LOYOLA
UNIVERSITY
CHICAGO

COMP 413: Intermediate Object-Oriented Programming

Dr. Robert Yacobellis
Advanced Lecturer
Department of Computer Science

Data Types and Data Structures

- Map
  - Unique “keys” link to “values”; get/set/remove; fast look-up
- Queue
  - First-in, first-out behavior (FIFO); enqueue/dequeue/peek
- Priority Queue
  - Orders elements by priority, FIFO within a priority
- Set
  - Unique elements (no dup’s); add, lookup, remove
- Stack
  - Last-in, first-out (LIFO); push/pop/peek

Data Types and Data Structures

- **Linear Data Structures**
  - **Linear data structures are sequential, like computer memory**
  - **Examples: arrays, lists, stacks, queues**
- **Nonlinear Data Structures**
  - **Nonlinear data structures are not sequential**
  - **They may reflect relationships between data elements – a data item could be attached to several other data elements**
  - **Examples: multi-dimensional arrays, trees, tries, graphs**

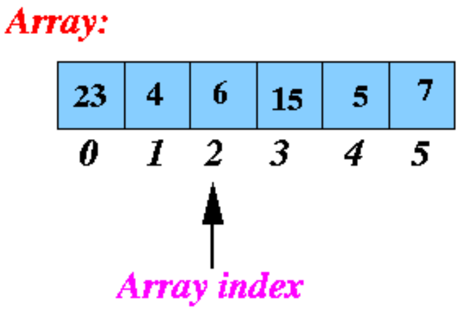
- http://www.differencebetween.com/difference-between-linear-and-vs-nonlinear-data-structures/

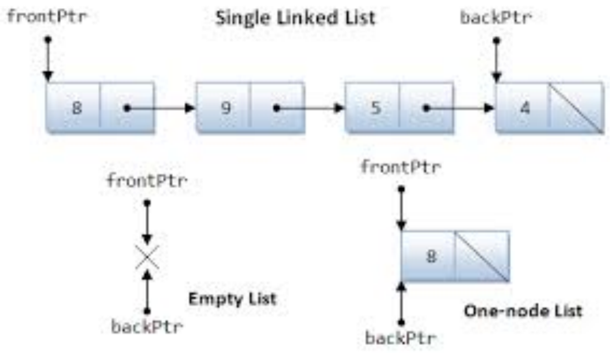
What is a “Trie”?? (Wikipedia)

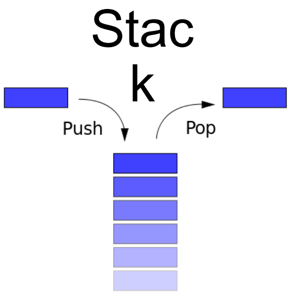
- In computer science, a **trie**, also called **digital tree** and sometimes **radix tree** or **prefix tree** (as they can be searched by prefixes), is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, **no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated**.
- All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.
- The term trie comes from re**trie**val. This term was coined by Edward Fredkin, who pronounces it /ˈtriː/ "tree" as in the word retrieval. However, other authors pronounce it /ˈtraı/ "try", in an attempt to verbally distinguish it from "tree".

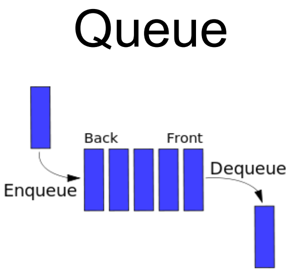
https://examples.javacodegeeks.com/core-java/trie-tutorial-java/

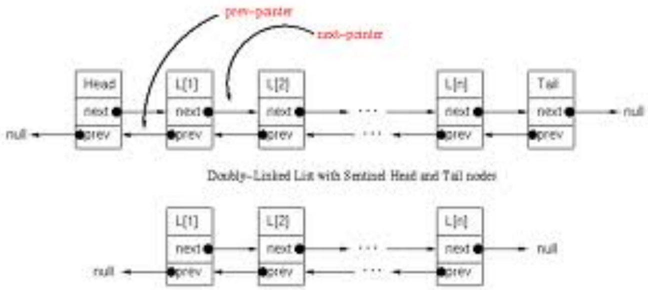
Data Types and Data Structures











Doubly-Linked List with Sentinel Head and Tail nodes

Simplified Doubly-Linked List

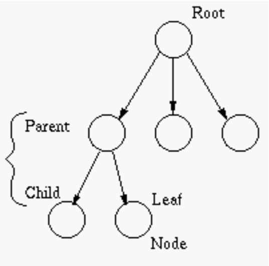
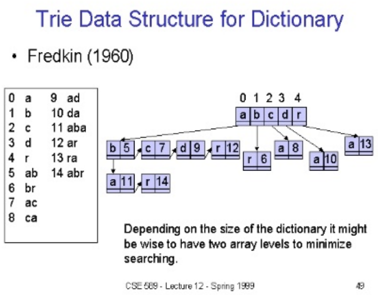


Figure: Tree data structure



Trie Data Structure for Dictionary

- Fredkin (1960)

0	a	9	ad
1	b	10	da
2	c	11	aba
3	d	12	ar
4	r	13	ra
5	ab	14	abr
6	br		
7	ac		
8	ca		

Depending on the size of the dictionary it might be wise to have two array levels to minimize searching.

CSE 589 - Lecture 12 - Spring 1999 49

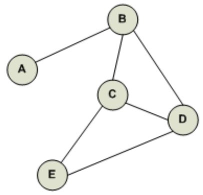


Fig 1. Undirected Graph

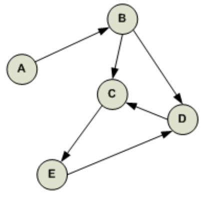


Fig 2. Directed Graph

Data Types and Data Structures

- **Position-Based**
  - **Client controls where adds/removes/gets occur**
  - **There's a first item, second item, … → item 0, 1, … (arrays, …)**
- **Policy-Based**
  - **Do not have the concept of position**
  - **Data structure decides where the next element goes; retrieval may appear to be "random"**
  - **Examples: set (except sorted set), map, priority queue, tree**
- https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy_data_structures.html

Tying Data Structures to Requirements

- **Does it Need to Support Fast Insertion/Deletion?**
  - **An array-based list is not a good implementation**
    - **Arrays are optimized for indexing, not insertion/deletion**
  - **A link list is better (for this requirement)**

- **Does it Need to Support Sorting/Sorted Printing?**
  - **A (hash-based) set is not a good implementation**
    - **This kind of set is search-key based, not order-based**
  - **A sorted set is better (for this requirement)**
  - **Similarly, a generic tree is not good; a red-black tree is better**

http://bigocheatsheet.com/

Tying Data Structures to Requirements

- **Which of the following data structures are good choices for implementing an efficient set abstract data type? (circle one or more)**
  - Array-based list
  - Binary tree (non-search)
  - Binary search tree
  - Hash table
  - Linked list
- **Since a set does not allow duplicate entries, the implementation must check each addition**

Data Structure Algorithm Performance

- **For each of the following use cases, what is the worst case asymptotic order (big-Oh as a formula of *n*) of complexity of the best-known algorithm? (specifically, <u>time</u> complexity)**

a) **<u>Looking up a name</u> in an <u>alphabetically ordered</u> phone book of *n* names →   O(log n) – binary search**

b) **<u>Looking up a name</u> in an <u>unordered</u> sequence of *n* names →   O(n) – worst case must look at all names**

c) **Finding the <u>median</u> value in an <u>unordered</u> sequence of *n* numbers →   O(n log n) – sort first (see item e)**

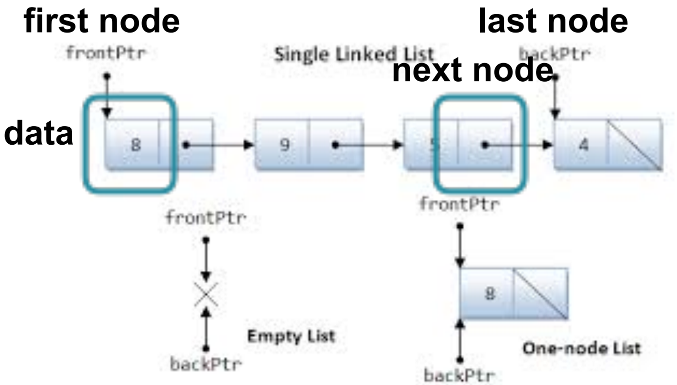
d) **Finding the <u>largest</u> value in an <u>unordered</u> sequence of *n* numbers →   O(n) – must process all entries**

e) **<u>Sorting</u> an <u>unordered</u> sequence of *n* numbers →   O(n log n) – heapsort and mergesort have this worst-case performance**

http://stackoverflow.com/questions/487258/plain-english-explanation-of-big-o

Data Abstraction – Location

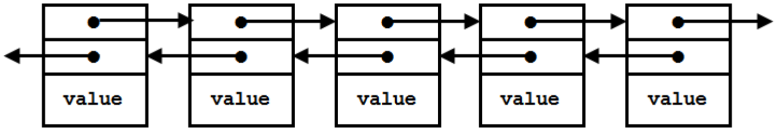
- **Addressing**
    - In abstract data types or data structures we often need to "address" or locate things related to the data structure – other items in it (eg, the <u>next</u> node in a link list) or values that it is managing (eg, the data value at a link list node)



    - In C or C++ we locate things using <u>pointers</u> corresponding to memory addresses; in Java, Python, or C# we mostly locate them using <u>object references</u> or object identifiers

Data Abstraction – Aggregation

- Data items or elements are often aggregated into structures/“product types” (but may be simple types)
  - An array or list of integers refers to simple/primitive values
  - An aggregate or product type (item or element) in a data structure has zero or more components, and then will be an object (in Java) or possibly a struct or record (in C, C++, or C#)
    - Structs are so-called “value types” – *they* are copied when assigned
    - Objects are “reference types” – their *object references* are copied when they are assigned, not their contents
  - Example: a node in a linked list is an aggregate containing its data (itself a simple value or an aggregate) and a next and possibly a previous list node address (a pointer or reference)



Data Abstraction – Variation

- **Data items or elements may also be structured into "sum types" or disjoint unions that allow variation**
  - **A common example is a binary type: an element with 2 possible types, together with a tag to show which type it is**
    - **The general case is an *n*-ary type, with *n* possible tag values**
    - **These might be differentiated using *case* or *switch* statements**
  - **A sum type value in a data structure only has one type at a time; languages with this concept let you determine which, or you can create an aggregate containing the tag and value**
  - **A common OO example is multiple implementations of an *interface*, for example, a *Shape* interface implementation that may in fact be a *Circle*, *Rectangle*, *Hexagon*, …**

http://en.wikipedia.org/wiki/Tagged_union

Example: Mutable Set Abstraction

- A <u>mutable set</u> still only allows unique elements, but may be modified over time through add and remove
  - Often sets are implemented as <u>immutable</u> data structures, and adding/removing requires creating a <u>new</u> set structure
- Common mutable set operations to be implemented
  - Add element
  - Remove element
  - Check whether an element is present
  - Check if the set is empty
  - Determine how many elements are in the set

Example: Mutable Set Implementation

- Implications of mutable set operations
  - Add element – must disallow duplicates → fast search
  - Remove element → fast search, easy restructuring
  - Check whether an element is present → fast search
  - Check if the set is empty → minor constraint
  - Determine how many elements are in the set → keep count
- Reasonable implementations: binary search tree, hash table (or map), bit vector (for <u>small</u> collections)
- Less reasonable: linear implementations like array or linked list → prohibitive in terms of performance

http://docs.oracle.com/javase/tutorial/collections/implementations/
http://imagenious.wordpress.com/2008/02/05/java-bitset-vs-primitive/

What Does “Object-Oriented” Mean?

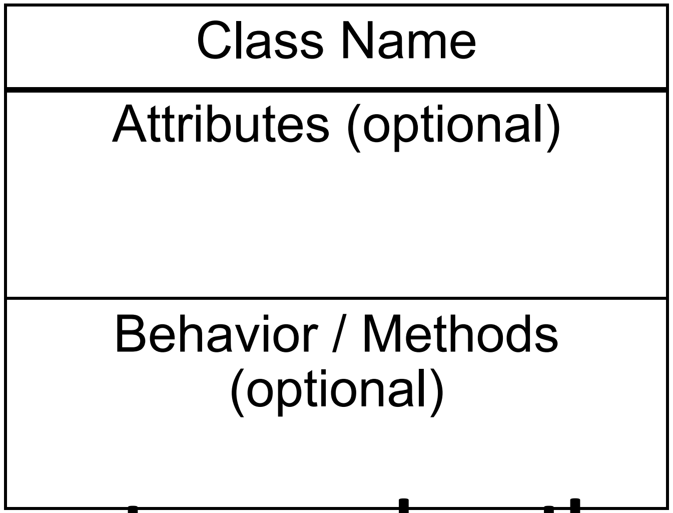
- Object-orientation associates operations with data they operate on; objects have both behavior & data
  - Objects are associated with real-world things or concepts – helps with analysis and design
  - Objects support program maintainability and extensibility
- Key O-O Principles
  - Abstraction – hiding details behind simple interfaces
  - Encapsulation – hiding implementation, combining data (attributes) and operations (methods), protecting data
  - Generalization/specialization – implemented via inheritance
  - Polymorphism – subclasses can stand in for superclasses and redefine their operations (methods)

Classes vs. Objects

- A class generalizes or abstracts a collection of objects
  - Classes provide blueprints for constructing objects – they act as software models of real world things or concepts
  - Classes specify the data elements (attributes, instance variables) and operations (methods) for resulting objects
  - A class can construct (instantiate) any number of objects
- Benefits of O-O Programming
  - Better abstractions by combining information and behavior
  - More comprehensible models of the real world, and less fragile implementations (more maintainable)
  - Better reusability via classes as encapsulated components that can be extended and work together to solve problems

The Unified Modeling Language UML

- **Information about OO classes can be represented by <u>UML class diagrams</u> linked by association lines:**



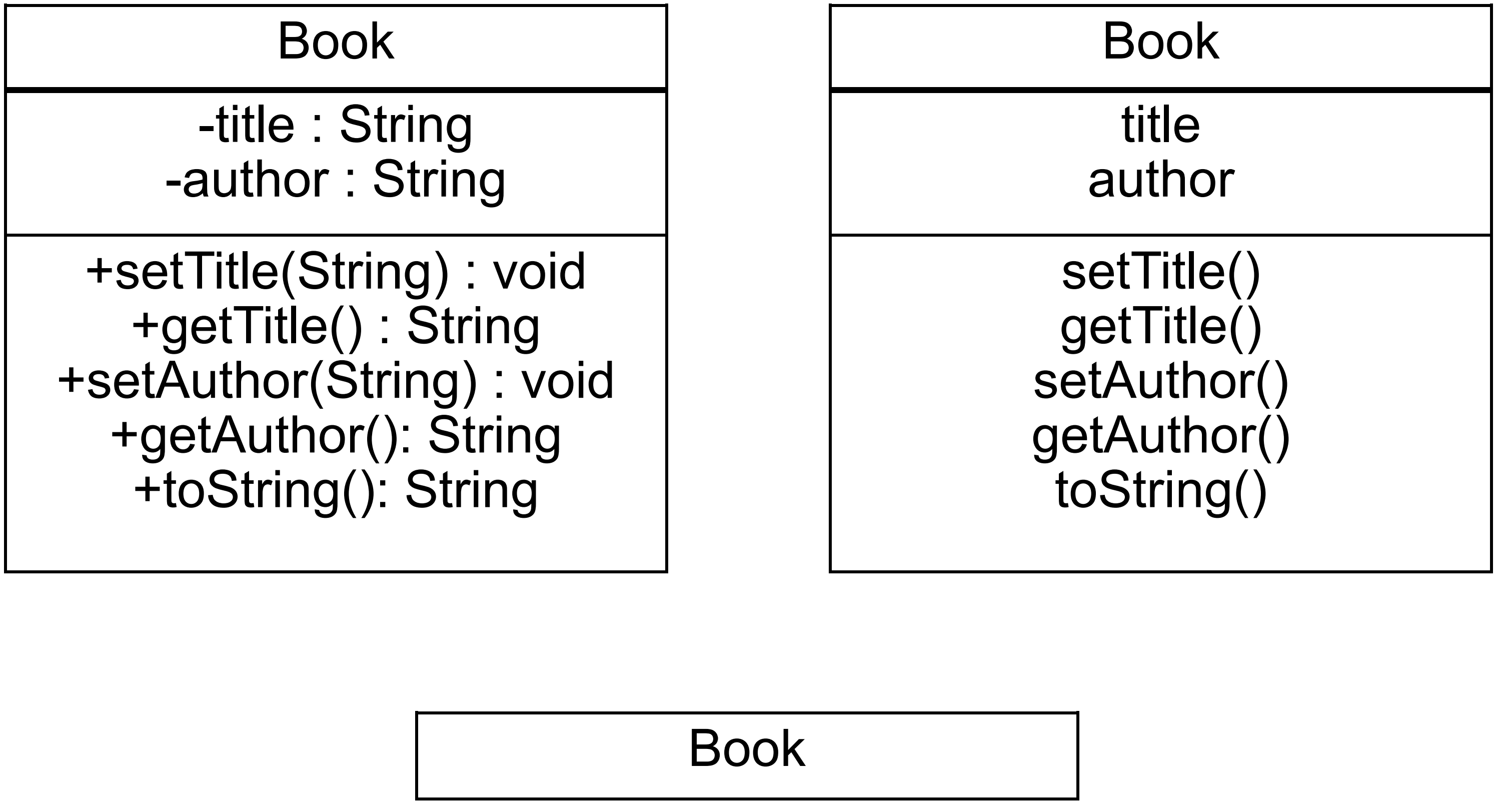
Class Name

Attributes (optional)

Behavior / Methods (optional)

- **Attributes, parameters, and methods are often preceded by visibility indicators (+, -, #, ~) and followed their data type names, like *name : String***

Example UML Class Diagrams



Book

-title : String
-author : String

+setTitle(String) : void
+getTitle() : String
+setAuthor(String) : void
+getAuthor(): String
+toString(): String

Book

title
author

setTitle()
getTitle()
setAuthor()
getAuthor()
toString()

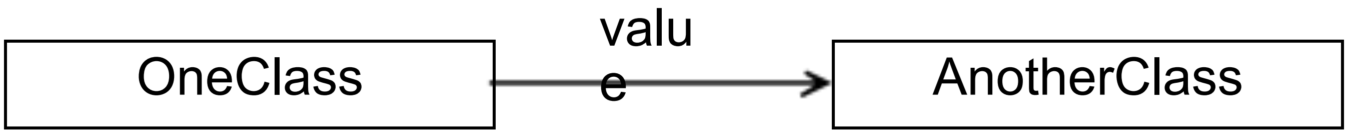
Book

UML Class Diagrams: Relationships

- Relationships between classes are shown by lines and arrows between class diagrams
  - Navigability, multiplicity, dependency, aggregation, and composition
  - Inheritance and interfaces
- Attributes can show multiplicity (so can relationships)
  - n → exactly n of these
  - * → 0 or more
  - m..n → between m and n
- Keywords, notes, and comments can also appear

UML Class Diagrams: Relationships (1)

- **One class has attribute *value* with another class type:**

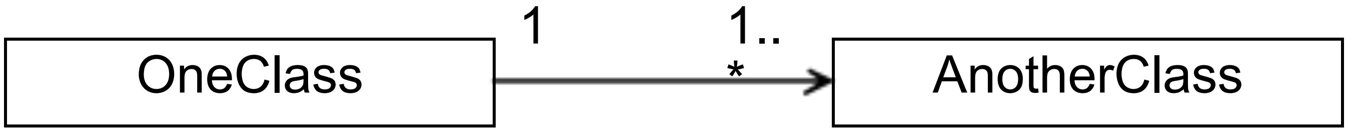


OneClass

valu
e

AnotherClass

- **Class multiplicity relationship (OneClass collection):**



OneClass

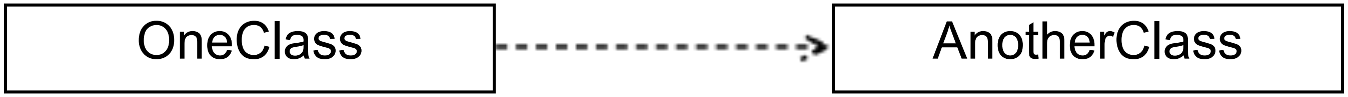
1

1..
*

AnotherClass

UML Class Diagrams: Relationships (2)

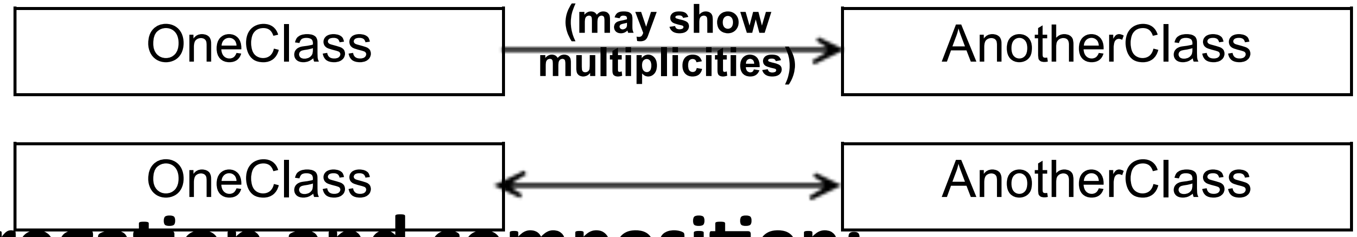
- **One class depends on another (dependency):**



OneClass

AnotherClass

- **Simple association and bi-directional association:**



OneClass

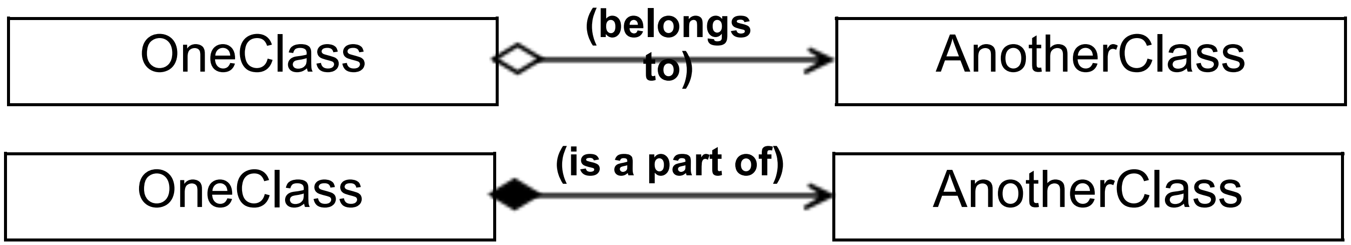
(may show multiplicities)

AnotherClass

OneClass

AnotherClass

- **Aggregation and composition:**



OneClass

(belongs to)

AnotherClass

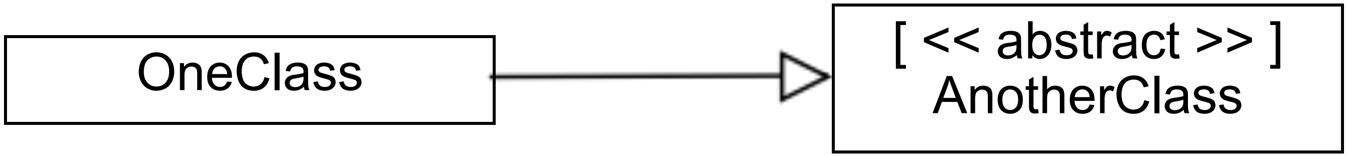
OneClass

(is a part of)

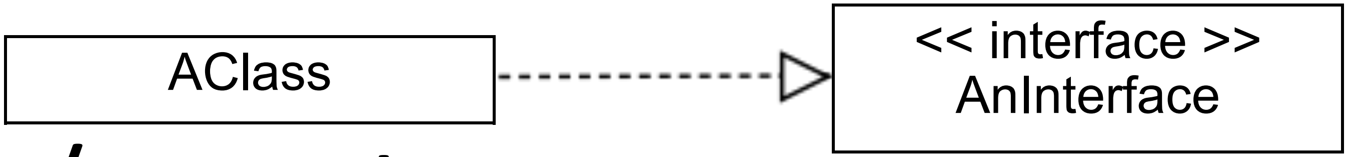
AnotherClass

UML Class Diagrams: Relationships (3)

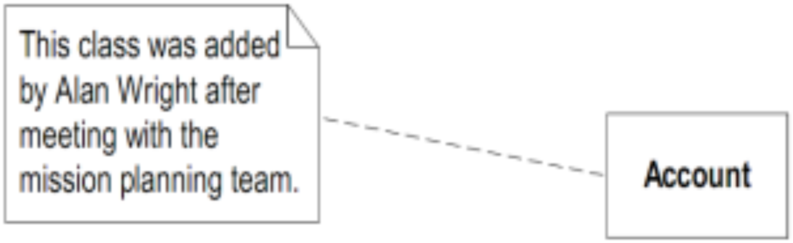
- **One class inherits from another (inheritance):**



- **A class implements an interface:**



- **Notes/comments:**

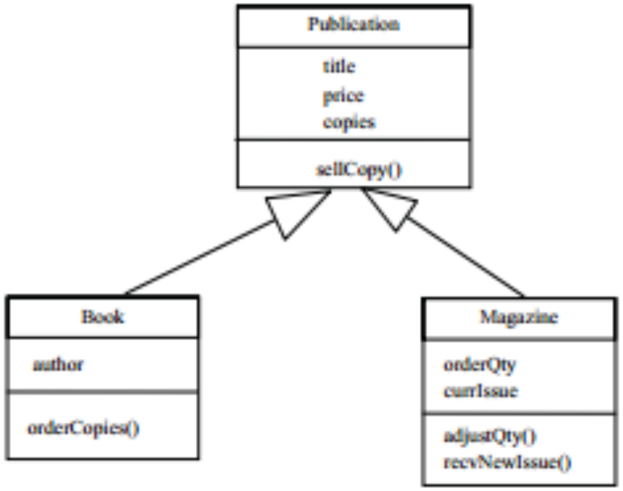

This class was added by Alan Wright after meeting with the mission planning team.

OO Concepts: Inheritance

- **Inheritance represents grouping of objects or classes into related “families” of super- and sub-types**
  - **One familiar hierarchy from nature:**
    Kingdom ← Phylum ← Class ← Order ← Family ← Genus ← Species
  - **Classes / objects “lower” in the hierarchy are related to those above by an “is a” or “is a kind of” relationship**
  - **Classes lower in the hierarchy inherit attributes and methods from all classes above them, and may add additional attributes and methods as appropriate → specialization**
  - **Classes lower in the hierarchy can “stand in” for those above**
    - **If Book inherits from Publication, a Book can act as a Publication**
    - **In that case only Publication methods are available by default; you must *cast* the object as a (Book) to use Book methods**

Generalization vs. Specialization

- **Multiple sub-types of a class can extend it differently:**



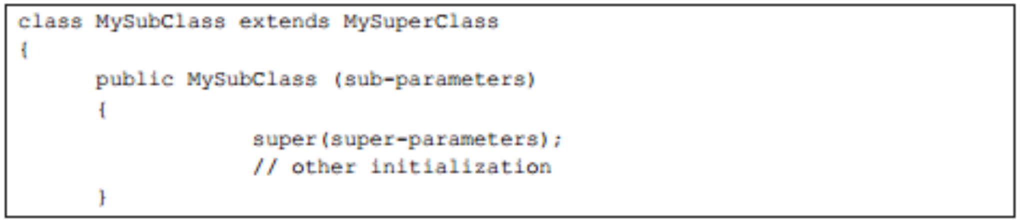
- **In addition, sub-types of a class can “override” non-*final* methods of the class by <u>redefining</u> them in order to provide specialized behavior, and can invoke superclass methods as needed via *super.method(…)***
  - **In Java, methods are invoked based on the <u>runtime type</u> of an object, not its declared type (the type of its variable)**

Java Class “Boilerplate”

```
package com.example.myclass; // like a C# namespace

import java.util.*; // like C# using, Python module, etc.
// gives access to APIs/classes and possibly static items via import static …

public class MyClass [ extends OtherClass ] { // only 1 public class per file
// source file must be named MyClass.java; classes use Pascal case

    private String name = " "; // private instance variable – uses Camel case

    public MyClass( [ parameters ] ) { // constructor – same name as class
        // no return type; super() automatically called unless explicit, then first
    }

    @Override // Java annotation – method below must override existing one
    public String toString() { // public method – uses Camel case; from Object
        return "Hi, my name is " + name; // compatible returned type value
    }
    …
}
```

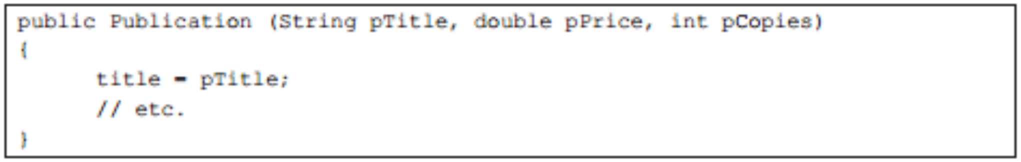
Inheritance in Java

- Java uses the *extends* keyword to denote inheritance
- Subclass constructors can reference superclass constructors, again by using the *super* keyword

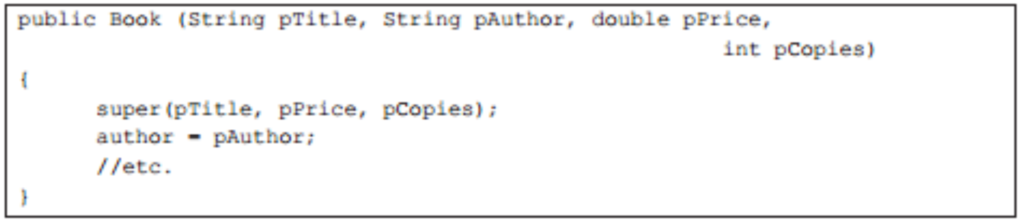


```
class MySubClass extends MySuperClass
{
      public MySubClass (sub-parameters)
      {
                  super(super-parameters);
                  // other initialization
      }
```

- Example using Publication and Book:



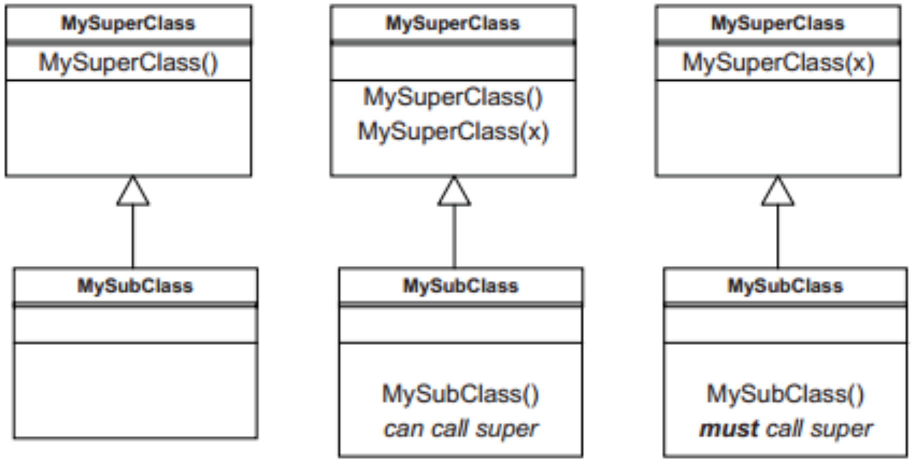
```
public Publication (String pTitle, double pPrice, int pCopies)
{
      title = pTitle;
      // etc.
}
```



```
public Book (String pTitle, String pAuthor, double pPrice,
                                                         int pCopies)
{
      super(pTitle, pPrice, pCopies);
      author = pAuthor;
      //etc.
}
```

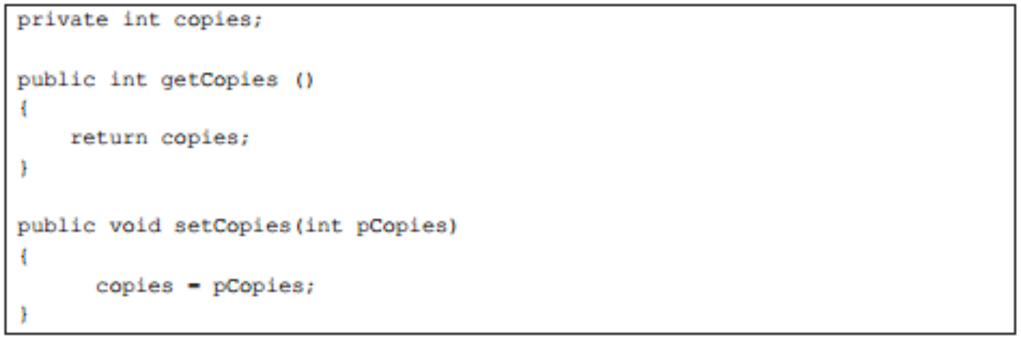
Java Constructor Rules

- **If a superclass has a parameter-less or default constructor, it will be called automatically in a subclass constructor (unless there is a *super()* call)**
- **If a superclass does not have a default constructor, every subclass constructor <u>must</u> have a *super()* call:**



Java Access Control – Instance Variables

- **We enforce encapsulation in Java by making instance variables *private* (or *protected* so subclasses can access them) and providing getter and setter methods (also called accessors and mutators)**
  - **Mutator methods in particular, together with instance variable initial values and/or constructor-provided values, must guarantee <u>consistent state</u> of instance variables**
  - **Example: In class Publication …**



```
private int copies;

public int getCopies ()
{
    return copies;
}

public void setCopies(int pCopies)
{
      copies = pCopies;
}
```

Java Access Control – Methods

- Class methods can also be marked *private* if they are only used within the class, or *protected* if they should be available for use within subclasses
- *public* or *protected* superclass methods can be accessed in subclasses directly (via inheritance)
  - Accessor and mutator methods mentioned above are often used this way in subclasses
- Special case: if a class method has access to an object of the same class or a subclass, it will have access to <u>private</u> instance variables and methods of that object!

Methods With Object Parameters

- **Special case: if a class method has access to an object of the same class or a subclass, it will have access to private instance variables and methods from its own class in that object!**
- **Example**

```
public class C {
     private int x = 0; // private instance variable initialized to 0
     public void showX() { System.out.println(x); } // prints x’s value
     public static void changeX(C c) { c.x = 42; }
     // even though x is private changeX can access it in the c object
     public static void main(String[] args) {
         C newC = new C(); // newC’s x is initialized to 0
         newC.showX(); // prints 0
         // this won’t compile since x is private: newC.x = 42;
         C.changeX(newC); // static method call, changes newC’s x!
         newC.showX(); // prints 42!
     }
 }
```

Java Abstract Classes

- Some classes, like Publication, are not designed to be instantiated (used to create objects), they only exist to be inherited / subclassed
- Such classes can be marked *abstract*:

  public abstract class Publication { … }

- Abstract classes cannot be instantiated, but otherwise they are just like "concrete" classes – they can define instance variables, constructors, and methods that inheriting classes can make use of
  - Abstract class methods can also be *abstract*, with no method body; then every concrete subclass must implement them

Overriding Superclass Methods in Java

- **Java implements polymorphism by allowing visible superclass methods to be <u>overridden</u> in subclasses, meaning that the subclass can redefine the methods**
  - **Such redefined methods must have definitions that match those in the superclass – the parameter lists of the overriding methods must be the same, and the return types must be compatible (the same type or a subtype)**
  - **The version of a method that Java uses at runtime depends on the created type of the object it is associated with (eg, a subclass object), <u>not</u> the type of the variable it is assigned to**
- **If the subclass method needs to use the overridden superclass method, it can call it as *super.method(…)***

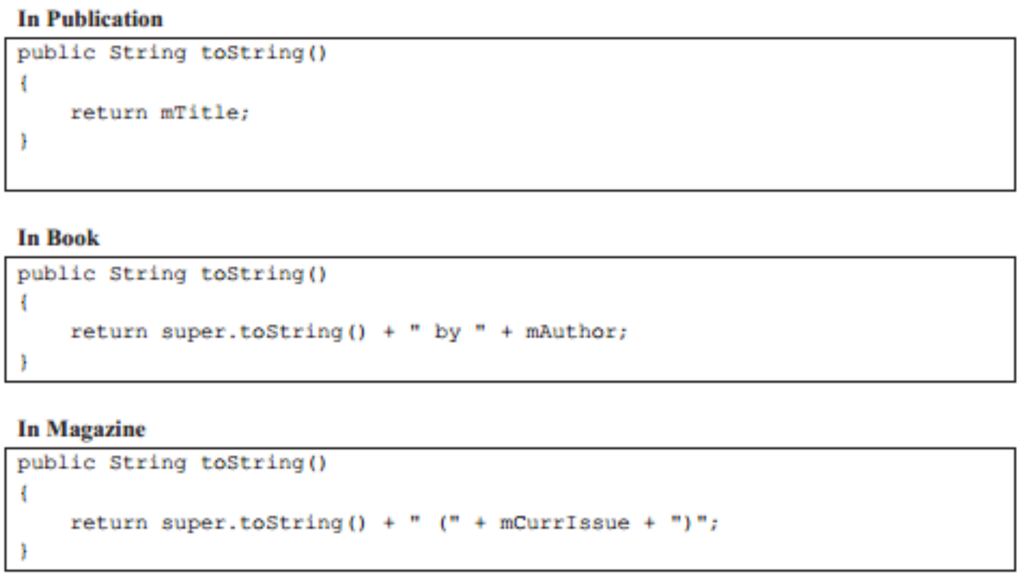
The Object Class in Java

- **Java provides an Object class which is the parent or superclass of all other classes; Java classes implicitly *extend* Object even if they don't say so**
  - **Object is at the top of the inheritance hierarchy for all classes**
- **Object is a concrete class that provides several methods, in particular *toString()* as shown above**
  - **In the Object definition of *toString()* what is returned is the type of the object and the object identifier (like its address)**
  - **We often want a more user-friendly or useful definition of *toString()*, and we can get that by overriding Object's version**
  - **The method signature of Object's *toString()* is:**

```
public String toString() // overriding definitions must match
```

Overriding Object's *toString()*

- **An example from the Publication/Book/Magazine class hierarchy of overriding Object's *toString()*:**



In Publication

```
public String toString()
{
    return mTitle;
}
```

In Book

```
public String toString()
{
    return super.toString() + " by " + mAuthor;
}
```

In Magazine

```
public String toString()
{
    return super.toString() + " (" + mCurrIssue + ")";
}
```

Reference Semantics vs. Value Semantics

- provide *primitive* or *value types* that are copied when they are assigned to variables – value semantics
  - For example, if int x = 3; int y = x; y = 42; in either language, variable x will still contain the int value 3 but y will be 42: in both cases the variables actually contain the values
  - Java provides 8 different *primitive types* that operate this way: int, short, byte, char, long, float, double, and boolean
  - C# has similar *value types*, but can create *structs*, also value types, which can have methods and other characteristics; in any case all C# value type variables contain their values, and all C# value types are copied when assigned

Reference Semantics vs. Value Semantics

- **provide objects, and variables assigned objects contain references to them reference semantics**
  - **For example, if**

    ```
    Book b = new Book("title", “author", 10, 1000);
    Book b2 = b; b.setTitle("new title");
    ```

    in either language, the single object that variable b and variable b2 both refer to will now have title "new title"
  - **Since variables b and b2 both refer to the same object in memory, doing a setTitle on one appears to change both**
  - **Note: It is possible to create a copy of an existing object by using *clone*, but this is not the default behavior for objects**

Reference Semantics vs. Value Semantics

- **Because of value vs. reference semantics, the types of method parameters can have an impact on whether changes made in a method are propagated outside**
  - **In both Java and C# the default behavior for a <u>value</u> type parameter is that assigning to that parameter only impacts the local copy of the parameter inside the method**
    - **C# (only) provides an *out* modifier to allow changing the value of a value type variable passed as a parameter**
  - **Both Java and C# work the same way with a <u>reference</u> type parameter – assigning to that parameter does not change <u>its</u> value outside the method; however, running methods that change that object's state produce visible changes outside**

Equality vs. Identity

- In Java, the == operator always tests for object identity, that is, whether two object references refer to the same object in memory (same address)
  - In the earlier example with Books, b == b2 is *true*
- The Object class provides an equals() method that does the same thing by default – it tests for identity; however, often we want equals() to test for equality in the sense that two objects have "the same" values in key instance variables, or some derived values
  - If b and b2 are Books, we may want b.equals(b2) to be *true* if b and b2 have the same title and author

Equality vs. Identity

- We can override equals(), subject to restrictions:
  - The method signature is: public boolean equals(Object obj)
  - The equals() method must implement an equivalence relation on non-null object references
    - Reflexive: if x is not null, x.equals(x) must be true
    - Symmetric: x.equals(y) must be true iff y.equals(x) is true
    - Transitive: if x.equals(y) is true and y.equals(z) is true then x.equals(z) must be true
    - Consistent: x.equals(y) must produce the same value each time, unless information used in the equality test has changed
    - For any non-null x, x.equals(null) must be false
    - hashCode(): if x.equals(y) then their hash codes must be equal

Equality vs. Identity

- **A typical implementation of equals() in class MyClass:**

```
@Override           // we’re overriding Object equals()
public boolean equals(Object that) {
      return this == that ||     // test for object identity first
       that instanceof MyClass // then test for correct class
       && << equality tests of final instance variables, etc >>
} // note: that instanceof MyClass fails if that == null
// the test depends on && having higher priority than ||
// now you must also override the Object hashCode() method
```

- **See** https://github.com/oop-cs-luc-edu/misc-java/blob/master/src/misc/IdentityAndEquality.java **and** http://www.artima.com/lejava/articles/equality.html