EECS 2030
Advanced Object-Oriented Programming

S2023, Section A

Inheritance, comparison with composition

Aggregation and Composition

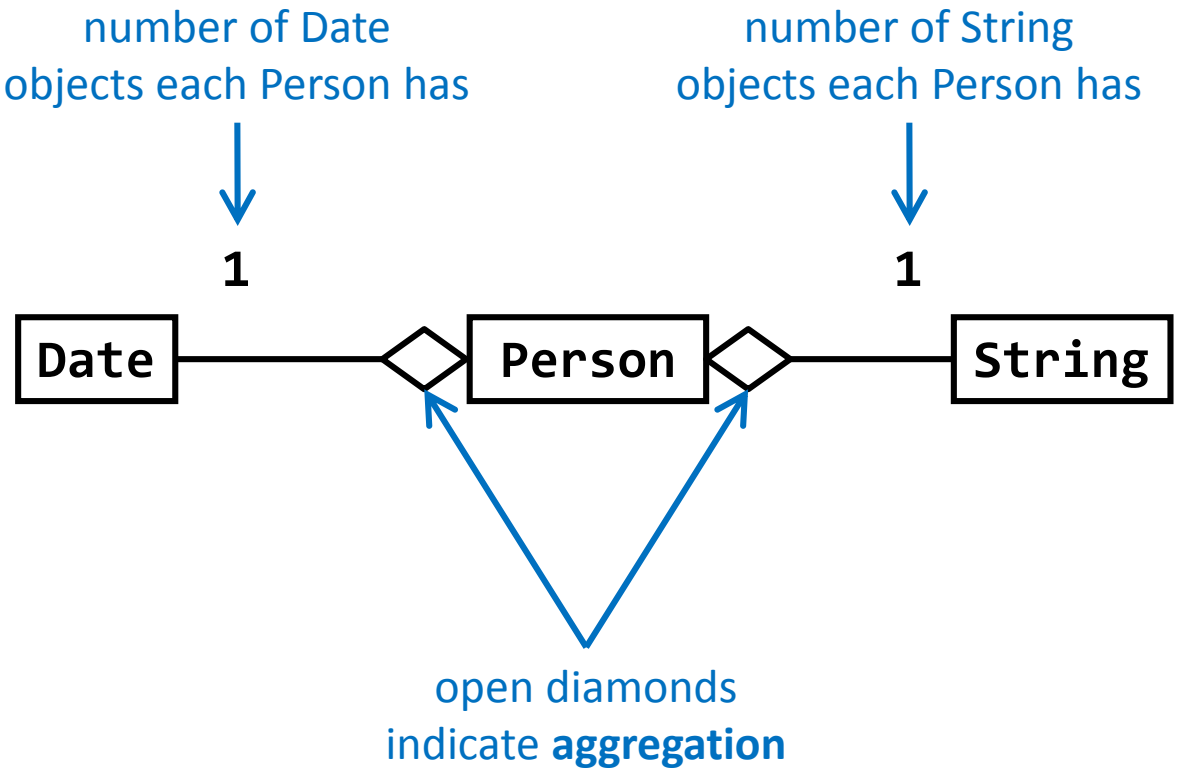
Composition implies ownership

- if the university disappears then all of its departments disappear
- a university is a composition of departments

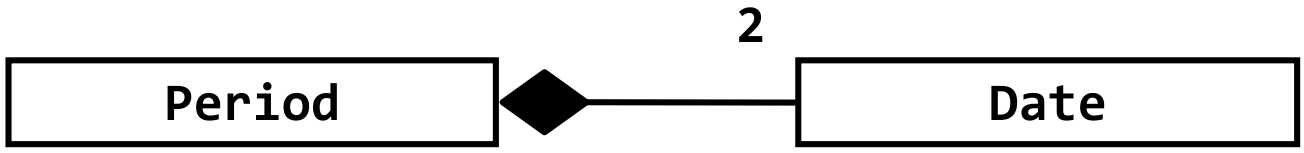
Aggregation does *not* imply ownership

- if a department disappears then the professors do not disappear
- a department is an aggregation of professors

UML Class Diagram for Aggregation



Period Class: Composition



`Period` is a **composition** of two **`Date`** objects

Collections as Fields

when using a collection as an field of a class **X** you need to decide on ownership issues

- does **X** own or share its collection?
- if **X** owns the collection, does **X** own the objects held *in* the collection?
  - *Shallow* vs. *deep* copy

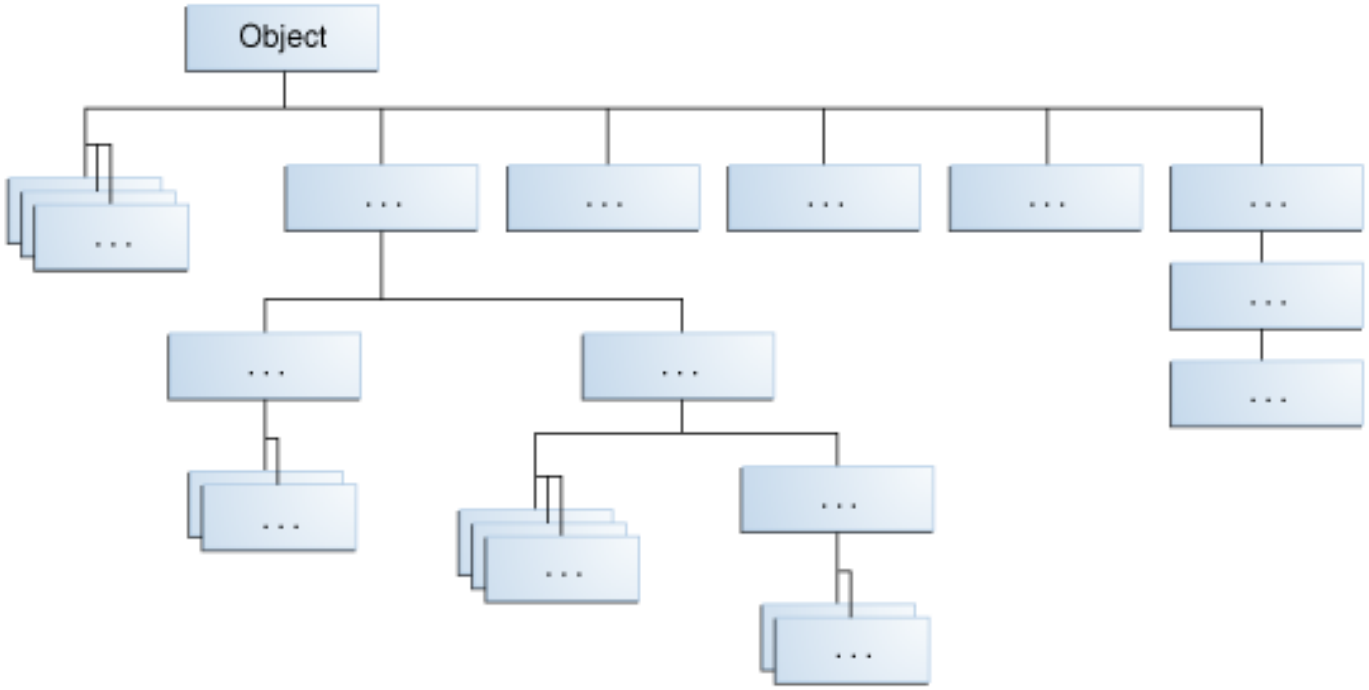
Inheritance in Java

inheritance is a relationship between two classes where one class is *derived from* another class

> *"The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself. "*

https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

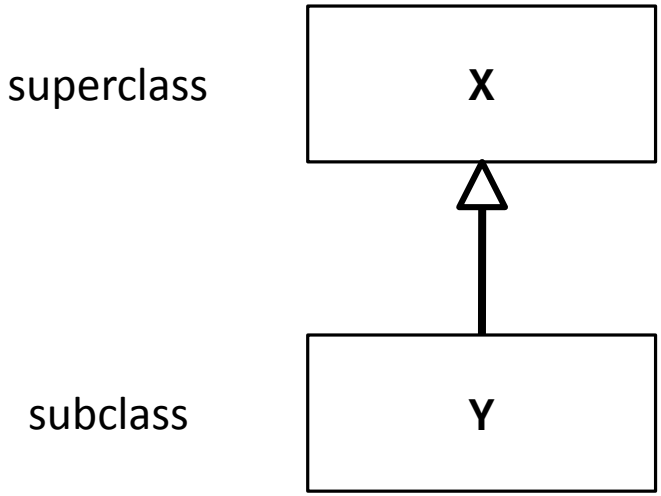
Inheritance in Java



Inheritance in Java

subclass (or derived class, **extend**ed class, child class)
a class that is derived from another class

superclass (or base class, parent class)
the class from which a subclass is derived



Inheritance in Java

in Java, the class **`java.lang.Object`** is unique

- it is *the only* class that has no superclass
- it is the class from which all other classes are descended from

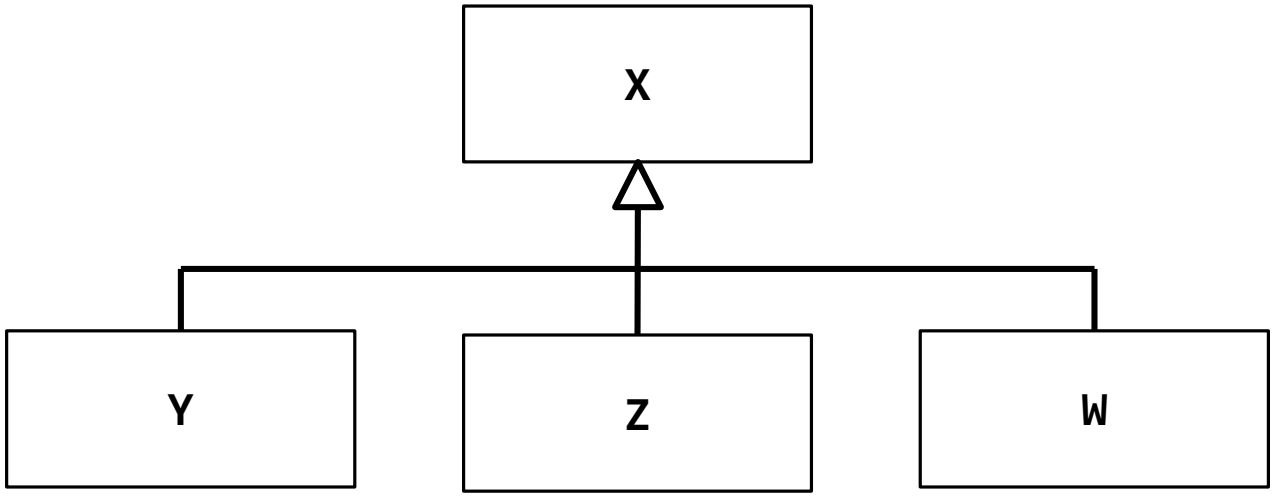
if you create a new class and do not explicitly state what the superclass is then the superclass for your new class is **`java.lang.Object`**

Inheritance in Java

in Java a superclass can have many subclasses

in Java a subclass can have only **one** superclass

called *single inheritance*

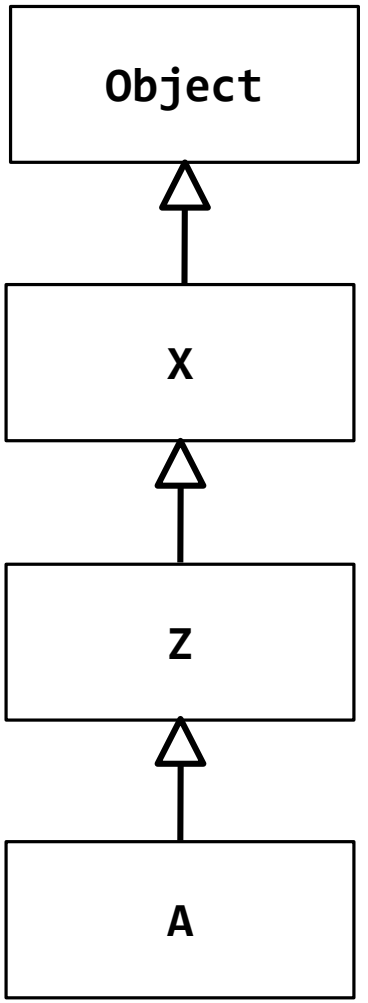


Inheritance in Java

a class can be derived from a class that is derived from a class, and so on, all the way back to **`java.lang.Object`**

a class is said to be *descended* from all of the classes in the inheritance chain going back to **`Object`**

all of the classes that a class is descended from are called the *ancestors*



Why Inheritance?

a subclass inherits all of the non-private members (fields 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

a new subclass:

- has direct access to the **`public`** and **`protected`** fields and methods without having to re-declare or re-implement them
- can introduce new fields and methods
- can re-define (override) its superclass' methods

Is-A

inheritance models the *is-a* relationship between classes

- *is-a* means *is-substitutable-for*

Is-A

from a Java point of view, **is-a** means you can use a derived class instance in place of an ancestor class instance

- for example, suppose that you have a method with one parameter of type **`Object`**

```
public SomeClass {
  public static someMethod(Object obj) {
    // does something with obj
  }
}


// client code of someMethod

String s = "hello";
SomeClass.someMethod(s);              // ok, String is-an Object

HashSet<Double> t = new HashSet<>();
SomeClass.someMethod(t);              // ok, HashSet is-an Object
```

Is-A Pitfalls

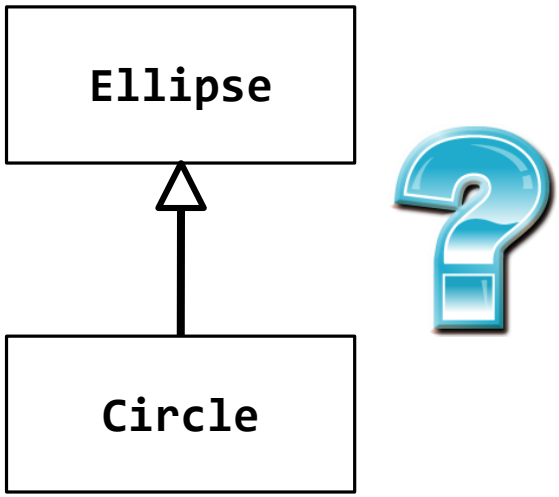
is-a has nothing to do with the real world

is-a has everything to do with how the implementer has modelled the inheritance hierarchy

the classic example:

`Circle` is-a **`Ellipse`**?

In English: “Circle is just a *special kind* of an ellipse”



`Circle` is-a `Ellipse`?

Mathematically, a circle is a kind of ellipse

but if **`Ellipse`** can do something that **`Circle`** cannot, then **`Circle`** is-a **`Ellipse`** is false for the purposes of inheritance

- 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

/**
 * Changes the width and height of the ellipse to the
 * specified width and height.
 *
 * @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;
}
```

`Circle` is-a `Ellipse`?

what if there is no **`setSize`** method?

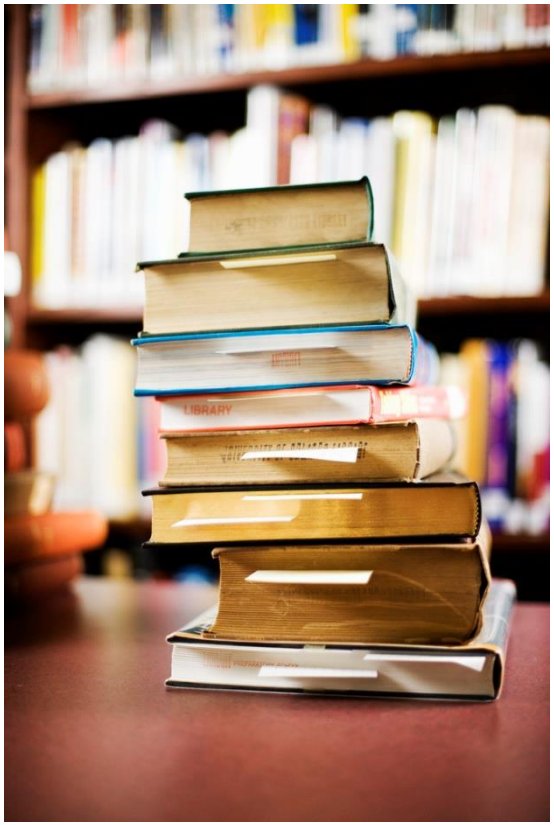
if a **`Circle`** can do everything an **`Ellipse`** can do then **`Circle`** *can* be derived from **`Ellipse`**

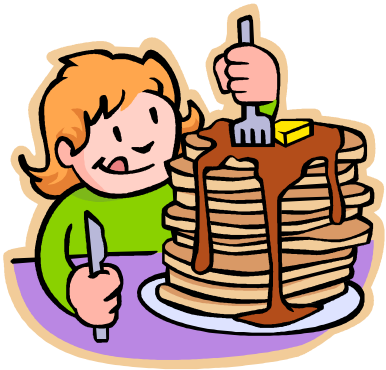
A Naïve Inheritance Example

- 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

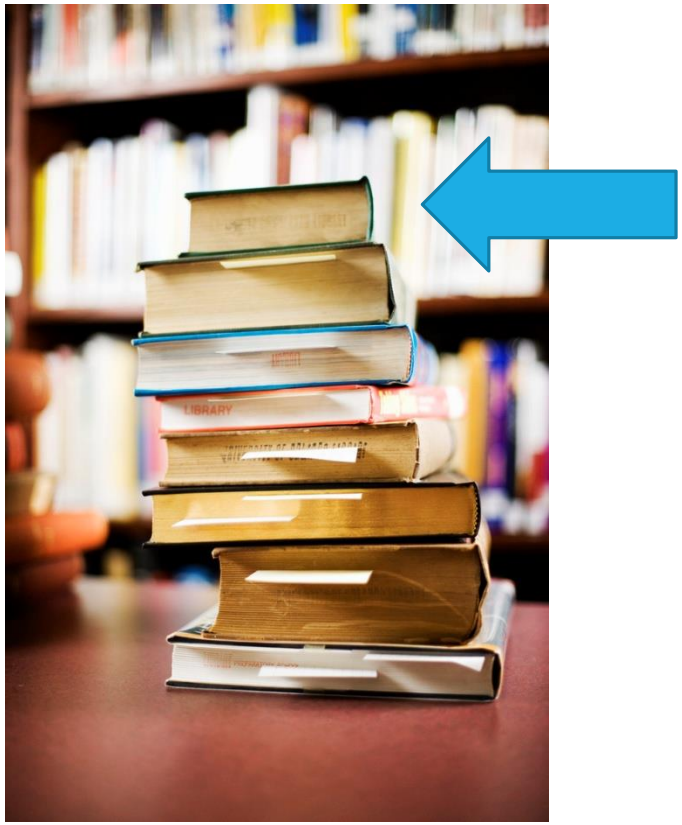
examples of stacks





Top of Stack

top of the stack



Stack Operations

classically, stacks only support two operations

1. push

   add to the top of the stack

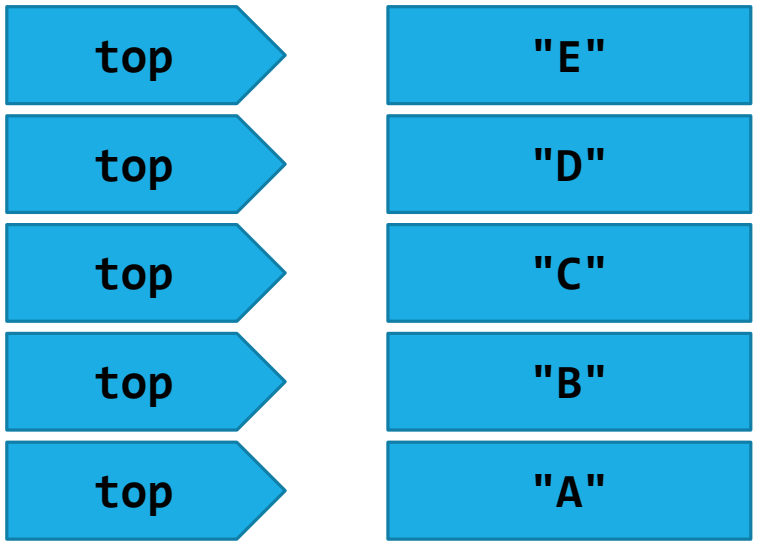
2. pop

   remove from the top of the stack

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

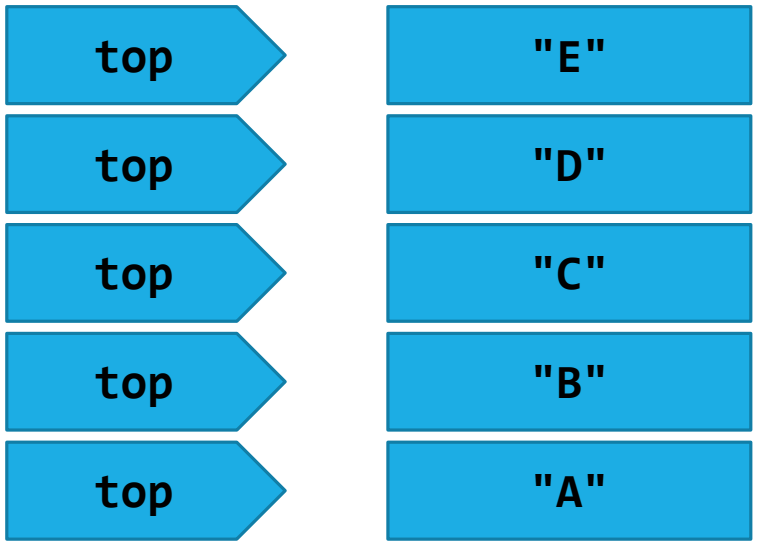
Push

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
  - remove any element 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: Problem

```
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`

Other ways to implement stack

use composition

`Stack` has-a **`List`**

the end of the list is the top of the stack

- **`push`** adds an element to the end of the list
- **`pop`** removes the element at the end of the list

Implementing stack using composition

```
import java.util.ArrayList;
import java.util.List;

/**
 * A better stack implementation that uses composition instead
 * of inheritance.
 *
 */
public class BetterStack {

   private List<Integer> elems;
```

Implementing stack using composition

```
/**
 * Initializes an empty stack.
 */
public BetterStack() {
   this.elems = new ArrayList<Integer>();
}
```

Implementing stack using composition

```
    /**
     * Pushes a value onto the top of the stack.
     *
     * @param value the value to push onto the stack
     */
    public void push(int value) {
        this.elems.add(value);
    }

    /**
     * Pops a value from the top of the stack.
     *
     * @return the value that was popped from the stack
     */
    public int pop() {
        int last = this.elems.remove(this.elems.size() - 1);
        return last;
    }
}
```

Code Reuse

Recipe

Inheritance and Code Reuse

`Counter` example from earlier in the course:

- starts counting from zero
- a user can ask for the counter's value (get)
- a user can advance the counter upwards by one (increment)
  - when the counter reaches **`Integer.MAX_VALUE`** advancing the counter causes the counter to wrap around to zero

what if you want some *other* behavior when the counter reaches **`Integer.MAX_VALUE`**?

Inheritance for code reuse: basics

`protected` access modifier allows subclasses to access a field defined in a superclass

We should also **modify the contract** of the advance method to indicate that *subclasses might change the behavior of the method*

A call to another constructor can only occur on the first line in the body of a **constructor**

A class that throws an **exception** while the superclass didn't is not a good idea (e.g., when counter value reaches **`Integer.MAX_VALUE`**)

```java
public class Counter {

  private int value;

  /**
   * Initializes this counter so that its current value is 0.
   */
  public Counter() {
    this.value = 0;
  }

  /**
   * Returns the current value of this counter.
   *
   * @return the current value of this counter
   */
  public int value() {
    return this.value;
  }
```

```
/**
 * Increment the value of this counter upwards by 1. If this
 * method is called when the current value of this counter is
 * equal to {@code Integer.MAX_VALUE} then the value of this
 * counter is set to 0 (i.e., the counter wraps around to 0).
 */
public void advance() {
  if (this.value != Integer.MAX_VALUE) {
    this.value++;
  }
  else {
    this.value = 0;
  }
}
```

```
/**
 * Increment the value of this counter upwards by 1. If this
 * method is called when the current value of this counter is
 * equal to {@code Integer.MAX_VALUE} then the value of this
 * counter is set to 0 (i.e., the counter wraps around to 0)
 * but subclasses can override this behaviour.
 */
public void advance() {
  if (this.value != Integer.MAX_VALUE) {
    this.value++;
  }
  else {
    this.value = 0;
  }
}
```

Inheritance for code reuse: contracts

why should we modify the contract of **`advance`**?

- because subclasses are supposed to be substitutable for the superclass
- if we do not modify the contract then there is no way for subclasses to change the behavior of **`advance`** and still be *substitutable*

Inheritance for code reuse: contracts

now we can extend **`Counter`** to implement different behavior when the counter value reaches **`Integer.MAX_VALUE`**

- for example we can create a counter that *stops* counting when its value reaches **`Integer.MAX_VALUE`**

Inheritance for code reuse: access

other behavior when the counter reaches `Integer.MAX_VALUE`?

if we extend the **`Counter`** class we can override the behavior of the **`advance`** method

- but this does not solve the problem because the field **`value`** in **`Counter`** is private and there are no methods that mutate the value to a specified value
  - our subclass has no way to modify the value of the counter ☹
  - this is a common problem when trying to extend a class that was not designed for inheritance

Inheritance for code reuse: access

to extend the **`Counter`** class we have to make the field **`value`** accessible to subclasses

`protected` access modifier allows subclasses to access a field defined in a superclass

while we are making changes we will add a second constructor that initializes the counter to a specified non-negative value instead of always initializing the counter to zero

```
public class Counter {
  protected int value;

  /**
   * Initializes this counter so that its current value is 0.
   */
  public Counter() {
    this.value = 0;
  }

  /**
   * Initializes this counter to the specified non-negative value.
   *
   * @param value the starting value of this counter
   * @throws IllegalArgumentException if value is negative
   */
  public Counter(int value) {
    if (value < 0) {
      throw new IllegalArgumentException();
    }
    this.value = value;
  }
```

```
public class StoppingCounter extends Counter {

  // no fields!
```

Inheritance for code reuse: fields

- the **`StoppingCounter`** class has no fields of its own
  - the current value of a **`StoppingCounter`** is stored in a field that belongs to the superclass
    - there is no need for **`StoppingCounter`** to add a new field to store the current value
- a subclass is allowed to add new fields
  - these fields are not visible to the superclass

```
public class StoppingCounter extends Counter {

  /**
   * Initializes this counter so that its current value is 0.
   */
  public StoppingCounter() {
    // what goes here?
  }
```

Constructors of Subclasses

the purpose of a constructor is to initialize the value of the fields of **`this`** object

how can a constructor set the value of a field that belongs to the superclass?

- by calling the superclass constructor and passing **`this`** as an implicit argument
  - works even if the superclass field is **`private`**

```
public class StoppingCounter extends Counter {

  /**
   * Initializes this counter so that its current value is 0.
   */
  public StoppingCounter() {
    super();
  }
```

Constructors of Subclasses

1. the first line in the body of every constructor ***must*** be a call to another constructor

   if it is not then Java will insert a call to the superclass default constructor

   if the superclass default constructor does not exist or is private then a compilation error occurs

2. a call to another constructor can only occur on the first line in the body of a constructor

3. a superclass constructor must be called during construction of the derived class

   any superclass constructor can be called (not just the no-argument constructor)

Inheritance for code reuse: constructors

we can add a second constructor that initializes the counter to a specified non-negative value

```java
/**
 * Initializes this counter to the specified non-negative value.
 *
 * @param value
 *            the starting value of this counter
 * @throws IllegalArgumentException
 *             if value is negative
 */
public StoppingCounter(int value) {
  // what goes here?
}
```

Inheritance for code reuse: constructors

note that you can use constructor chaining in the subclass

- the first line of a constructor should be another constructor call but it does not have to be a call to a superclass constructor
  - all that is required is that a superclass constructor is eventually called

for example, the two constructors of **`StoppingCounter`** could be correctly implemented as follows

```
public class StoppingCounter extends Counter {

  /**
   * Initializes this counter so that its current value is 0.
   */
  public StoppingCounter() {
    this(0);  // calls the constructor below (chaining)
  }

 /**
   * Initializes this counter to the specified non-negative value.
   *
   * @param value
   *            the starting value of this counter
   * @throws IllegalArgumentException
   *             if value is negative
   */
  public StoppingCounter(int value) {
    super(value);  // calls the superclass constructor
  }
```

What is a Subclass?

a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and fields

inheritance does more than copy the API of the superclass

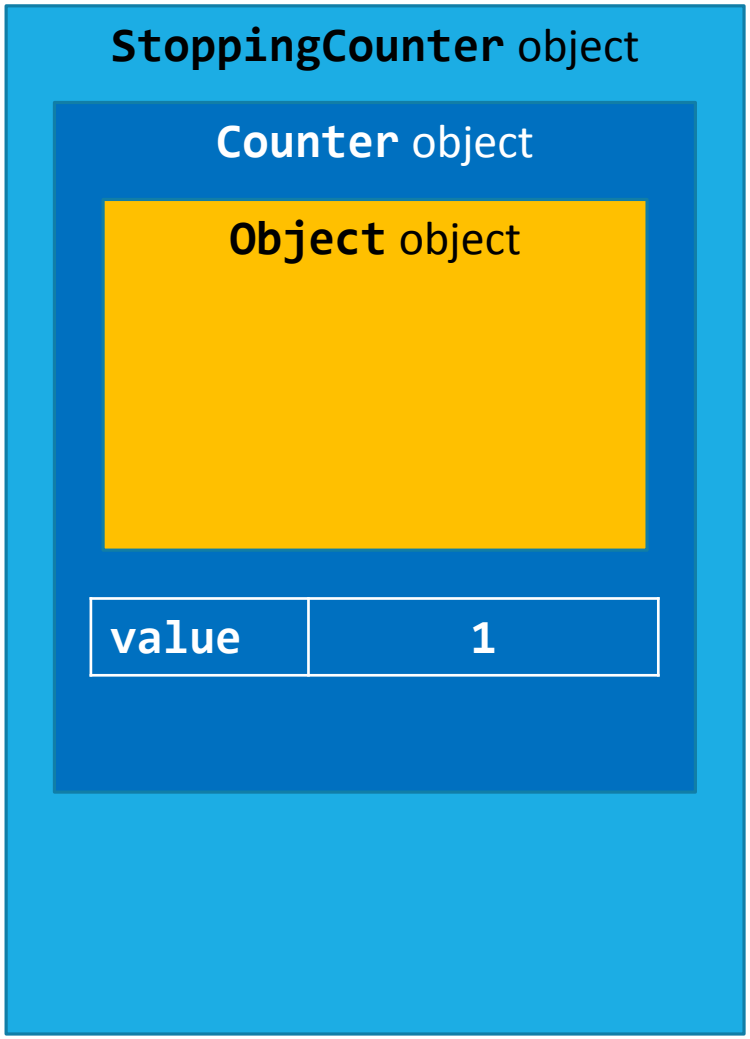
- the subclass contains a subobject of the parent class
- the superclass subobject needs to be constructed (just like a regular object)
  - the mechanism to perform the construction of the superclass subobject is to call the superclass constructor

why is the constructor call to the superclass needed?

- because **`StoppingCounter`** is-a **`Counter`** and the **`Counter`** part of **`StoppingCounter`** needs to be constructed

```
StoppingCounter c =
    new StoppingCounter(1);
```

1. **StoppingCounter** constructor starts running
- initializes new **Counter** subobject by invoking the **Counter** constructor
    2. **Counter** constructor starts running
    - initializes new **Object** subobject by (silently) invoking the **Object** constructor
        3. **Object** constructor runs
        - and finishes
    - sets **value**
    - and finishes
- finishes



Inheritance for code reuse: overriding

we can now complete the **`StoppingCounter`** implementation by overriding **`advance`**

```java
/**
 * Increment the value of this counter upwards by 1. If this method is
 * called when the current value of this counter is equal to
 * {@code Integer.MAX_VALUE} then the value of this counter remains
 * {@code Integer.MAX_VALUE} (i.e., the counter stops counting
 * at {@code Integer.MAX_VALUE}).
 */
@Override
public void advance() {
  if (this.value != Integer.MAX_VALUE) {
    this.value++;
  }
}
```

Inheritance for code reuse: overriding and exceptions

suppose that we want a counter that throws an exception when its value reaches **`Integer.MAX_VALUE`**

- it turns out that this is not a good idea
  - revisit this when we discuss how postconditions interact with inheritance

```java
package lectures.simpleinheritance;

public class ThrowingCounter extends Counter {

  // no fields!
```

```java
/**
 * Initializes this counter so that its current value is 0.
 */
public ThrowingCounter() {
  super(0);
}

/**
 * Initializes this counter to the specified non-negative value.
 *
 * @param value
 *            the starting value of this counter
 * @throws IllegalArgumentException
 *             if value is negative
 */
public ThrowingCounter(int value) {
  super(value);
}
```

```
  /**
   * Increment the value of this counter upwards by 1. If this method is
   * called when the current value of this counter is equal to
   * {@code Integer.MAX_VALUE} then a {@code RuntimeException} is thrown.
   *
   * @throws RuntimeException
   *             if this method is called when the counter is at its
   *             maximum value
   */
  @Override
  public void advance() {
    if (this.value != Integer.MAX_VALUE) {
      this.value++;
    } else {
      throw new RuntimeException();
    }
  }
}
```

Alternative to inheritance

notice that the counters differ only in what happens when the counter is advanced past its maximum value

- using inheritance, we created separate classes that override the **advance** method

an alternative approach is to encapsulate the behavior of what happens when the counter is advanced past its maximum value in an object (similar to what we did with a stack; but no is-a substitutability anymore)