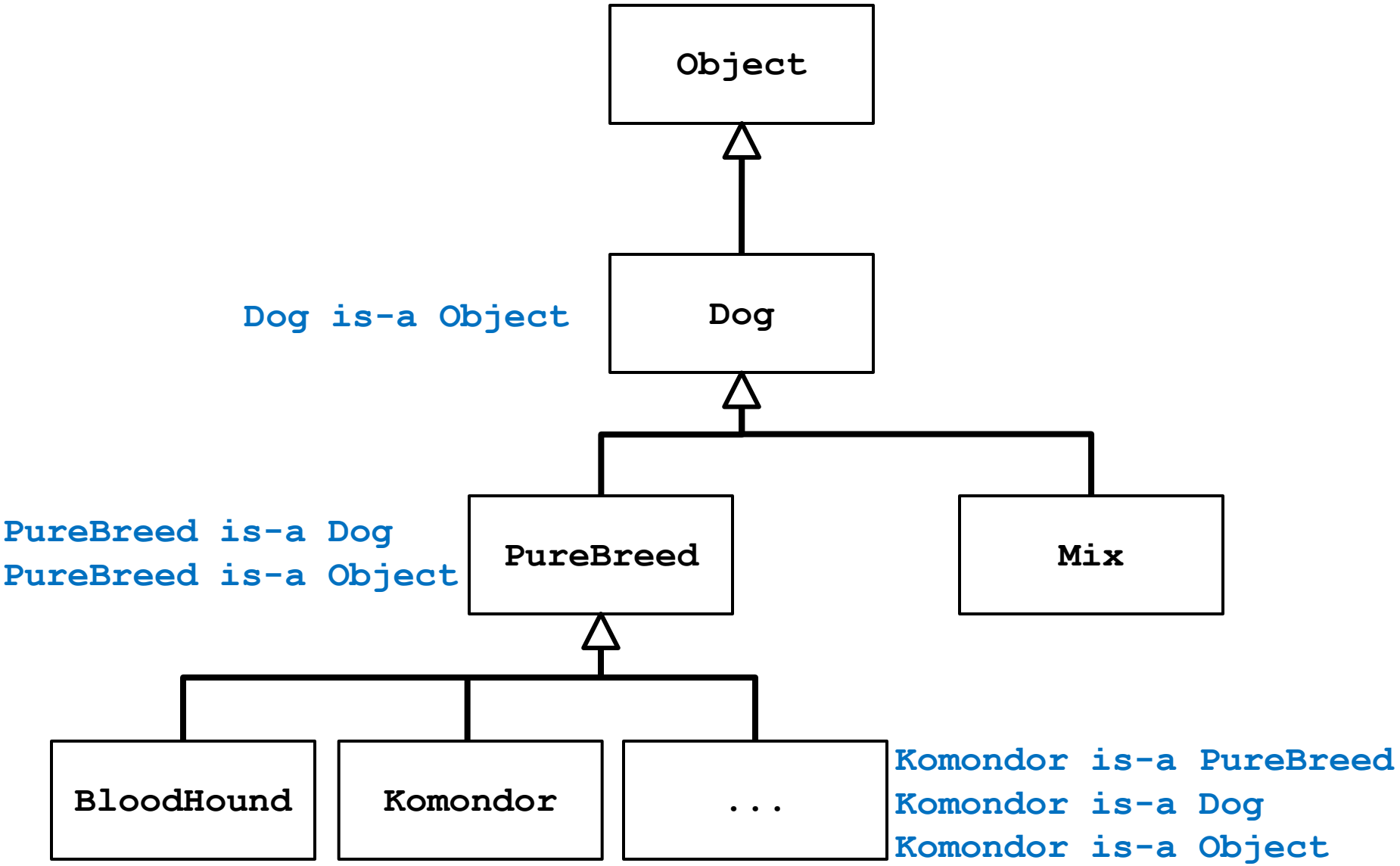
Inheritance

Notes Chapter 6

Inheritance

- you know a lot about an object by knowing its class
  - for example what is a Komondor?

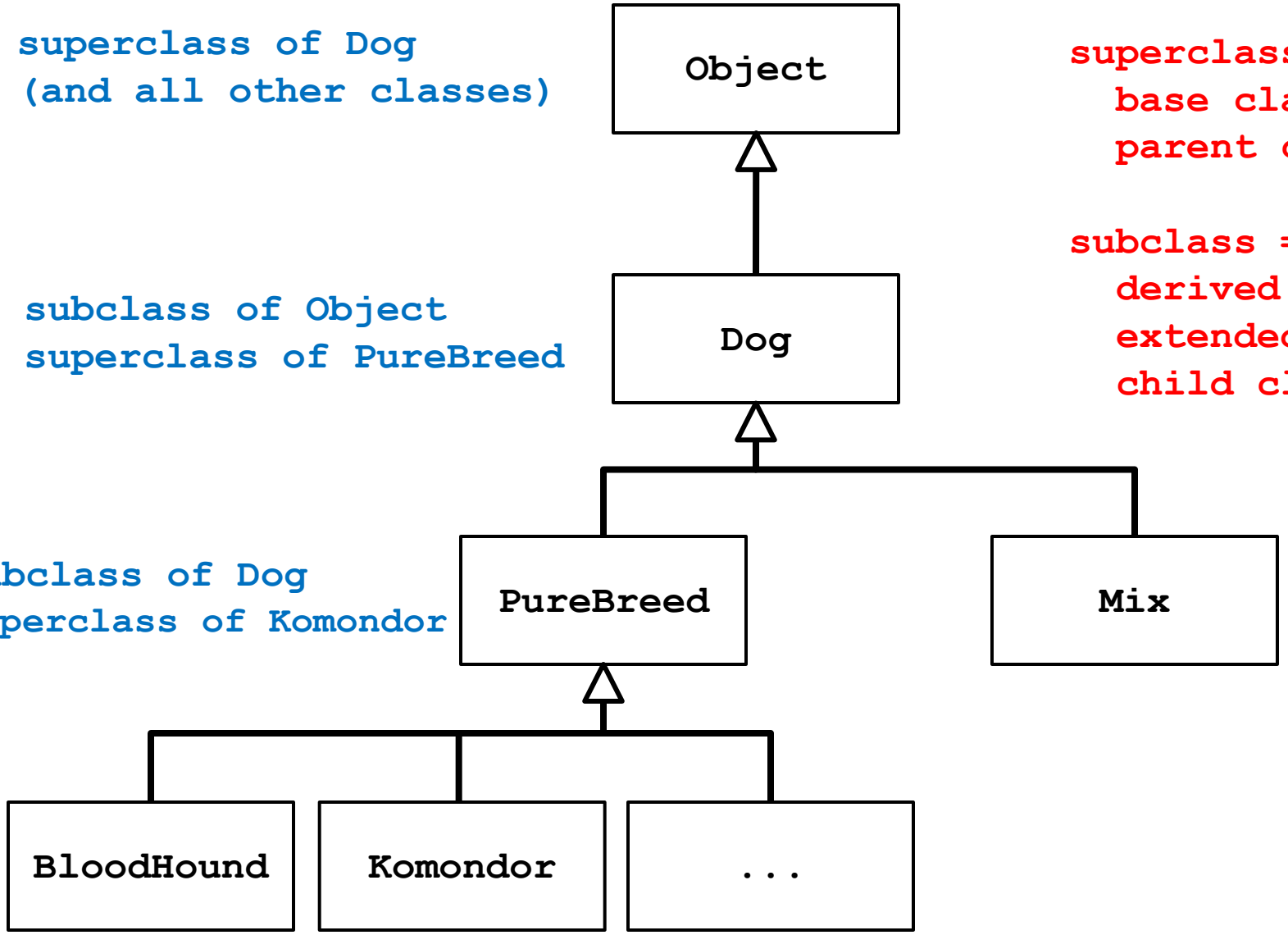
http://en.wikipedia.org/wiki/File:Komondor_delvin.jpg


Object
Dog is-a Object
Dog
PureBreed is-a Dog
PureBreed is-a Object
PureBreed
Mix
BloodHound
Komondor
...
Komondor is-a PureBreed
Komondor is-a Dog
Komondor is-a Object

superclass of Dog
(and all other classes)

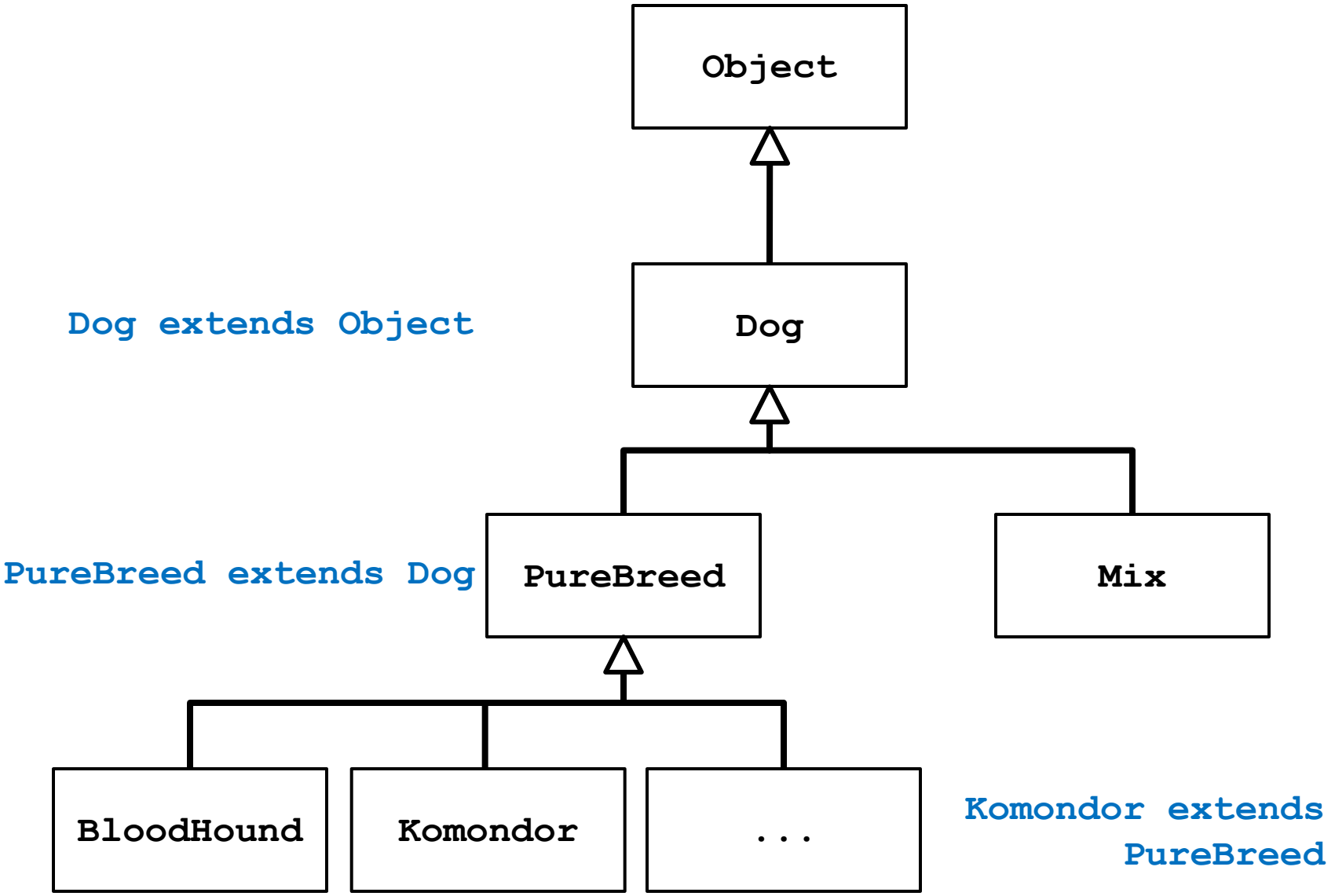
subclass of Object
superclass of PureBreed

subclass of Dog
superclass of Komondor



superclass ==
base class
parent class

subclass ==
derived class
extended class
child class


Object
Dog extends Object
Dog
PureBreed extends Dog
PureBreed
Mix
BloodHound
Komondor
...
Komondor extends PureBreed

Some Definitions

- we say that a subclass is derived from its superclass
- with the exception of **`Object`**, every class in Java has one and only one superclass
  - Java only supports *single inheritance*
- a class **`X`** can be derived from a class that is derived from a class, and so on, all the way back to **`Object`**
  - **`X`** is said to be descended from all of the classes in the inheritance chain going back to **`Object`**
  - all of the classes **`X`** is derived from are called ancestors of **`X`**

Why Inheritance?

- a subclass inherits all of the non-private members (attributes and methods ***but not constructors***) from its superclass
  - if there is an existing class that provides some of the functionality you need you can derive a new class from the existing class
  - the new class has direct access to the **`public`** and **`protected`** attributes and methods without having to re-declare or re-implement them
  - the new class can introduce new fields and methods
  - the new class can re-define (override) its superclass methods

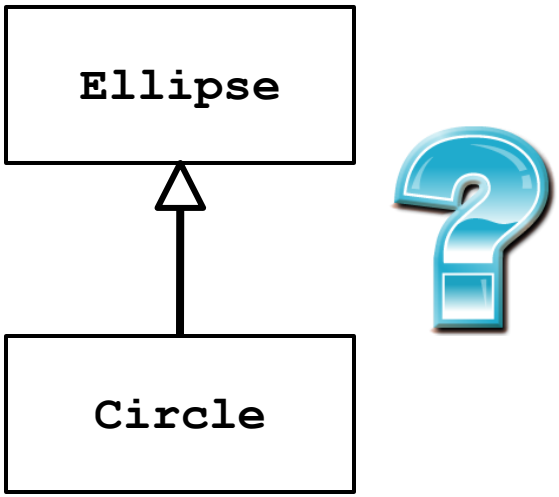
Is-A

- inheritance models the is-a relationship between classes
- from a Java point of view, is-a means you can use a derived class instance in place of an ancestor class instance

```
public someMethod(Dog dog)
{ // does something with dog }

// client code of someMethod

Komondor shaggy = new Komondor();
someMethod( shaggy );

Mix mutt = new Mix ();
someMethod( mutt );
```

Is-A Pitfalls

- is-a has nothing to do with the real world
- is-a has everything to do with how the implementer has modelled the inheritance hierarchy
- the classic example:
  - **`Circle`** is-a **`Ellipse`**?



`Circle` is-a `Ellipse`?

- if `Ellipse` can do something that `Circle` cannot, then `Circle` is-a `Ellipse` is false
  - remember: is-a means you can substitute a derived class instance for one of its ancestor instances
    - if `Circle` cannot do something that `Ellipse` can do then you cannot (safely) substitute a `Circle` instance for an `Ellipse` instance

```
// method in Ellipse
/*
 * Change the width and height of the ellipse.
 * @param width The desired width.
 * @param height The desired height.
 * @pre. width > 0 && height > 0
 */
public void setSize(double width, double height)
{
  this.width = width;
  this.height = height;
}
```

- there is no good way for **`Circle`** to support **`setSize`** (assuming that the attributes **`width`** and **`height`** are always the same for a **`Circle`**) because clients expect **`setSize`** to set both the width and height
- can't **`Circle`** override **`setSize`** so that it throws an exception if **`width != height`**?
  - no; this will surprise clients because **`Ellipse`** **`setSize`** does not throw an exception if **`width != height`**
- can't **`Circle`** override **`setSize`** so that it sets **`width == height`**?
  - no; this will surprise clients because **`Ellipse`** **`setSize`** says that the **`width`** and **`height`** can be different

- But I have a Ph.D. in Mathematics, and I'm *sure* a Circle is a kind of an Ellipse! Does this mean Marshall Cline is stupid? Or that C++ is stupid? Or that OO is stupid? [C++ FAQs http://www.parashift.com/c++-faq-lite/proper-inheritance.html#faq-21.8]
  - Actually, it doesn't mean any of these things. But I'll tell you what it does mean — you may not like what I'm about to say: it means your intuitive notion of "kind of" is leading you to make bad inheritance decisions. Your tummy is lying to you about what good inheritance really means — stop believing those lies.

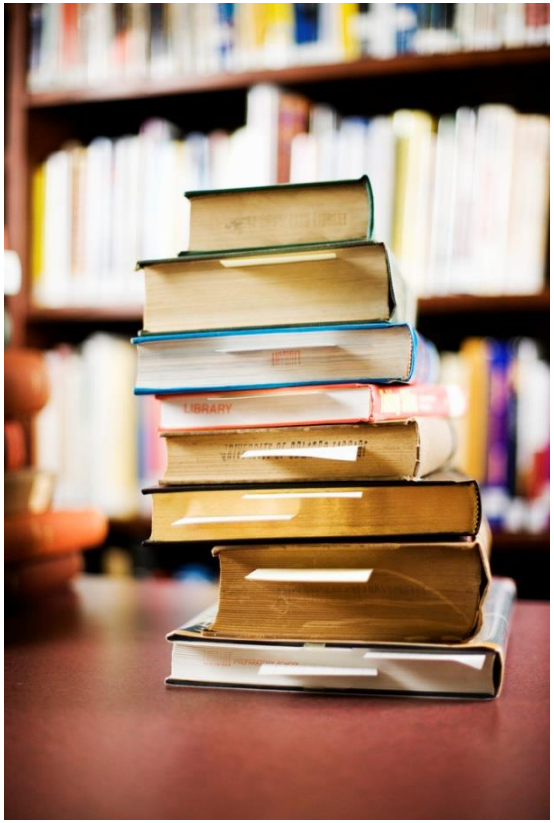
- what if there is no **`setSize`** method?
  - if a **`Circle`** can do everything an **`Ellipse`** can do then **`Circle`** can extend **`Ellipse`**

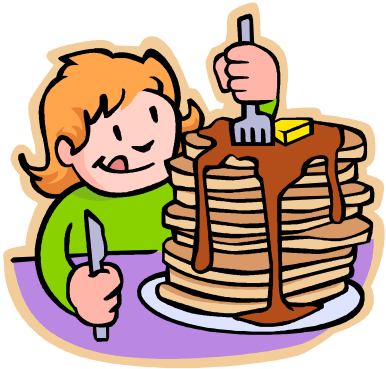
A Naïve Inheritance Example

- a stack is an important data structure in computer science
  - data structure: an organization of information for better algorithm efficiency or conceptual unity
    - e.g., list, set, map, array
- widely used in computer science and computer engineering
  - e.g., undo/redo can be implemented using two stacks

Stack

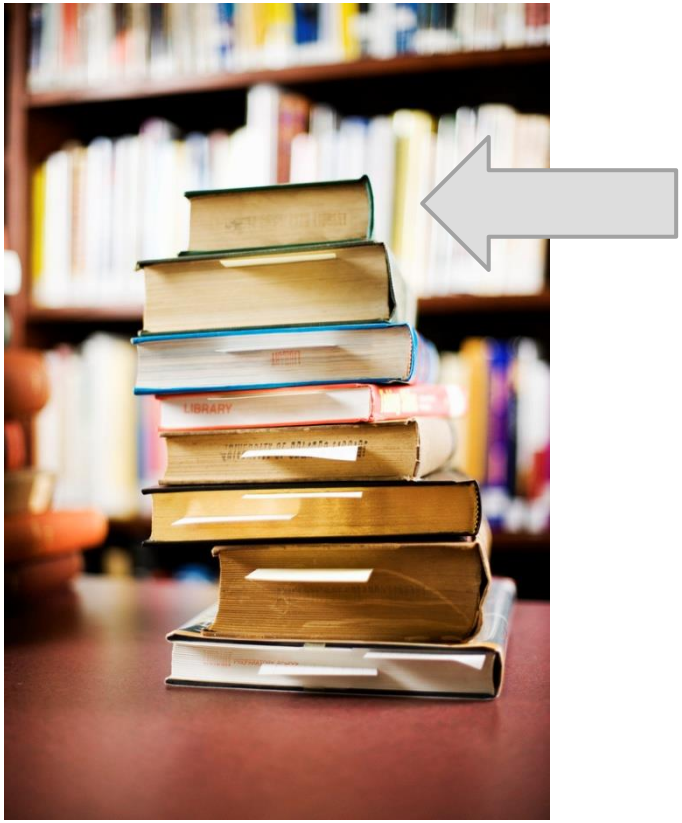
- examples of stacks





Top of Stack

- top of the stack



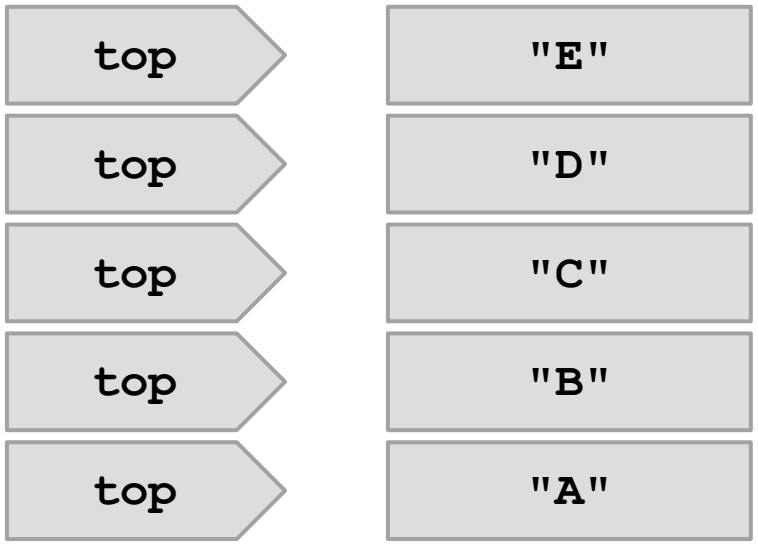
Stack Operations

- classically, stacks only support two operations
  1. push
     - add to the top of the stack
  2. pop
     - remove from the top of the stack

- there is no way to access elements of the stack except at the top of the stack

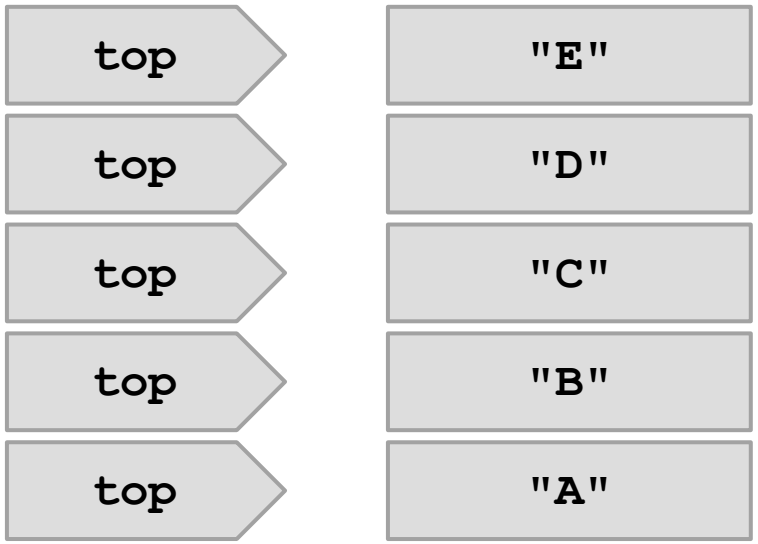
Push

1. `st.push("A")`
2. `st.push("B")`
3. `st.push("C")`
4. `st.push("D")`
5. `st.push("E")`



Pop

1. `String s = st.pop()`
2. `s = st.pop()`
3. `s = st.pop()`
4. `s = st.pop()`
5. `s = st.pop()`



Implementing stack using inheritance

- a stack looks a lot like a list
  - pushing an element onto the top of the stack looks like adding an element to the end of a list
  - popping an element from the top of a stack looks like removing an element from the end of the list

- if we have stack inherit from list, our stack class inherits the **`add`** and **`remove`** methods from list
  - we don't have to implement them ourselves

- let's try making a stack of integers by inheriting from **`ArrayList<Integer>`**

Implementing stack using inheritance

```
import java.util.ArrayList;

public class BadStack extends ArrayList<Integer> {


}
```

use the keyword **`extends`** followed by the name of the class that you want to extend

Implementing stack using inheritance

```
import java.util.ArrayList;

public class BadStack extends ArrayList<Integer> {

  public void push(int value) {
    this.add(value);
  }

  public int pop() {
    int last = this.remove(this.size() - 1);
    return last;
  }

}
```

push = add to end of this list

pop = remove from end of this list

Implementing stack using inheritance

- that's it, we're done!

```
public static void main(String[] args) {
  BadStack t = new BadStack();
  t.push(0);
  t.push(1);
  t.push(2);
  System.out.println(t);
  System.out.println("pop: " + t.pop());
  System.out.println("pop: " + t.pop());
  System.out.println("pop: " + t.pop());
}
```

```
[0, 1, 2]
pop: 2
pop: 1
pop: 0
```

Implementing stack using inheritance

- why is this a poor implementation?
- by having **`BadStack`** inherit from **`ArrayList<Integer>`** we are saying that a stack is a list
  - anything a list can do, a stack can also do, such as:
    - get a element from the middle of the stack (instead of only from the top of the stack)
    - set an element in the middle of the stack
    - iterate over the elements of the stack

Implementing stack using inheritance

```
public static void main(String[] args) {
   BadStack t = new BadStack();
   t.push(100);
   t.push(200);
   t.push(300);
   System.out.println("get(1)?: " + t.get(1));
   t.set(1, -1000);
   System.out.println("set(1, -1000)?: " + t);
 }
```

```
[100, 200, 300]
get(1)?: 200
set(1, -1000)?: [100, -1000, 300]
```

Implementing stack using inheritance

- using inheritance to implement a stack is an example of an incorrect usage of inheritance
- inheritance should only be used when an is-a relationship exists
  - a stack is not a list, therefore, we should not use inheritance to implement a stack
- even experts sometimes get this wrong
  - early versions of the Java class library provided a stack class that inherited from a list-like class
    - **`java.util.Stack`**