




# **the joys of polymorphism**

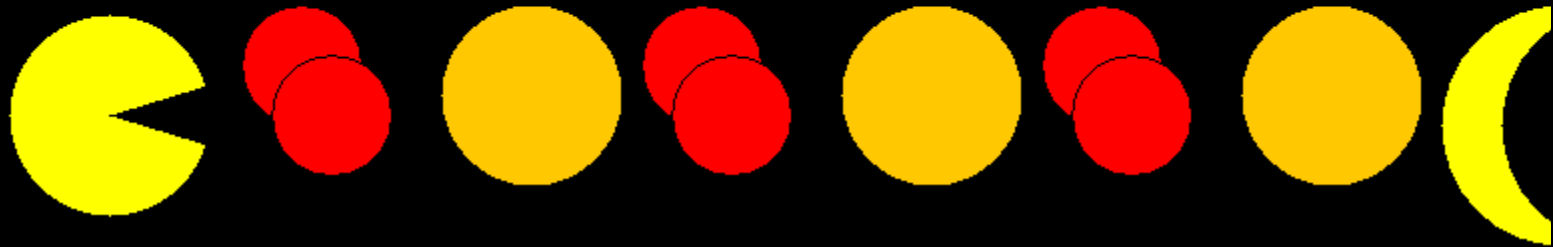
# polymorphism defined

**polymorphism** refers to the ability of an object to exhibit behaviors associated with different types. Java makes polymorphism possible through inheritance and overriding methods.

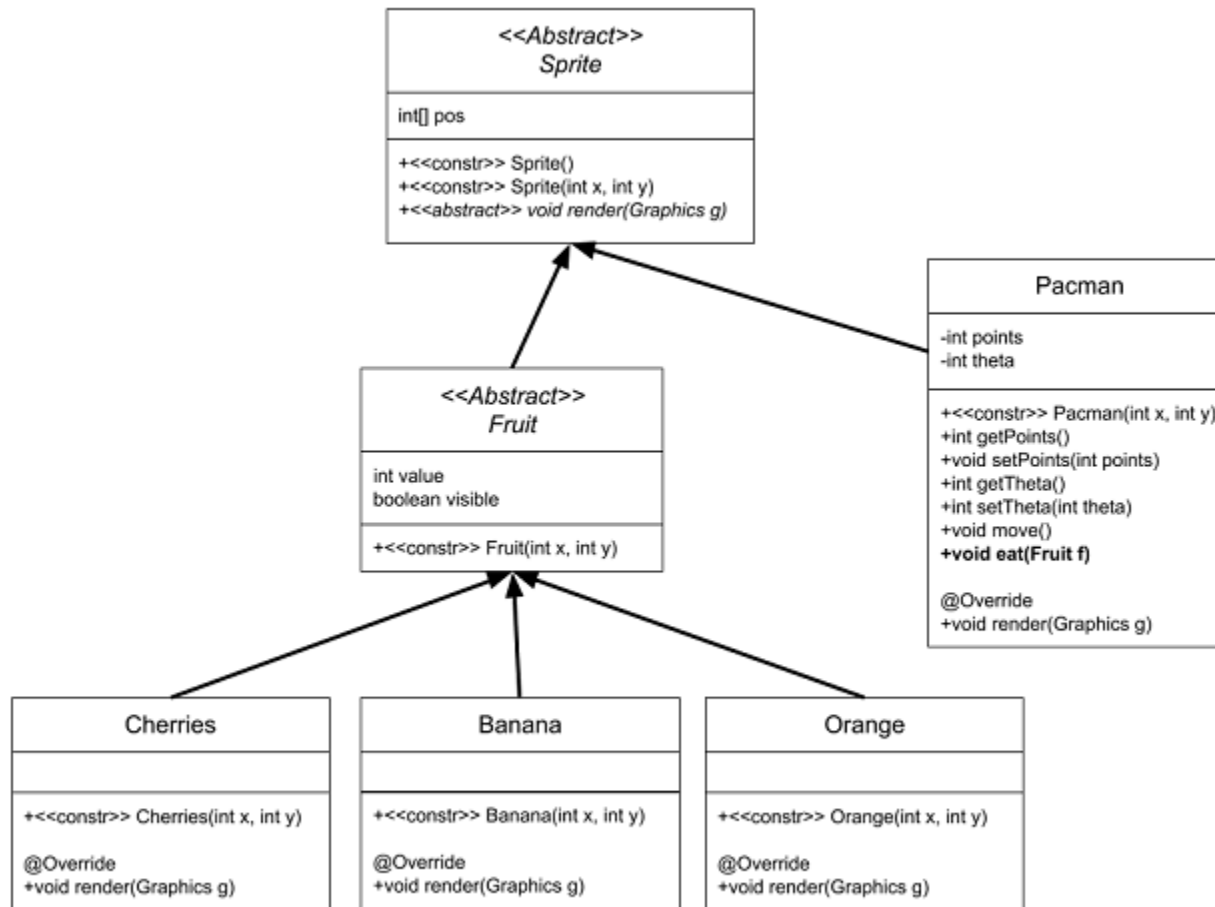
For example, all of the different types of "fruit" below were extended from a common `Fruit` class. They each have a `render()` method, with completely different instructions on how they should be drawn.

 Polymorphic Man of Mystery

Points: 0



# UML diagram for PacMan example



# using polymorphism

Typically when instantiating an object, the reference type will match the type of object being constructed. For example:

```
1 Orange f = new Orange(100, 250);
```

However, some interesting things happen when we allow the reference type to be a superclass of the object we are constructing.

```
1 Fruit f = new Orange(100, 250);
```

If *S* is a subclass of *T*, then assigning an object of type *S* to a reference of type *T* facilitates polymorphism.

We are allowed to do this because an *Orange* object *is a* fruit. This allows us the flexibility to assign the same reference *f* to a completely different type of fruit without breaking the program.

```
1 Fruit f = new Orange(100, 250);  
2 f.render();  
3 f = new Banana(200, 250);  
4 f.render();
```

# Usage 1 - Method Return Types

## Random Fruit Demo

```
1 public Fruit getRandomFruit(){  
2     // will randomly return a new Banana, Orange, or Cherry object  
3 }
```

# Usage 2 - Arrays

## Arrays with mixed fruit types

```
1 Fruit[] fruit = new Fruit[8];  
2 for(int i = 0; i < fruit.length(); i++){  
3     fruit[i] = getRandomFruit();  
4 }
```

# Usage 3 - Formal Parameters

## PolyMan Demo

```
1 public void eat(Fruit f){  
2     // determine if PolyMan is over the fruit,  
3     // update points and make fruit disappear.  
4 }
```

# How this works

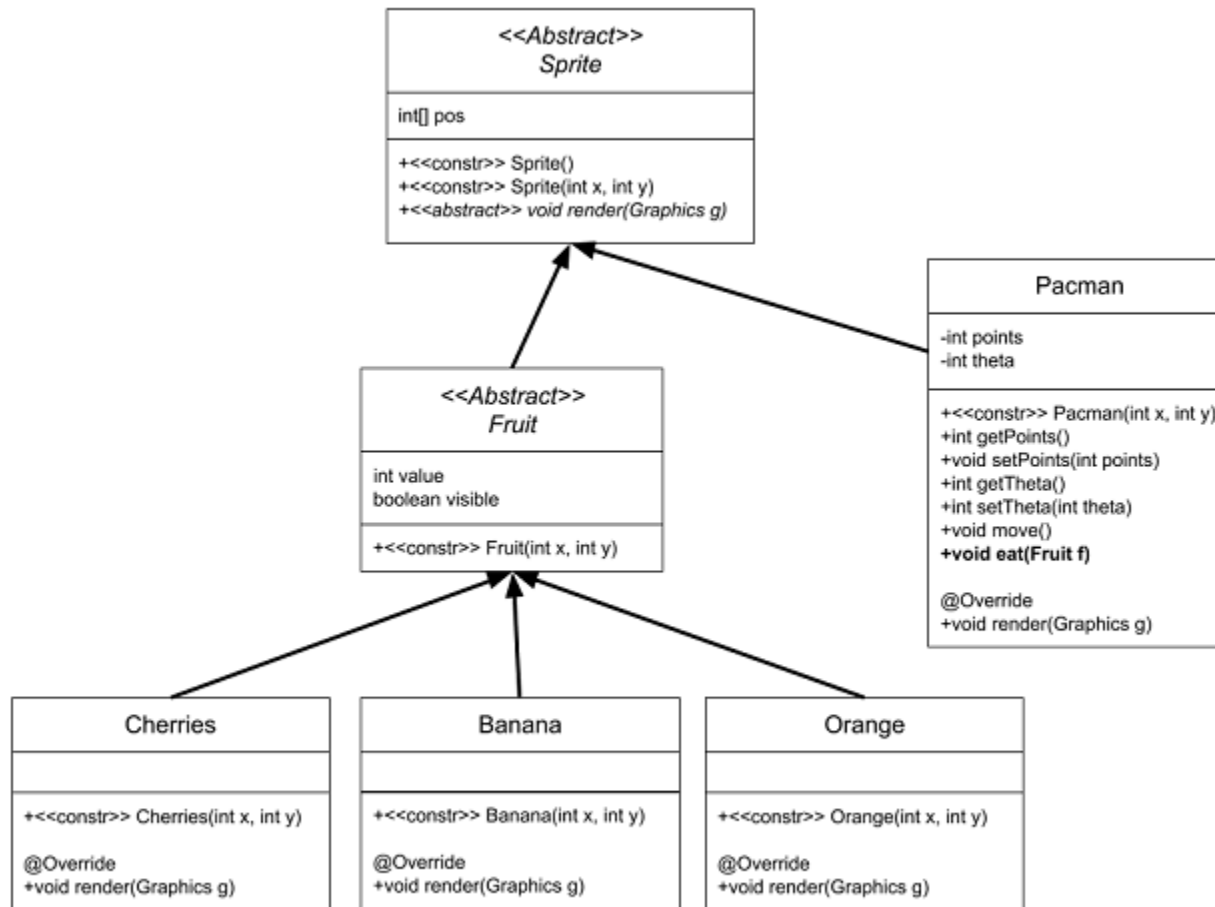
## Compile Time vs. Runtime

**Compiler:** Only checks to see if the method called on a reference has been provided by the reference type class or a superclass. For example, when we declare `Fruit f = new Orange();` and call `f.render()` it only looks to see if `render()` is defined in the `Fruit` class (it is not) and then checks *up* the hierarchy in the `Sprite` class (it is). The compiler doesn't bind the actual overridden method details from the `Orange` class to the object.

**JVM:** The JVM executes the overridden method at runtime. This is called *dynamic binding* and it is essential to making polymorphism work in Java.



# lab 031



- Design your own subclass of `Fruit` to work with this program.