

A black and white photograph of a hammer and several nails on a wooden surface. The hammer is positioned diagonally across the frame, with its head on the left and handle extending towards the top right. The hammer head is dark and metallic, showing signs of use. The handle is made of light-colored wood with a visible grain. Several nails of different sizes are scattered on the wooden surface in the foreground and background. The background is a plain, light-colored surface.

Designing for Inheritance Ch6

If all you have is a hammer,
everything looks like a nail.
-- Abraham Masslow, 1966

Topics

- 1) What makes inheritance useful?
- 2) What makes inheritance problematic?

Ex: Java Stack Inherits from Vector

- Java 1.0 had Stack is-a Vector
- What's good about its inheritance?

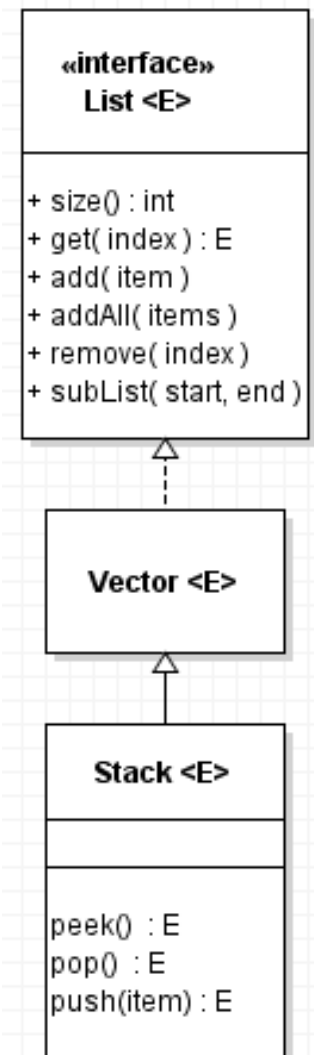
good:

- supports polymorphism with vector/list
- does not need to re-implement storage, size(),...

- What's bad about its inheritance?

bad:

- stack has list functions for direct access, add/remove away from top
- violates command/query



Encapsulation Goal

- Encapsulation goal with Inheritance:
each class manages its own state
 - use super in constructors and for overridden methods.
 - use visibility modifiers to provide sufficient access but maintain encapsulation.
 - avoid protected: fields should be private except for a “protected interface” to derived classes
- But, inheritance is not great for encapsulation (more later).

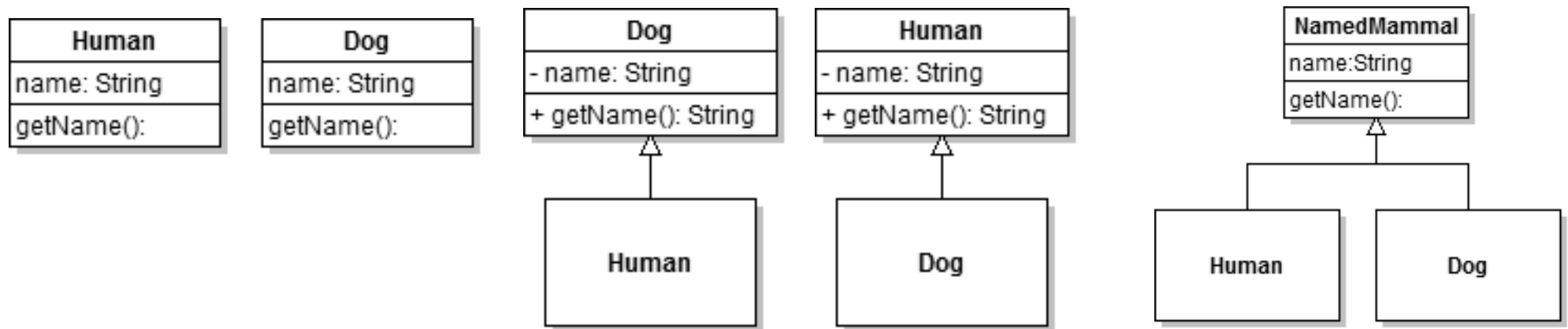
When to use Inheritance?

- What is sufficient grounds to use inheritance?
 - Code reuse?
 - Is-a relationship?
 - Polymorphism?

should question ourselves: is this good enough?

Reason 1: Code Reuse

- Idea: Inherit shared functionality from a base class.



- Human & Dog have duplicate code (fields & methods), but *neither “is-a” the other*

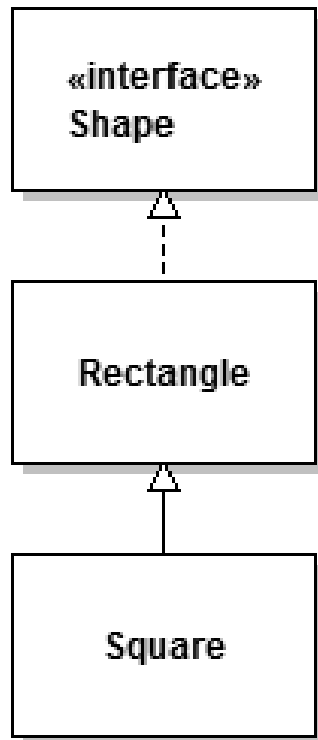
- Limitation

code reuse does not justify inheritance

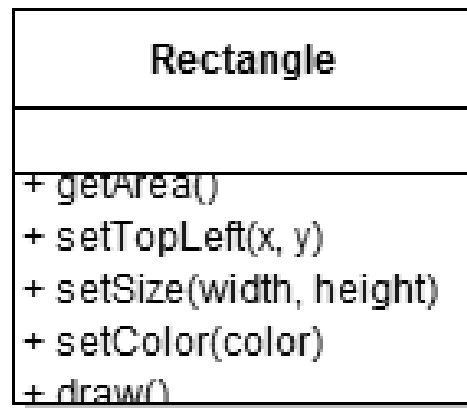
- (Could create a “NamedMammal” base-class)

Reason 2: Is-A

- Idea: Inheritance represents a [sub-set relationship](#)
- Example:



- Square is-a Rectangle, and gives reuse.
- But [sub-class is inconsistent with behaviour of super-class](#)



What is an example method in Rectangle inconsistent with Square?

- How can we describe this problem?

Is-A: LSP

- **Liskov Substitution Principle (LSP)**

B can inherit from A only if..

for each method in A, B's method:

1) **accepts** all parameters

that A's method accepts (or more) and

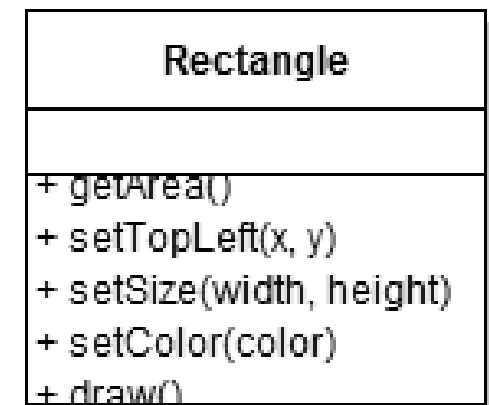
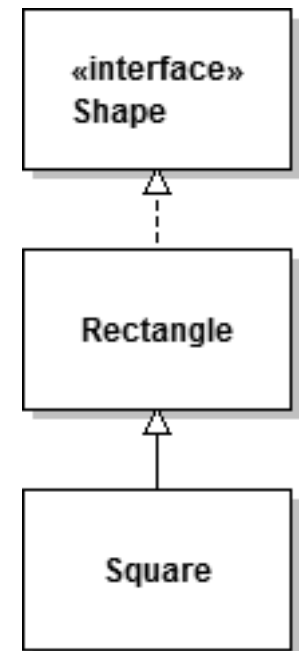
2) **does everything** with those values

that A's method does (or more).

- What methods in Rectangle fail LSP for Square?

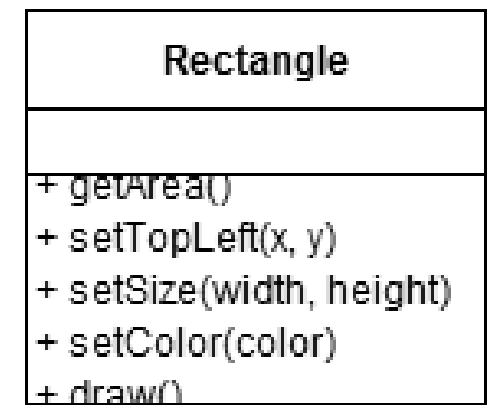
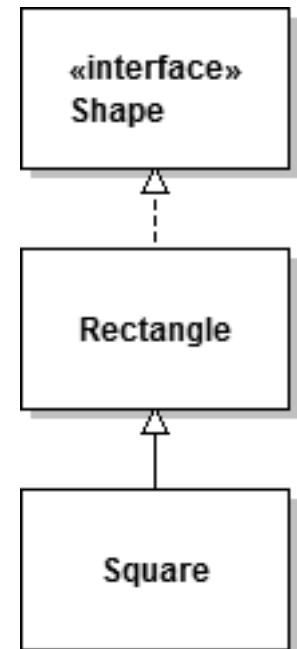
- setSize(): a Square cannot have its width and height changed independently

- Square does not do the same things with all values as Rectangle: fails LSP.



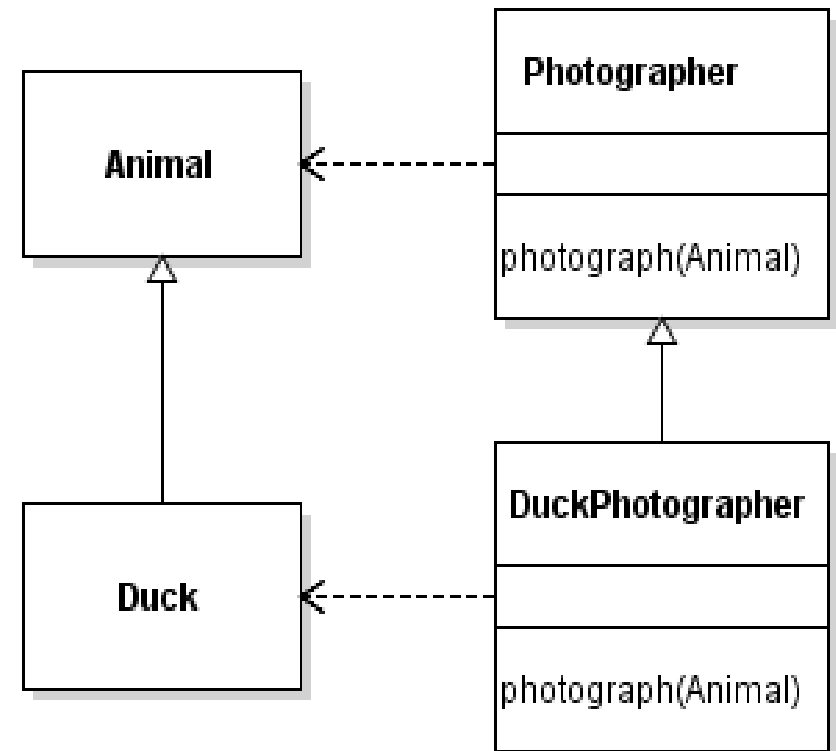
Is-A: LSP & Immutable

- LSP & Immutable
 - Would making Rectangle and Square immutable help?
 - yes, it removes the methods causing a problem
- Is-A Limitation: Must. *satisfy LSP*
(in order for inheritance to make sense)



Is-A LSP: Example

- Photographer can photograph any Animal.
DuckPhotographer only wants to photograph Ducks.
- DuckPhotographer::photograph()
wants to reject non-ducks
 - Could throw an
IllegalArgumentException?
- DuckPhotographer
violates LSP:
 - does not handle the full set of
objects that the base class does



Is-A LSP

- Rephrase LSP:
 - Client code using a reference to the base class must be able to..
use the derived class without ever knowing it
 - i.e., behaviour is unchanged.



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Is-A LSP: SOLID

LSP is part of a common set of 5 OOD principles:

- S SRP Single Responsibility Principle
“Class has one responsibility”
- O OCP Open Closed Principle
“Be open for extension, closed for modification”
- L LSP Liskov Substitution Principle
“Subtype objects interchangeable with base objects”
client can change between them, never know, never care
- I ISP Interface Segregation Principle
“Favour many client specific interfaces”
- D DIP Dependency Inversion Principle
“Depend on abstractions, not concrete classes”

Reason 3: Polymorphism

- Idea: Work with derived classes through..
variables (references) of type base-class
- Client code can flexibly work with new derived types without needing to change
 - Open-Closed Design Principle:
Code is open for reuse, but closed for modification.
- Example: New TextBox inherit Rectangle
 - Share code: code to draw border
 - Is-a: TextBox is-a Rectangle
 - Polymorphism: add to a PictureBox
- But, is that enough?

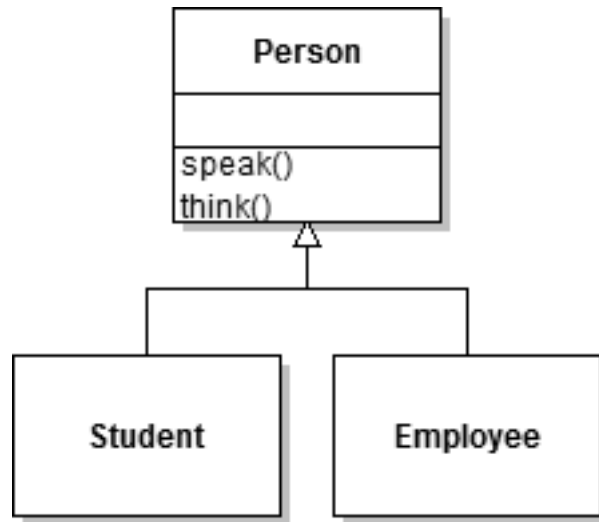
Limits of Inheritance



Pixabay [Pexel]

Inflexible type

- Example



- What about when a student..
becomes a TA?

- Cannot..
dynamically change object type

- Limitation

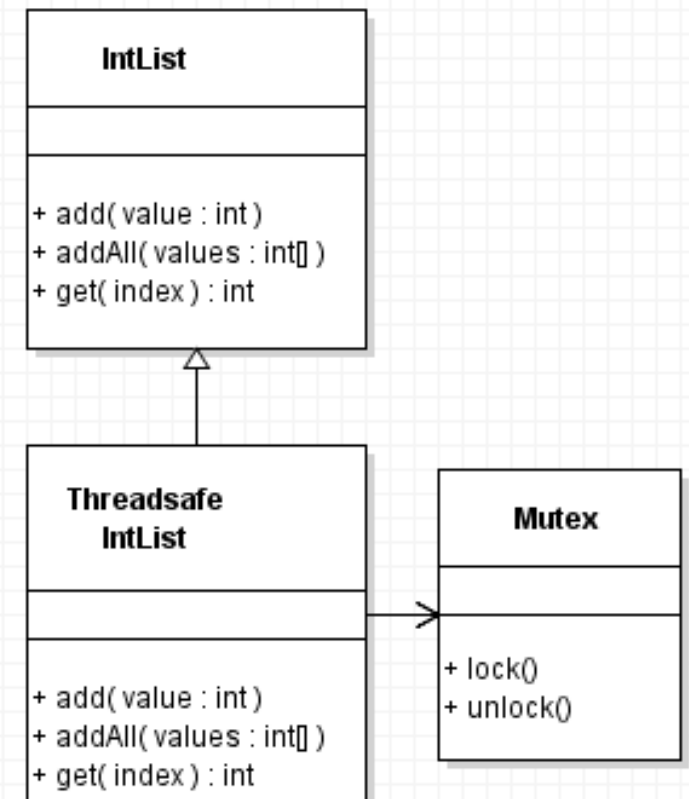
- inheritance is rigid

- Don't use inheritance for anything that may change type
 - Use composition (references) vs inheritance

Encapsulation

- Consider using inheritance to modify the behaviour of a class to make a threadsafe variety
 - Derived class can override each method of base class
 - Add a lock() and unlock() to each method
- What's good?
 - Code reuse
 - Polymorphism
- What's bad?

[see below](#)



```
class ThreadsafeIntList extends IntList {
    private Mutex mutex = new Mutex();

    @Override
    void add(int value) {
        mutex.lock();
        super.add(value);
        mutex.unlock();
    } ...
}
```


IntList Problems

can't call bcuz add has a lock
(which still be locked by addAll function below)
-> problem: self-use

```
class IntList {
    private int[] data = new int[0];

    void add(int value) {
        int newSize = data.length + 1;
        int[] big = new int[newSize];
        IntStream.range(0, data.length)
            .forEach(i -> big[i] = data[i]);
        big[newSize - 1] = value;
        data = big;
    }

    void addAll(int[] values) {
        for (int value : values) {
            add(value);
        }
    }

    int get(int index) {
        return data[index];
    }
}
```

Self Use:
- addAll() calls add()

```
class ThreadsafeIntList extends IntList {
    private Mutex mutex = new Mutex();

    @Override
    void add(int value) {
        mutex.lock();
        super.add(value);
        mutex.unlock();
    }

    @Override
    void addAll(int[] values) {
        mutex.lock();
        super.addAll(values);
        mutex.unlock();
    }

    @Override
    int get(int i) {
        mutex.lock();
        int value = super.get(i);
        mutex.unlock();
        return value;
    }
}
```

start here

Should addAll() call lock() / unlock()?

Self Use

- Self Use

- base class calls a method which can be overridden by derived class

- Problem

- Derived class needs to know when its functions will be called so it does not try to double lock.
 - Derive class depends on the internal implementation details of the base.
 - This *breaks encapsulation*

- Solution

- Base class must either
 - *avoid self-use, document it*

```
class IntList {  
    void add(int value) {  
        ...  
    }  
  
    void addAll(int[] values) {  
        for (int value : values) {  
            add(value);  
        }  
    }  
}
```

```
class ThreadsafeIntList extends IntList {  
    @Override  
    void add(int value) {  
        mutex.lock();  
        super.add(value);  
        mutex.unlock();  
    }  
  
    @Override  
    void addAll(int[] values) {  
        mutex.lock();  
        super.addAll(values);  
        mutex.unlock();  
    }  
}
```

Self use solution

- Self use is a problem when base class calls its own methods which can be overridden
- Solutions
 - Move shared functionality to `private helper function and call it (cannot be overridden)`
 - or
 - Document any self-use (and commit to it) so derived class can account for it

```
class IntList {  
    private void addInternal(int value) {  
        ... (same as add() )  
    }  
  
    void add(int value) {  
        addInternal(value);  
    }  
  
    void addAll(int[] values) {  
        for (int value : values) {  
            addInternal(value);  
        }  
    }  
}
```

```
class ThreadsafeIntList extends IntList {  
    @Override  
    void add(int value) {  
        mutex.lock();  
        super.add(value);  
        mutex.unlock();  
    }  
  
    @Override  
    void addAll(int[] values) {  
        mutex.lock();  
        super.addAll(values);  
        mutex.unlock();  
    }  
}
```

Limits of Inheritance

- inheritance is rigid
 - Cannot change object type after instantiation
- inheritance breaks encapsulation
 - Self-use must be avoided or documented
- inheritance tightly couples derived class to base class
 - Local change to base class has non-local effects
 - Adding method to base class adds behaviour to derived class:
 - may break guarantees of derived class.
 - may unexpectedly override a derived class's extra function, changing its behaviour.
 - may not compile if added function would override a derived class's extra function but different return type.

Better Inheritance

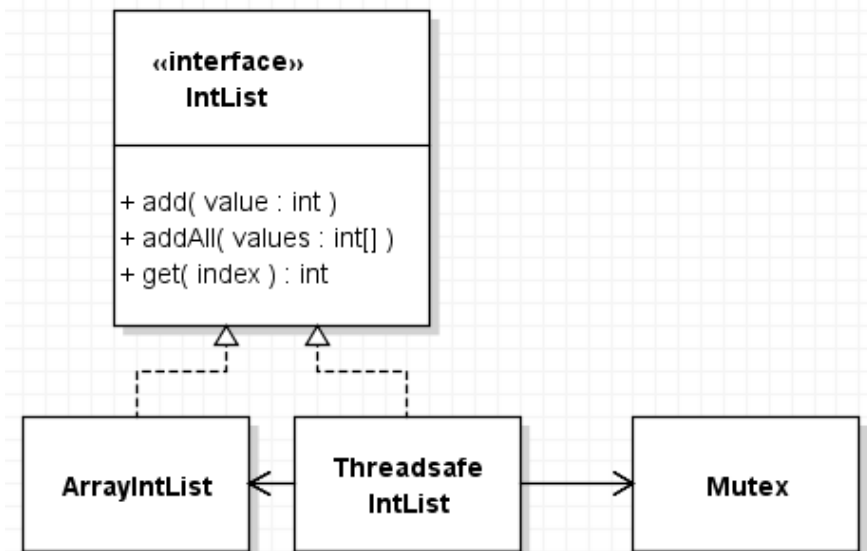
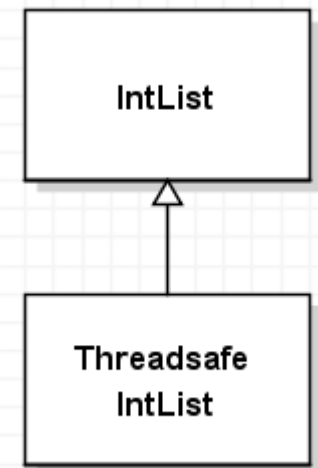
Polymorphism using Interfaces

- Implementation Inheritance
 - ..inheriting from a concrete class to reuse its code
 - *Problematic!*
- Interface Inheritance
 - Implementing an interface to support polymorphism
 - *Very useful!*
- Basic Plan
 - When needing polymorphism, use composition:
..have your client code depend on interfaces
 - Have small classes which implement the interfaces.
 - Flexibly compose objects at runtime
 - Flexibly add new small objects

Replace Inheritance with Wrapper

- Instead of inheriting from concrete class `IntList`, have “derived” class holds a reference to it.
 - `ArrayIntList` implements the `IntList` interface
 - `Wrapper Object`
Hold a reference to a concrete `IntList`
 - `forwarding`
Each derived method calls the wrapped object
 - Forwarding is also called `delegation`

```
class ThreadSafeIntList implements IntList {  
    private Mutex mutex = new Mutex();  
    private IntList list = new ArrayIntList();  
  
    @Override  
    void add(int value) {  
        mutex.lock();  
        list.add(value);  
        mutex.unlock();  
    } ...  
}
```



IntList with Wrapper Class

```
interface IntList {  
    void add(int value);  
    void addAll(int[] values);  
    int get(int index);  
}
```

```
final class ArrayIntList implements IntList {  
    private int[] data = new int[0];  
  
    @Override  
    public void add(int value) {  
        ...  
    }  
  
    @Override  
    public void addAll(int[] values) {  
        for (int value : values) {  
            add(value);  
        }  
    }  
  
    @Override  
    public int get(int index) {  
        return data[index];  
    }  
}
```

```
final class ThreadsafeIntList implements IntList {  
    private Mutex mutex = new Mutex();  
    private IntList list = new ArrayIntList();
```

```
    @Override  
    public void add(int value) {  
        mutex.lock();  
        list.add(value);  
        mutex.unlock();  
    }  
    delegate: call list to do the work
```

Could use
DI

```
    @Override  
    public void addAll(int[] values) {  
        mutex.lock();  
        list.addAll(values);  
        mutex.unlock();  
    }
```

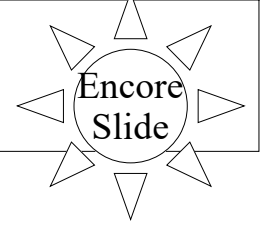
```
    @Override  
    public int get(int i) {  
        mutex.lock();  
        int value = list.get(i);  
        mutex.unlock();  
        return value;  
    }
```

ThreadsafeIntList:
- Is-a IntList and
- Has-a IntList
It is the [Decorator Pattern](#)

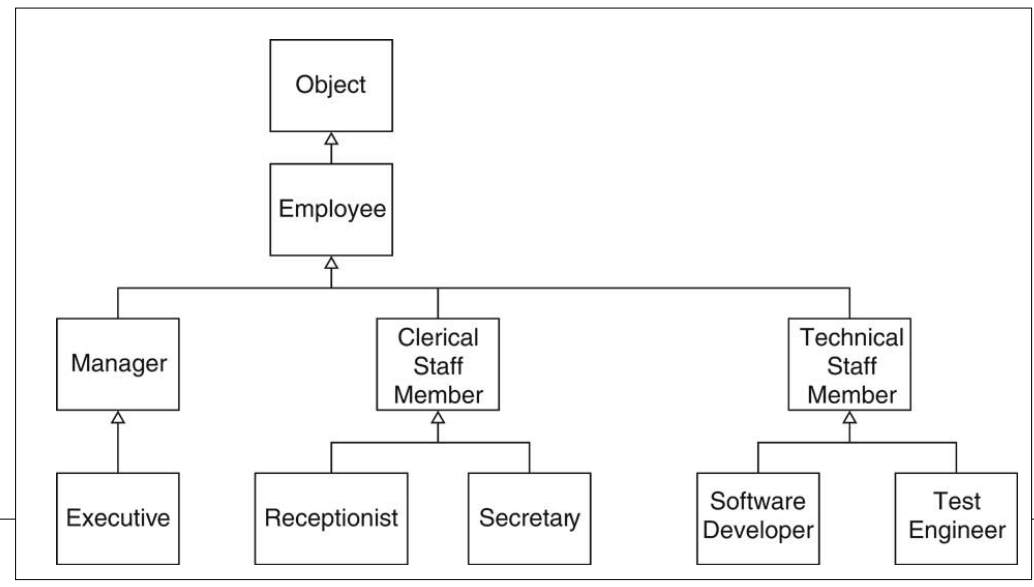
24-03-19

inheriting something or implementing an interface,
at the same time, holding a reference to it

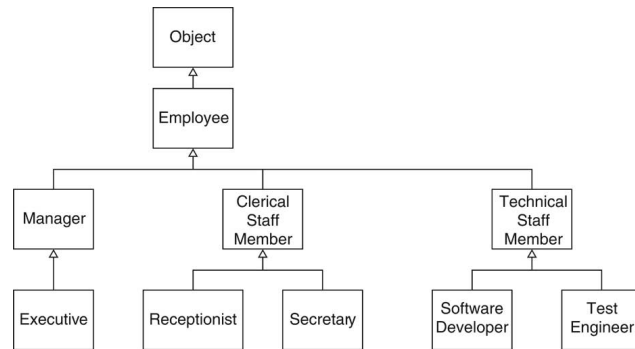
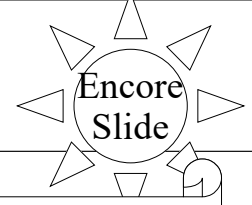
Replacing Implementation Inheritance



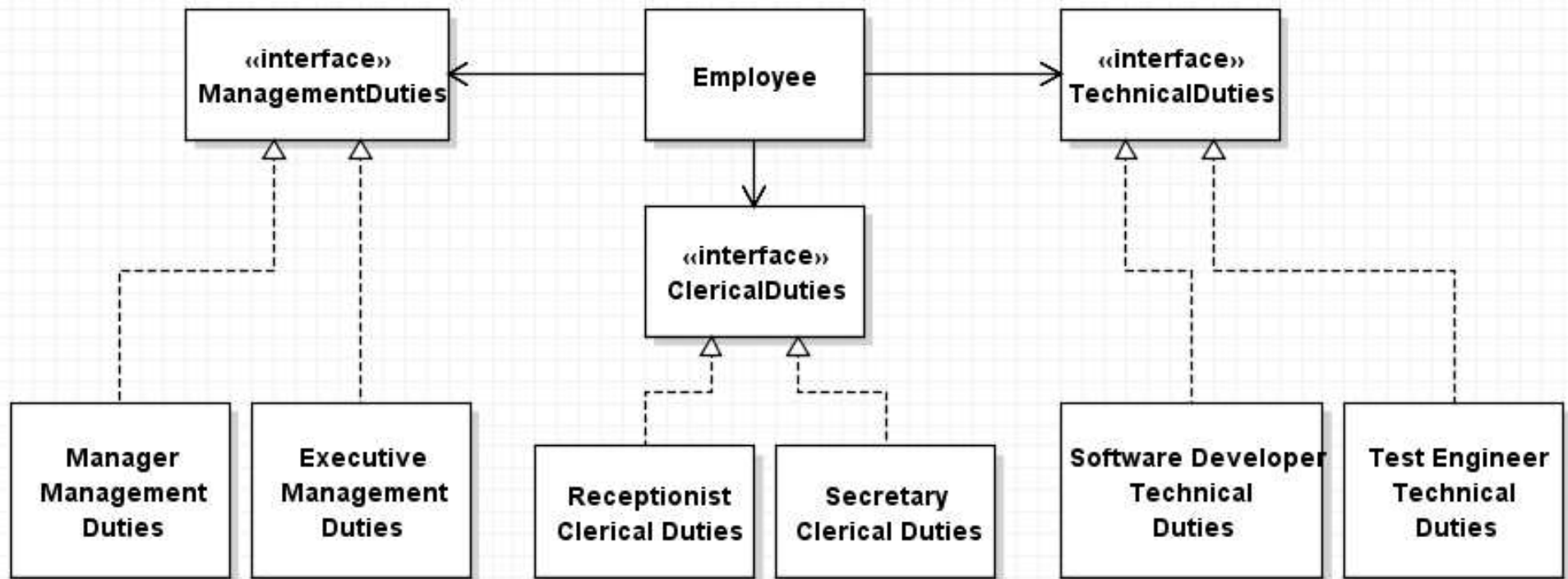
- Inheritance hierarchies of concrete classes are bad
 - Rigid types for all objects
 - Reuse of implementation (code) means many dependencies on super classes.
- Ex: Add a “senior” role to Manager and Clerical Staff:
 - Senior feature:
 - Get more money
 - Can sign for credit card
 - Not clear how to fit into inheritance hierarchy

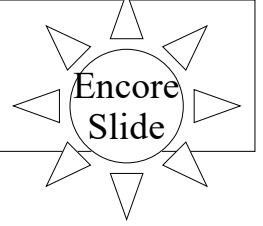


Use Composition Instead



Composition is more flexible.
Can change object an runtime.
Can have multiple duties.





- **Design Principle:**

Program to an interface, not an implementation

- Flexibility to reference a different concrete class later

- **Design Principle:**

Prefer composition over inheritance

- Composition allows *runtime flexibility to change* (reference a new object)
- Reduces rigid coupling from static inheritance hierarchy

we still use (interface) inheritance, but just bring in an extra layer of composition to isolate from negative effects of inheritance's features

Summary

- Use inheritance only when supported by:
 - is-a relationship & LSP
 - polymorphism
- Limits on Inheritance
 - Good to “Inherit” (implement) interfaces!
 - OK to inherit from classes you control (same package)
 - OK to inherit from classes designed for inheritance (Ex: “Template Pattern”)
 - Only when you are OK living with base class’s API
- Consider using composition instead (as well).