

# CS2110: SW Development Methods

---

## Inheritance in OO and in Java Part 1: Introduction

### Readings:

- A few pages in Ch. 2 of MSD text introduce this
- Section 3.3 of MSD text for Java details

# First, A Reminder: Classes, Type

- Object references have a type
  - What the rules are for an object
    - Its state, behavior
    - What methods can be called on it
- The class that's used to instantiate an object is one form of object-type
  - Others? (Details coming.)
    - Its **superclass** for its class
    - An **interface** that its class “supports”

# Motivation(s) for Inheritance

- Often we need a new class that is almost exactly like an existing class
  - A principle to code by:  
*DRY: Don't repeat yourself!*  
*Write once!*
  - Sometimes we just need to add something new,  
or change the behavior of a method

# Motivations (2)

- Sometimes our classes model things that in the real world have a “type of” or an “is a” relationship
  - Good software reflects the real-world
- Sometimes at run-time we have a collection of objects of unknown (but related types) but they have a common interface
  - Set of graphics objects in a window and draw()
  - Design goal: “hide” details of design decisions behind this interface

# OO Programming and Inheritance

- Inheritance allows us to:
  - Create a new class that “extends” an existing class
    - Avoid re-writing code for the common parts of these classes
  - Have more flexibility at run-time in calling operations on objects that might be of different types
    - Recall we use reference variables to “point to” objects

# Inheritance Helps Out

- So, inheritance can support the three things on the last slide:
  1. Code reuse
  2. SW that better matches the problem domain
  3. Flexible Design

# Terminology

- A new class is defined from an existing class
  - New class is called the subclass in Java
    - AKA a derived class, descendant class, etc.
  - Existing class is called the superclass in Java
    - AKA a base-class, parent class
- We say:
  - The subclass inherits from the super class
  - The subclass extends the superclass.

# Terminology (cont' d)

- Fields and methods in the superclass are also in the subclass class.
  - They're inherited.
- For methods, the subclass *can possibly* define a new implementation that replaces the inherited definition
  - The subclass overrides the inherited behavior
  - The method is over-ridden
  - Methods **not** over-ridden are simply inherited

# Example #1

- Existing class: Person
  - Fields: name, homeAddress
  - Methods: constructor, getters, setters  
getMailingAddress()
- We want a new class: Employee
  - Additional fields: employeeId, workAddress
  - Additional methods: getters, setters for new fields
  - Altered method: new version of  
getMailingAddress()

# How to in Java for Example #1

- Create new class Employee
  - public class Employee extends Person {...*
- In Employee, add only code for what's changed or new
  - New fields: employeeId, workAddress
  - New methods: getters and setters
  - Over-ridden method: getMailingAddress()
- **See code on website with these slides**
- **See example in textbook: Section 3.3**

# More How-To for Example #1

- We want Employee's constructor to use Person's constructor on the current object
  - Not to create a new object -- on itself!
  - Java keyword used for superclass' constructor:  
`super( parameters );`
  - Must be first line in subclass' constructor

# Another Use of super

- Can we access a method in the superclass?
  - Yes! Normally just use the name.
  - What's defined in the superclass is part of the subclass too
- But what if it's over-ridden
  - Often we over-ride to replace behavior and don't care about old behavior
  - But *super.foo(params)* calls *foo()* defined in the superclass
    - Instead of *foo()* defined in the current class (the subclass)

**T or F? Code in a subclass' methods can access always access each field defined in the superclass and subclass.**

1. True
2. False

# Private vs. Protected

- Can subclass methods access private things defined in superclass?
  - No!
- If desired, superclass can allow this by declaring fields and methods

*protected*

# **MySub extends TheSuper. Which can be true?**

1. TheSuper has method foo(), and MySub does not
2. MySub can have a new version of foo().
3. Both 1 and 2.
4. Neither 1 and 2.

# A Timely Example

- Imagine many types of clocks
  - Digital, analog
  - Watch, alarm, grandfather, cuckoo
- A superclass called Clock
  - Various methods, including advance()

# Clock Example (cont' d)

- Imagine variations on clocks that change or extend the basic behavior
  - 24-hour vs. 12-hour: override display()
  - Cuckoo clock: add cuckoo functions
  - Alarm clock: add alarm features
- Wouldn't it be nice to:
  - Share common code
  - Handle collections of clocks (see next slide)

# Clock Activities

- It's time to change the clocks by one hour!
  - Say allMyClocks is a collection of all my clocks (of various types)

```
// The following is Java-like pseudo-code  
for each clock c in allMyClocks {  
    c.advance(60); // move ahead one hour  
    c.display(); // make them display the time  
}
```

## Clock Activities (2)

- Could allMyClocks store references to any of the possible clocks? (Yes.)
  - Could the right version of display() be called for each type of clock? (Yes.)
- How? Use references to the superclass, Clock
  - Any subclass object “is a” instance of a Clock object

# “Is-a” and Inheritance

- We say: any subclass object (AlarmClock etc.) “is a” instance of a Clock object
- What’s this mean exactly?
  - *Substitutability principle*: Wherever we see a reference to a Clock object in our code, we can legally replace that with a reference to any subclass object
  - Implies that we can “use” the subclass object in any way that’s legal for the superclass

# Class Object

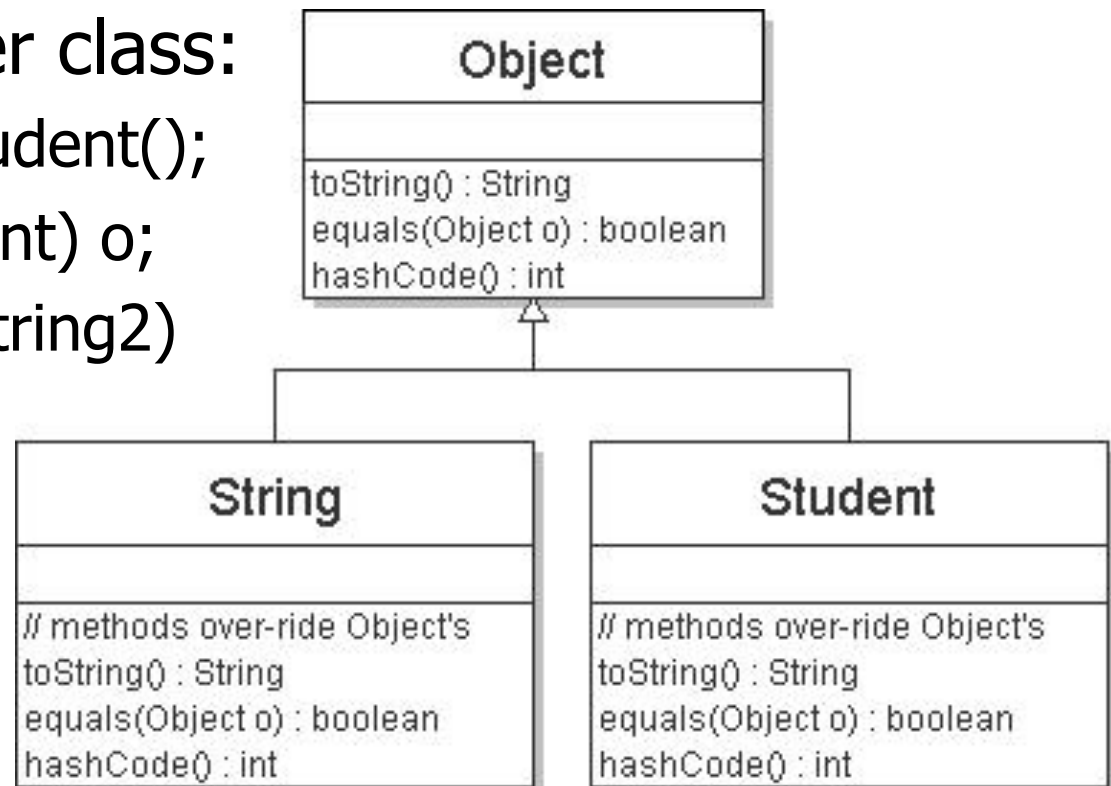
- See Section 3.3.5
- All classes in Java are subclasses of Object
- Object contains some useful fields
  - But more importantly, methods are inherited
  - The implementation matters less than the interface!
    - All objects of any class in Java have certain methods we can count on
    - Subclasses can over-ride these

# Some Methods in Object

- public boolean equals(**Object** obj)
  - Collections and other utility classes rely on this, so they work with your classes
  - **Must** match this interface to over-ride
    - Otherwise, you've introduced a new but different method
- public String toString()
  - We over-ride this for output, string concatenation
- public int hashCode()
  - We'll need this later for Map collections

# Object and “Is-a”

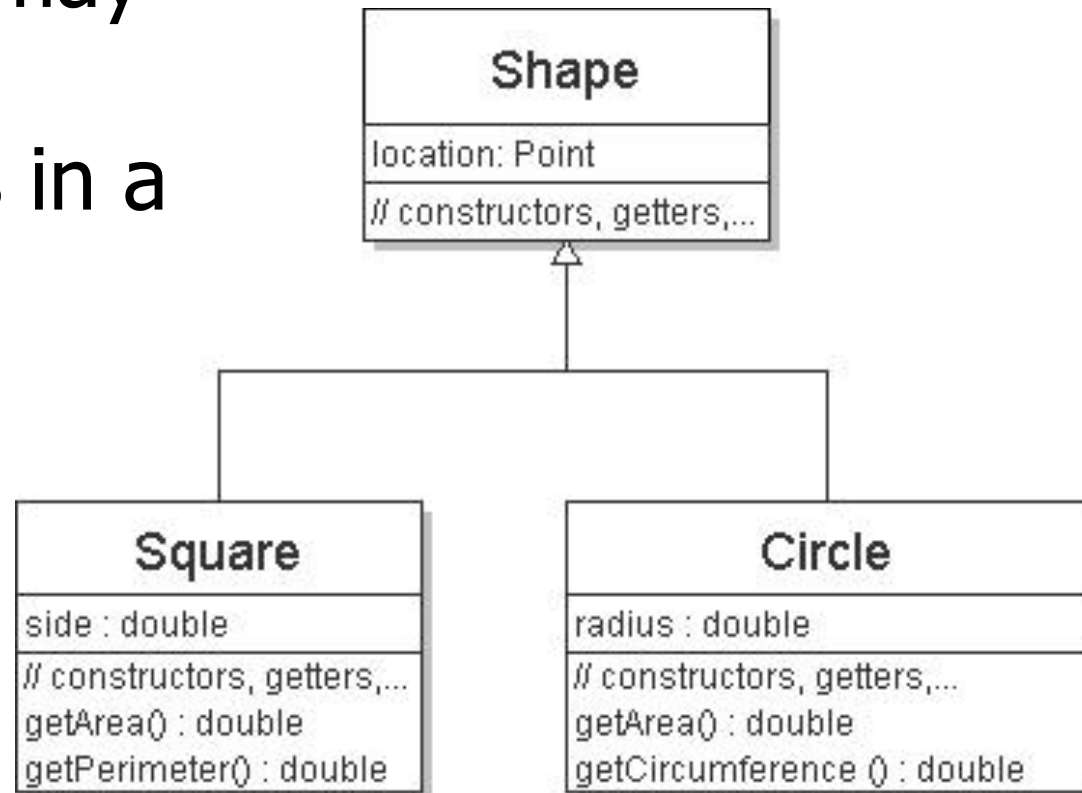
- Any place Object could be used, we could substitute another class:  
Object o = new Student();  
Student s = (Student) o;  
aString1.equals(aString2)





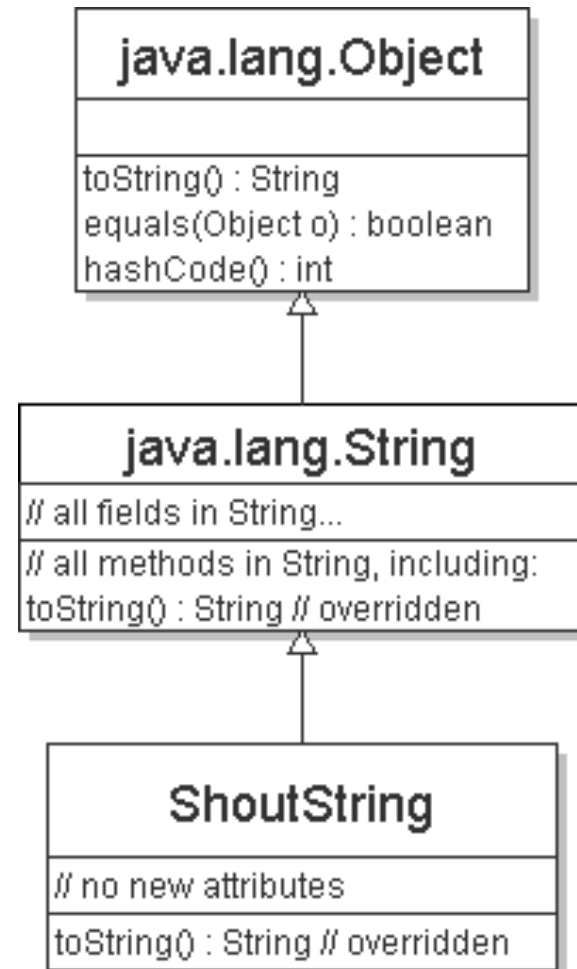
# Inheritance Hierarchy

- Inheritance may reflect IS-A relationships in a domain, e.g. Geometry



# Inheritance for New Functionality

- Inheritance used to modify an existing class to add new functionality
  - E.g. What if we wanted a String that always “prints” in UPPER-CASE
  - Note: in reality, we’re not allowed to extend String (more on this later)



# Code for ShoutString

- In file ShoutString.java:

```
public class ShoutString extends String {  
    public String toString() {  
        // note that this is the current ShoutString object  
        // note that we've inherited toUpperCase from String  
        return this.toUpperCase();  
    }  
}
```

- Note: Again, we're not allowed to extend `java.lang.String` (since it's declared `final` – more later)



# Ways Object is Useful: Generic Type

- Object is a “generic type” – can refer to anything
- ArrayList – can define it to hold superclasses, including Object
  - Advantage : more generally useful!
  - Disadvantage: must take care to cast to correct type when removing item from list
- Note to the “experienced”:
  - Java 1.4 and older: had to use Object

# Example Code

- Code:

```
ArrayList<Object> myList = new
    ArrayList<Object>();
myList.add(1); // add int
myList.add("hello"); // add string
myList.add(new Object()); // add Object
int i = (Integer) myList.get(0);
String s = (String) myList.get(1);
Object o = myList.get(2);
System.out.println(i + " " + s + " " +
o);
```
- Prints:

```
1 hello java.lang.Object@10b62c9
```

# Run-time Polymorphism

- For that same list, consider this loop:

```
for (int j=0; j<myList.size(); ++j)
    System.out.print(myList.get(j) + " ");
```
- What do you think is printed?  
1 hello java.lang.Object@10b62c9
- What's happening here?
  - Each list item is an Object reference
  - We call toString() on that Object reference
  - At run-time, Java calls the appropriate toString() method for the primitive type or sub-class
- This is called *run-time polymorphism*

# Polymorphism Is Powerful

- A object-reference has a type
  - Can be a super-class type
- We call a method on that reference
- At run-time, Java figures out what “sub-type” the reference really points to
- Calls the appropriate sub-type method on the object

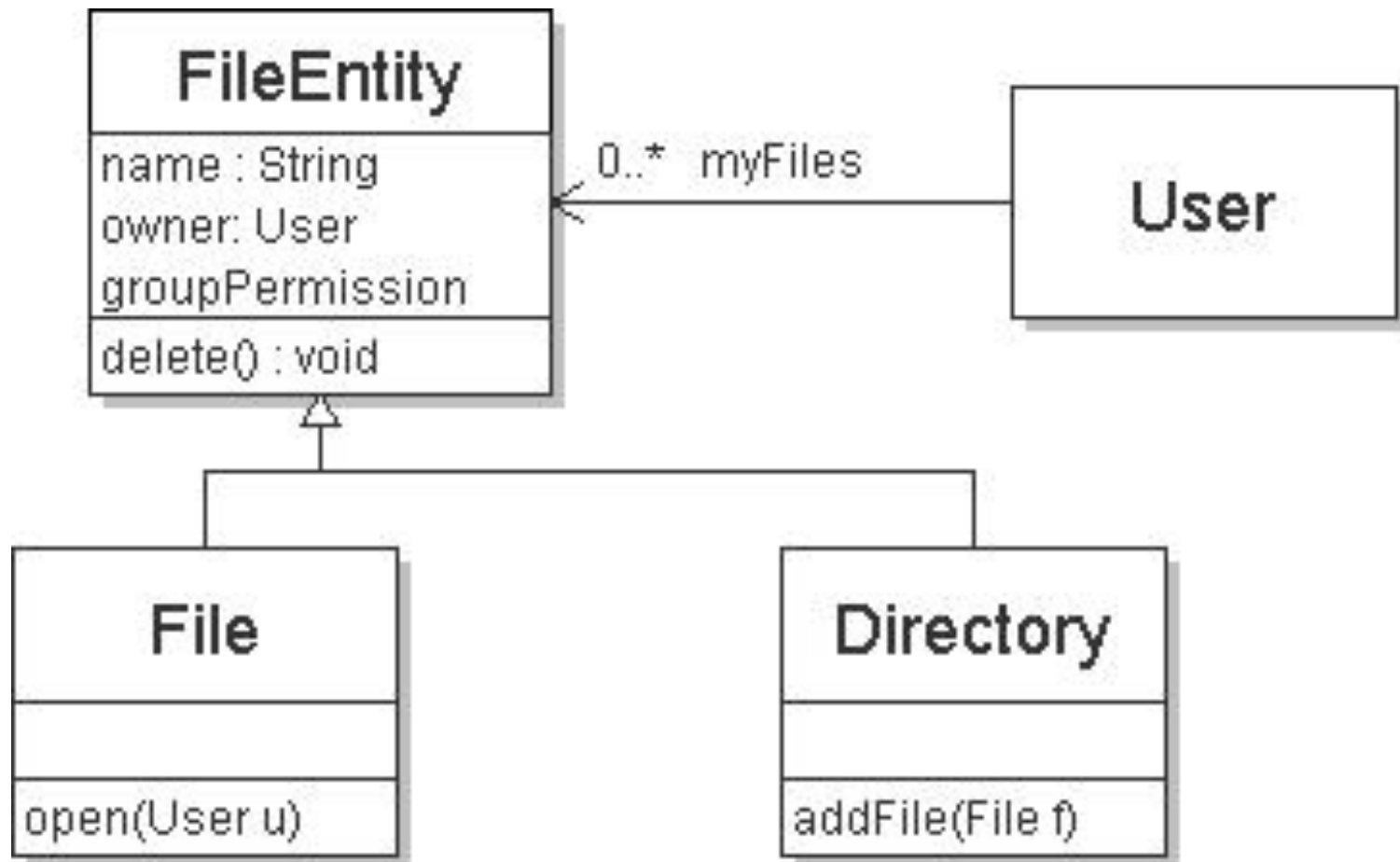
# Inheritance Example #1

- From the file-system example problem:  
File and Directory
  - Shared attributes: name, permission, owner
  - Shared methods, e.g.: delete()
  - Stored together: User object has a list of File objects and Directory objects
- But should one be a superclass of the other?

# Inheritance Example #1

- Create a new superclass: FileEntity
  - Encapsulates fields: name, owner, etc.
  - In User, we now have this field:  
myFiles (list of FileEntity objects)
- File extends FileEntity
  - Adds new fields, methods. Over-rides some methods
- Directory extends FileEntity too

# Class Diagram for Example #1



# But... Something's Different

- For Object, we really could instantiate objects of type Object
- Do we really want objects of type FileEntity?
  - Any given object is either a File or a Directory
  - There's no kind of FileEntity that is neither of these two
- Create *abstract class* when the purpose of superclass is just
  - to capture commonalities
  - to be used as a type for polymorphism
  - and never used to instantiate new objects of that superclass
- It's like saying: "this new type isn't complete as-is, so it must be extended to be useful"

# Abstract Classes in Java

- See Section 3.3.2 in MSD book
- If a class is declared *abstract*
  - You can't create instances of it
  - You can extend it to create a non-abstract sub-class
  - You can use the abstract class' name as a type.

E.g.

```
FileEntity fe = aDirObject.getNext();  
// returns either a file or a dir.
```

- Can use as type when defining an object-reference, perhaps in a method's parameter list

# Abstract Methods

- An abstract class may include *abstract methods*
  - No implementation given in the superclass
  - Declare certain methods *abstract*, and then just a signature and a semicolon. E.g.  
    public abstract void play(); // no body, just ‘;’
  - Each subclass must provide an implementation
    - The superclass requires each subclass to over-ride the method
- Purpose: to guarantee all instances of the abstract class *support an interface*

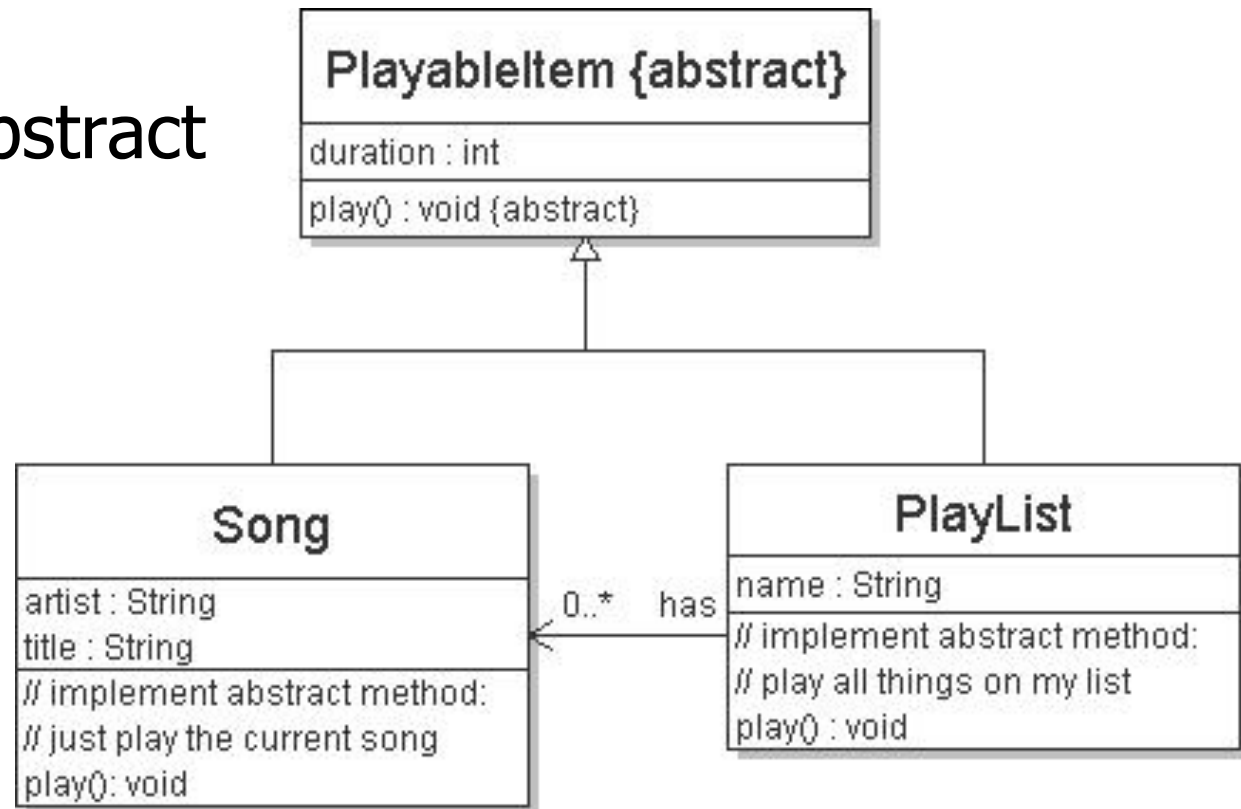
# Example #2: PlayLists and Songs

- We saw this in lab (finding and modeling classes)



# First Update

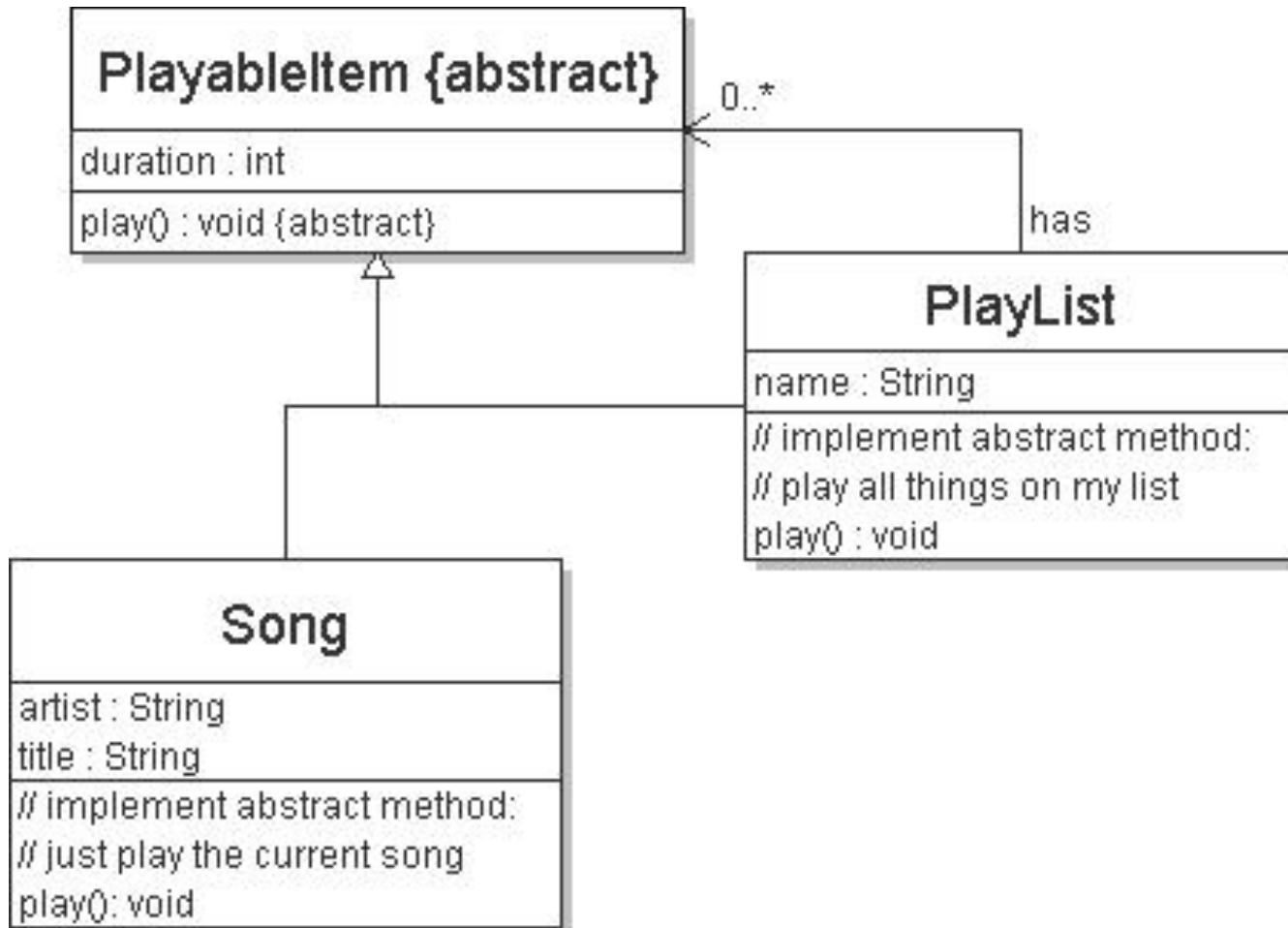
- Abstract class
  - duration
  - play() abstract method



# Could PlayLists “Nest”?

- Could a PlayList contain Songs and other PlayLists?
- This is a common pattern
  - The “Composite Design Pattern”
- Make PlayList contain
  - A list of PlayableItems, not Songs
  - PlayList.play() calls play() on each item
  - Polymorphism at work

# Class Diagram for Composite



# Polymorphism in Action

- Playlist contains a list of Playable  
`ArrayList<Playable> list = ....`
- `PlayList.play()` calls `play()` on each item

```
for (int i=0; i<list.size(); ++i) {  
    list.get(i).play();  
}
```

- Will call the `Song.play()` or `PlayList.play()` depending on what item i really is

**oref is defined of type Super which implements method foo(), but oref really points to an object of type Sub which also implements foo(). oref.foo() calls which?**

1. The implementation of foo defined in Super
2. The implementation of foo defined in Sub
3. It's an error

# Abstract classes: which is true?

1. All methods are not implemented.
2. Can't create concrete instances of that class
3. Can't create subclasses from one
4. More than one of the above are true

# Inheritance: Summary (1)

- Specialization / Extension
  - Inheritance of implementation
  - Use Java keyword “extends”
  - Superclass provide all or some implementations of methods
  - Subclass may
    - add new state or behavior
    - override methods
  - Often reflects a true IS-A relationship

# Summary (2)

- Run-time Polymorphism
  - When a method is called on a reference to a superclass that is actually referring to an instance of a subclass, then...
  - The code executed will be the version defined by the subclass
  - Java determines at run-time what kind of object the superclass reference is really pointing to

# Summary (3)

- Abstract classes
  - Can't create instances of the superclass
  - Used in inheritance of implementation when the superclass **requires** the subclass to implement some behavior
  - Abstract methods (like “stubs”)
    - Note how abstract methods specify interface!
  - Abstract class may give implementation of some methods