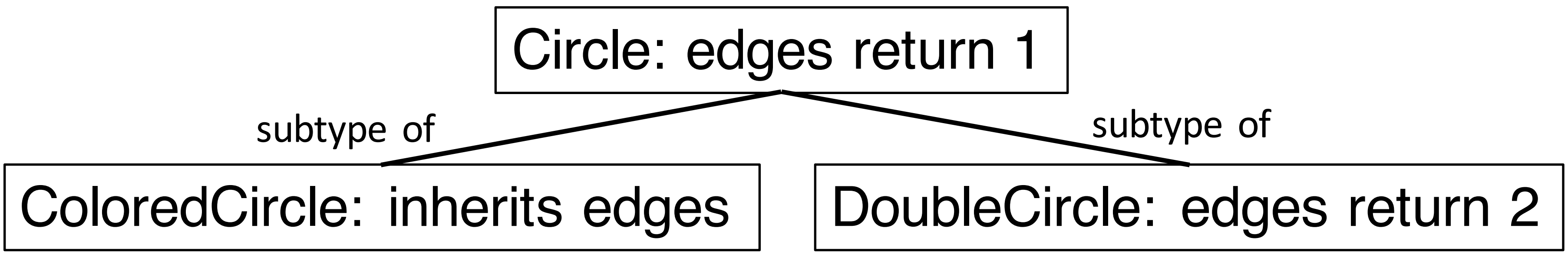
Implementation: Static




```
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)

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

An object has

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

Circle object:

radius
edges: binary area: binary

ColoredCircle object:

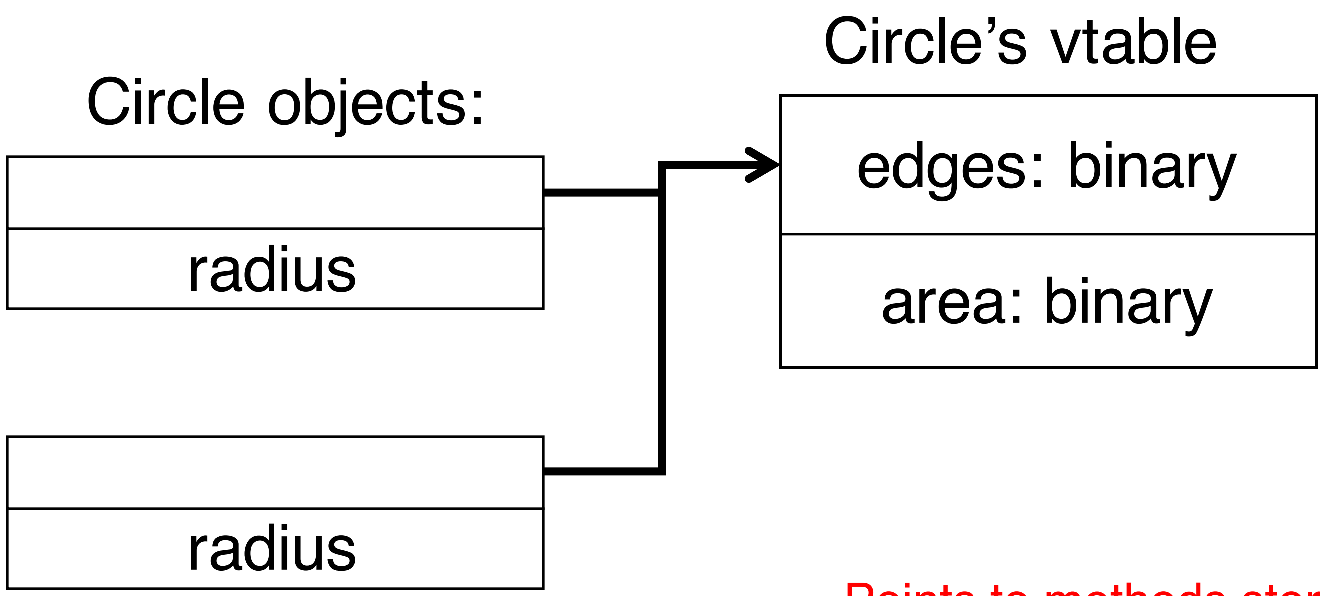
radius color
edges: binary area: binary getColor: binary

Limitations: 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***



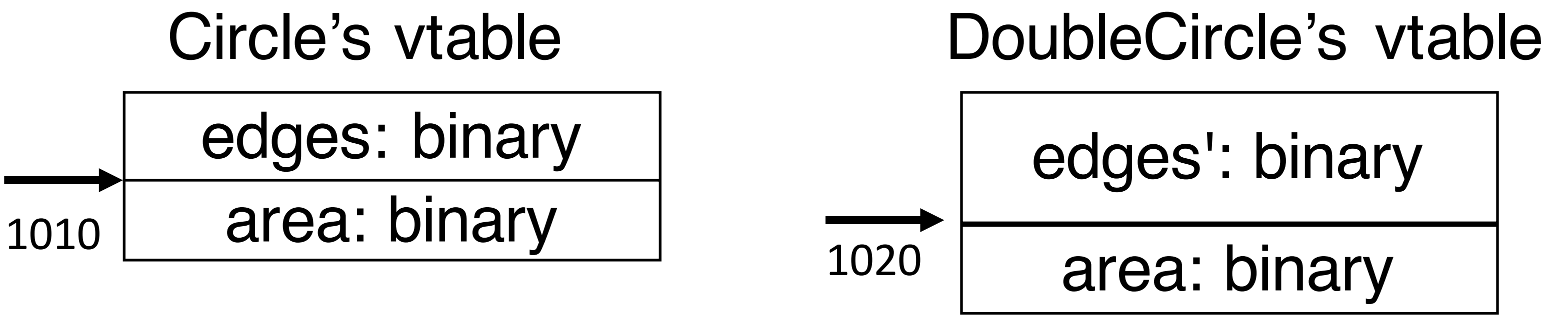
Fields stored in each object

Points to methods stored in one shared vtable

Virtual Table (vtable)
- Tentative design

A (shared) table containing methods

Overloading?

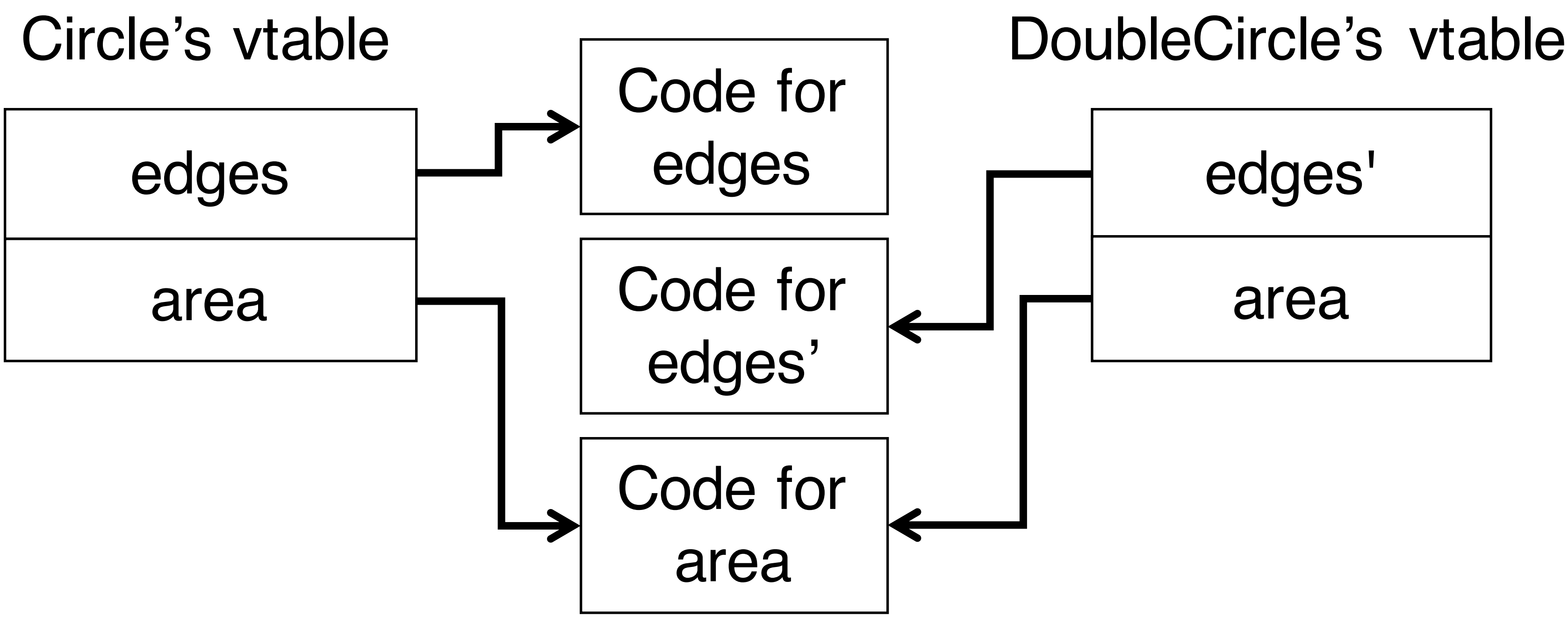



```
foo (Circle s) {
    s.area(); // code has different offsets
}
```

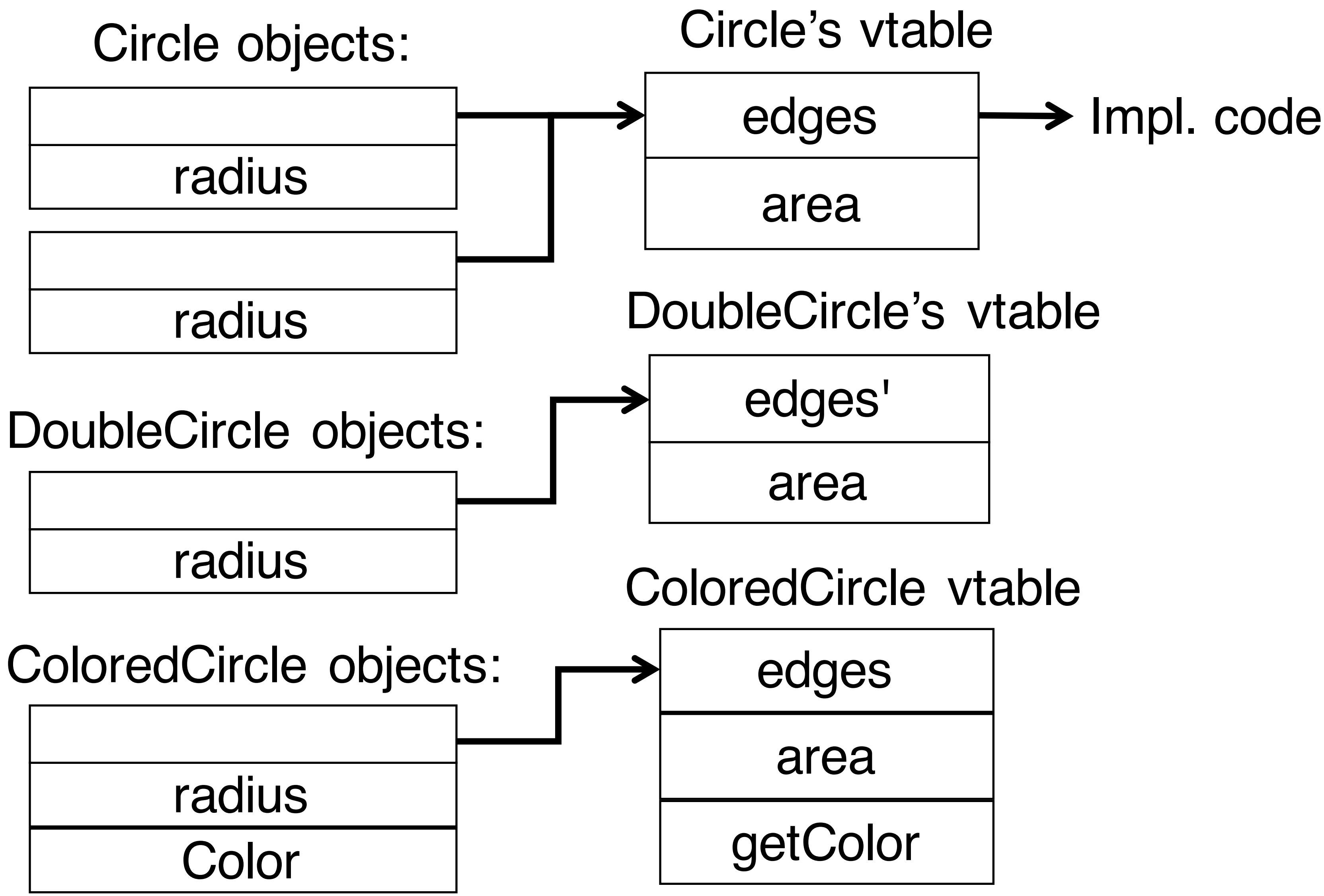
Limitation: foo is compiled to different binaries with different offsets for different types of s

Virtual Table (vtable)

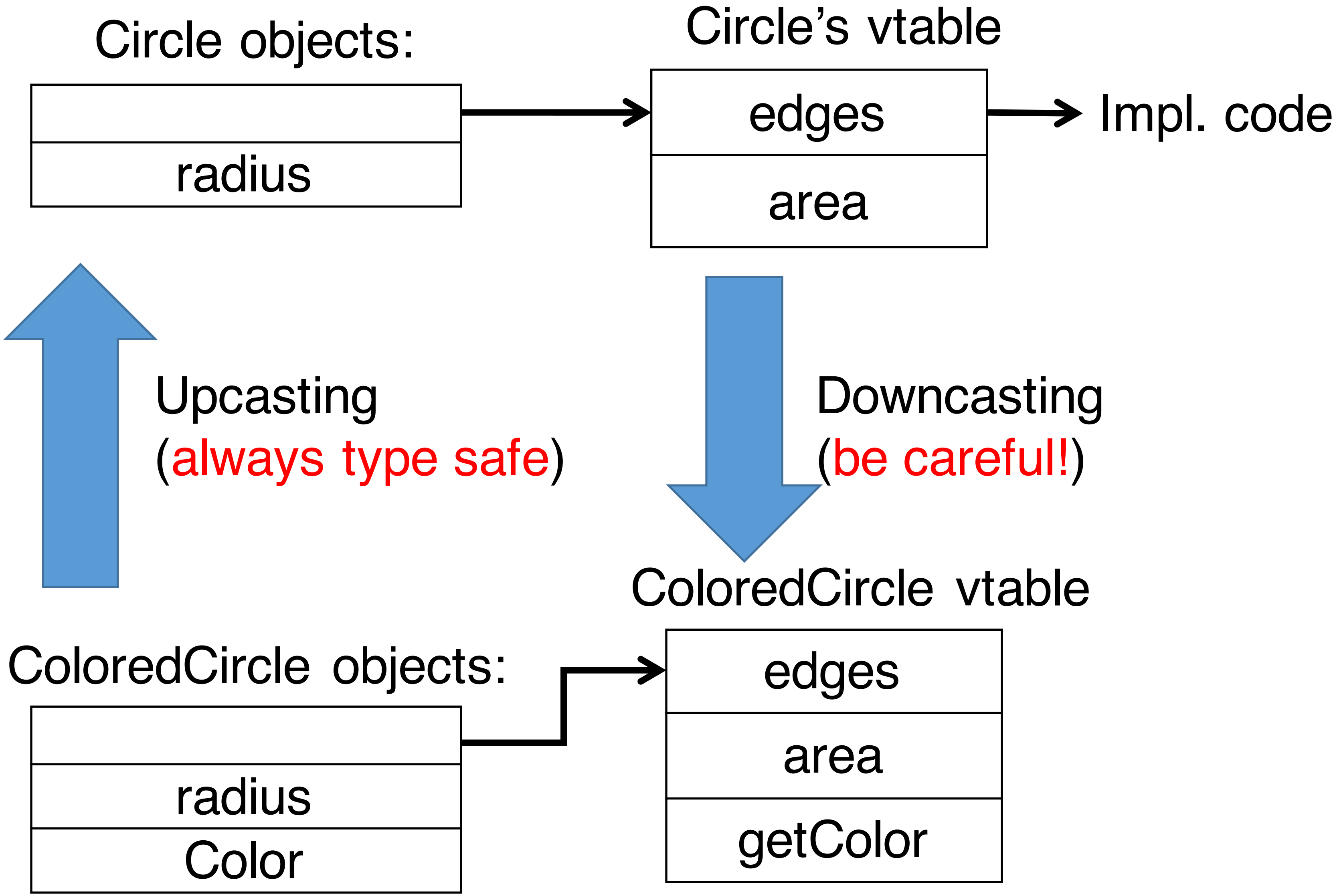
A (share) table containing pointers to methods



Virtual Table (vtable)



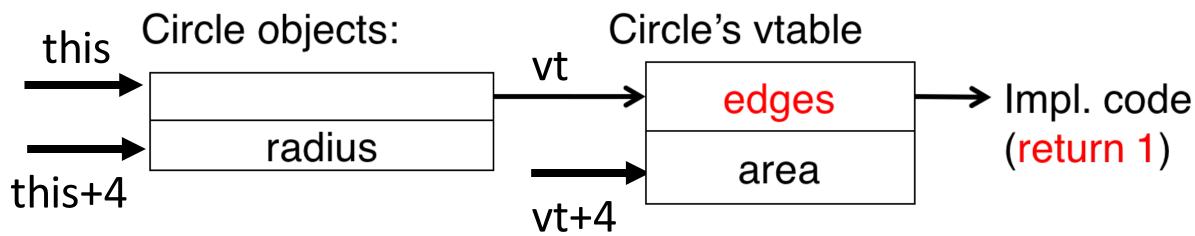
Downcasting/Upcasting

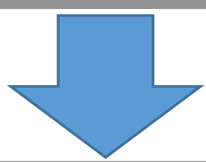


Member Lookup: Case 1

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

When s is an object of class Circle





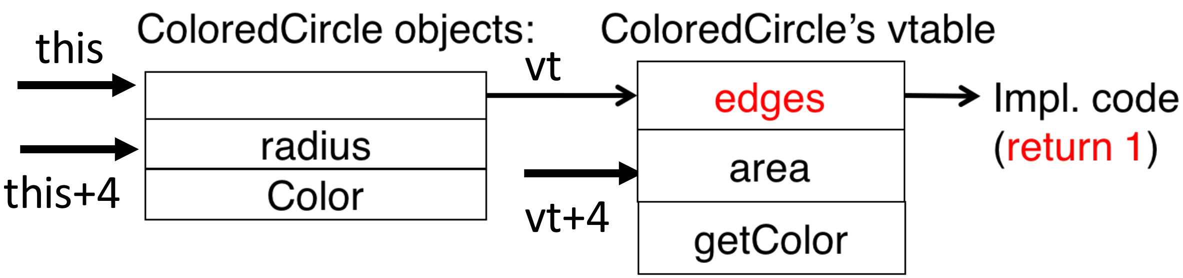
```
foo (Circle s) {
    vt = *this;
    *(this+4); //value of radius
    call *(vt+4); // method area
    call *vt; // method edges
}
```

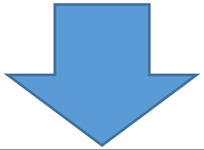
s.edges returns 1

Member Lookup: Case 2

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

When s is an object of class ColoredCircle





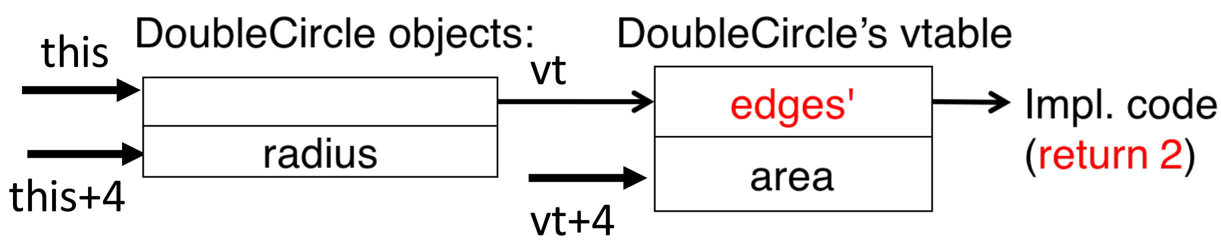
```
foo (Circle s) {
    vt = *this;
    *(this+4); //value of radius
    call *(vt+4); // method area
    call *vt; // method edges
}
```

s.edges returns 1
Upcasting is type safe

Member Lookup: Case 3

When s is an object of class DoubleCircle

```
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
}
```

s.edges returns 2

Dynamic Dispatch with VTables

```
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

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

Methods are dynamic

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

Methods are static

```
foo (Circle s) {
  *(this+4); //value of radius
  call Circle.area; // not in vt
  call Circle.edges;// not in vt
}
```

Cost of Dynamic Dispatch

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

Object-Oriented Programming

Key elements:

- Encapsulation
- Subtyping
- Inheritance