

CS 3

# Introduction to Software Engineering

7: Type Hierarchy

Chapter 7

# Questions?

# Abstractions and Implementations

Recall “complex number” example/demo:

- Complex numbers as data abstraction.
- Gave two implementations:
  - CartesianComplex: represent number by real, imaginary parts.
  - PolarComplex: represent by magnitude and angle.

Implement same operations, but differently.

- Demo program drew Mandelbrot set.
  - Two versions, one using each Complex implementation.

# Lessons

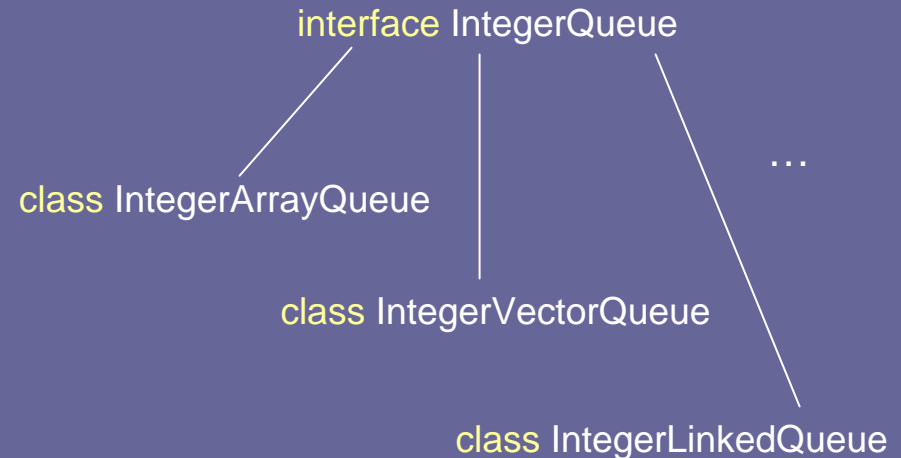
- Intent of example:
  - An abstraction can have  $> 1$  possible implementation.
  - Class hides implementation details from client code.
  - Implementation can be changed without affecting client.

# Lessons

- Intent of example:
  - An abstraction can have  $> 1$  possible implementation.
  - Class hides implementation details from client code.
  - Implementation can be changed without affecting client.
- Looking closer...
  - Two Complex representations have different performance.
    - So client code *might* care about that implementation “detail”.
    - May want different implementations for different purposes.
    - Another example: Array vs. linked list for stack/queue.
  - Switching between implementations required copying/pasting/searching/replacing!
    - Want *the same client* to be able to use *any implementation*.
  - Type hierarchies provide some tools to help attack these issues.

# Approach

- Define a *interface* for an ADT.
  - Declares and specifies methods.
- Each implementation of the ADT is a *class*.
  - Defines representation.
  - Implements methods.
- In client code:
  - *Create* objects using a specific implementation.
  - *Refer to* objects by interface where implementation doesn't matter.

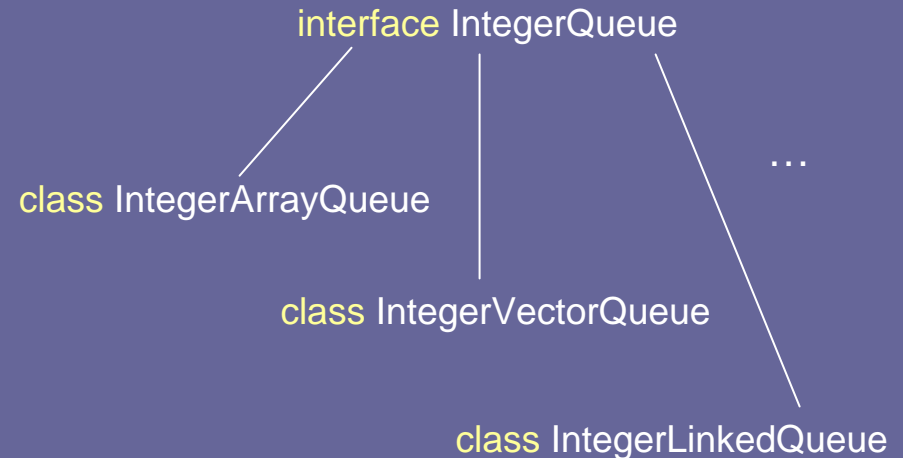


```
/* Once queue is created, its implementation
   doesn't matter. */
IntegerQueue q = new IntegerArrayQueue();
```

# Interface and Class

```
public interface IntegerQueue {  
    void put(int i);  
    int get();  
}
```

```
public class IntegerArrayQueue  
    implements IntegerQueue  
{  
    private int[] a;  
    private int first, last;  
    public IntegerArrayQueue() {  
        a = new int[SIZE];  
        first = last = 0;  
    }  
    public void put(int i) { ... }  
    public int get() { ... }  
}
```



```
/* Once queue is created, its implementation  
   doesn't matter. */
```

```
IntegerQueue q = new IntegerArrayQueue();
```

# Interface and Class

```
public interface IntegerQueue {  
    void put(int i);  
    int get();  
}
```

```
public class IntegerArrayQueue  
    implements IntegerQueue  
{  
    private int[] a;  
    private int first,last;  
    public IntegerArrayQueue() {  
        a = new int[SIZE];  
        first = last = 0;  
    }  
    public void put(int i) { ... }  
    public int get() { ... }  
}
```

- Interface defines the abstraction.
  - Detailed method specs go here.\*
  - For implementors:  
“What it takes” to be a queue.
  - For clients:  
What a queue can do for you.
- Class provides implementation.
  - Representation.
  - Method implementations.
  - Constructor(s).
  - Need specifications for:
    - Abstraction function.
    - Representation invariant.
    - Constructor usage.
    - Methods not in interface.

\*(but are omitted from the slide to save space)

# Interfaces are Types

```
IntegerQueue q = new IntegerArrayQueue();
```

- IntegerArrayQueue is a *subtype* of IntegerQueue.
  - Think “subset”:  
 $\{\text{IntegerArrayQueue objects}\} \subseteq \{\text{IntegerQueue objects}\}$
  - Every IntegerArrayQueue **is an** IntegerQueue
  - $\{\text{IntegerQueue objects}\} =$   
 $\{\text{IntegerArrayQueue objects}\} \cup \{\text{IntegerVectorQueue objects}\} \cup$   
...
- If someone hands you an IntegerQueue, it might be an IntegerArrayQueue, or something else.
  - Might come from a class that didn't exist when your code was written.
  - Might never be the same class twice!

# Types Tell You Things

```
void processInts(IntegerQueue q) {...}
```

- Parameter type is IntegerQueue.
- Procedure is telling you,  
“Give me [something that acts like] a queue of integers.”
- It’s *not* specifying how the queue should be implemented.
  - For that, it would use a type like IntegerArrayQueue.
- Procedure expecting IntegerQueue is **more flexible** than one requiring IntegerArrayQueue.

# Types Tell You Things

```
IntegerQueue setupQueue(int[] a) { ... }
```

- Return type is IntegerQueue.
  - This type says,
    - “Give me an array of integers, and I’ll give you [something that acts like] a queue of integers.”
- Without committing to a particular queue implementation.
- Using interface type gives **procedure** freedom to choose implementation.
  - Caller cannot depend on getting a particular type of queue.

# Types Tell You Things

```
IntegerQueue q = new IntegerArrayQueue();
```

- Declaration says,
  - “After creating this object, I will no longer care what class generated it.”
- In other words, as far as **subsequent code** is concerned, any kind of IntegerQueue will do.
  - All that matters is that q **supports the IntegerQueue methods**.
- More flexible than declaring q at type IntegerArrayQueue.
  - Prevents accidental implementation dependency in subsequent use of q.
  - To use a different implementation, just change constructor call.

# In Real Life

- Java Collections library. (In `java.util`.)
- Interfaces like `Collection`, `List`, `Set`.
- Classes like `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`.
  
- Good practice:
  - Use interface types to declare variables, parameters, return types.
  - Use classes only to create new collections.

# Other Uses of Hierarchy

- Subtyping relation creates “disjoint union” types:
  - $\text{IntegerQueue} = \text{IntegerArrayQueue} \cup \text{IntegerVectorQueue} \cup \dots$
- In some cases, convenient to think of interface + all implementing classes *together* as defining *one type*.
  - interface **ChessPiece** { ... }
  - class **Pawn** implements **ChessPiece** { ... }
  - class **Rook** implements **ChessPiece** { ... }
  - class **Bishop** implements **ChessPiece** { ... }
  - **ChessPiece** = **Pawn**  $\cup$  **Rook**  $\cup$  **Bishop**  $\cup$  ...
  - ChessPiece is the “main idea.”
  - Most parameters, variables, *etc.* in program will have type ChessPiece, not Rook or Pawn.

# Abstract Base Classes

- Disjoint union example from book:  
 $\text{IntList} = \text{EmptyIntList} \cup \text{FullIntList}$

```
public abstract class IntList {  
    // IntLists are immutable lists of Objects.  
    // A typical IntList is a sequence [x1,...,xn].  
  
    public abstract Object first()  
        throws EmptyException;  
    public abstract IntList rest()  
        throws EmptyException;  
  
    ...  
}
```

```
public class EmptyIntList extends IntList {  
    ...  
    public Object first() throws EmptyException{  
        throw new EmptyException(...);  
    }  
    ...  
}
```

```
public class FullIntList extends IntList {  
    private final Object val; // Not final in book – why not?  
    private final IntList next; // Not final in book – why not?  
    public Object first() { return val; }  
    ...  
}
```

# Abstract Base Classes

- Disjoint union example from book:  
 $\text{IntList} = \text{EmptyIntList} \cup \text{FullIntList}$
- IntList is an **abstract class**.
- Abstract = can't create objects; therefore, useless w/o **concrete subclasses**.
- Abstract class can have some fields, implement some methods.
  - Should be things all (or most) “variants” of type have in common.
  - If nothing in common, should probably use interface instead.
- Can also declare methods that subclasses must implement; mark these **abstract**.

# Interfaces versus Abstract Classes

- Liskov seems to favor abstract base class over interface.
- I have the opposite view:
  - Using abstract class constrains future implementations.
  - Implementation inheritance is complicated.
- Compromise possible:
  - Use interface as supertype to define abstraction.
  - Provide abstract base class implementations *can* inherit, if convenient.
  - Example: `java.awt.event.KeyListener` (interface) and `java.awt.event.KeyAdapter` (abstract class).

# Complex #'s Revisited

- Recall: Had to copy/paste/replace to get CartesianComplex code to use PolarComplex instead.
- Might make more flexible by defining supertype:

```
interface Complex {  
    double real();  
    double imag();  
    double mag();  
    double ang();  
    Complex add(Complex);  
    Complex mul(Complex);  
}
```

# Complex #'s Revisited

```
interface Complex {  
    double real();  
    double imag();  
    double mag();  
    double ang();  
    Complex add(Complex);  
    Complex mul(Complex);  
}
```

```
class PolarComplex implements  
    Complex {  
    private double r,th;  
    public PolarComplex(double r,  
                        double th) { ... }  
    ...  
    public Complex mul(Complex z) {  
        ???  
    }  
}
```

# Complex #'s Revisited

```
interface Complex {  
    double real();  
    double imag();  
    double mag();  
    double ang();  
    Complex add(Complex);  
    Complex mul(Complex);  
}
```

```
class PolarComplex implements  
    Complex {  
    private double r,th;  
    public PolarComplex(double r,  
                        double th) { ... }  
    ...  
    public Complex mul(Complex z) {  
        ???  
    }  
}
```

- Return PolarComplex or CartesianComplex?
- What if z is CartesianComplex?

# Related Example from Book

- Sparse and Dense Polynomials.
- Different representations have space tradeoffs; best choice depends on the polynomial being represented.
- Can end up trying to add a SparsePoly to a DensePoly.
  - How to handle this case?
  - Which class to use for the result?
    - Sparse+Sparse can be dense!
    - Dense+Dense can be sparse!
    - Dense \* Dense can be sparse!
- Question: Are SparsePoly and DensePoly two separate, complete implementations of Poly? Or are they two variants in a single disjoint union implementation?
- We'll revisit this next time (probably).

# Inheritance

```
class B extends A { ... }
```

- Central concept of object-oriented programming.
- Two things happening:
  - B is a **subtype** of A. (Every B-object is an A-object.)
  - B **inherits** A's representation:
    - All of A's fields.
    - A's method implementations, except those it **overrides**.

# Why [When] Inheritance?

Situations for using inheritance:

- Two or more classes have lots of code in common.
  - May discover this after they have been written.
    - “Factor out” common parts into a superclass.
  - Superclass should be abstract (unless there’s a good reason it shouldn’t).
    - It captures what descendants have in common, which may not be enough to stand on its own.
- An existing class does almost everything you need, but not quite.
  - Create a subclass that adds what you want.
  - Can also override or enhance existing behaviors as necessary.

# Enhancing Existing Classes

- Many class libraries expect you to program this way.
- For instance: `javax.swing.JPanel`.
  - Mostly used for grouping/layout of Swing components.
  - But also useful for creating **custom components**:
    - Override `paintComponent` to customize appearance.
    - Add fields to store custom **state**.
    - Add methods to implement special **behaviors**.
    - Example: `PlottingPanel` in complex number demo.

# Inheritance versus Composition

- Inheritance

- Every A-object **is a** B-object.
- Calling A-object's methods automatically executes B's code, unless overridden.
- Call to overridden method from inherited method executes A's version.
- A's method code sees B's **protected** internals.

- Composition

- Every A-object **has a** B-object.
- A must implement all methods, but may **delegate** actual work to internal B-object.
- Call to “non-delegated” method from delegated method runs B's version.
- B's internals hidden from A's code.
  
- No subtyping relationship – but an interface may help.

```
class B {  
    public void bmethod1() {...}  
    public void bmethod2() { bmethod1(); }  
    ...  
}
```

// Inheritance

```
class A extends B {  
    public void bmethod1() { ... }  
}
```

// Composition

```
class A {  
    private B myB;  
    public void bmethod1() { ... }  
    public void bmethod2() {  
        myB.bmethod2();  
    }  
}
```

# Inheritance versus Composition

- Inheritance
  - Every A-object **is a** B-object.
  - Calling A-object's methods automatically executes B's code, unless overridden.
  - Call to overridden method from inherited method executes A's version.
  - A's method code sees B's **protected** internals.
- Composition
  - Every A-object **has a** B-object.
  - A must implement all methods, but may **delegate** actual work to internal B-object.
  - Call to "non-delegated" method from delegated method runs B's version.
  - B's internals hidden from A's code.
  - No subtyping relationship – but an interface may help.

```
class B implements BI {  
    public void bmethod1() {...}  
    public void bmethod2() { bmethod1(); }  
    ...  
}
```

// Inheritance

```
class A extends B {  
    public void bmethod1() { ... }  
}
```

// Composition

```
class A implements BI {  
    private B myB;  
    public void bmethod1() { ... }  
    public void bmethod2() {  
        myB.bmethod2();  
    }  
}
```

# Inheritance versus Composition

- Object-oriented programming advocates like the ability to model **is-a** relationships.
- Usual advice: model **has-a** relationships with composition, **is-a** relationships with inheritance.

```
class B implements BI {
    public void bmethod1() {...}
    public void bmethod2() { bmethod1(); }
    ...
}

// Inheritance
class A extends B {
    public void bmethod1() { ... }
}

// Composition
class A implements BI {
    private B myB;
    public void bmethod1() { ... }
    public void bmethod2() {
        myB.bmethod2();
    }
}
```

# Inheritance versus Composition

- Object-oriented programming advocates like the ability to model **is-a** relationships.
- Usual advice: model **has-a** relationships with composition, **is-a** relationships with inheritance.
- I say, **Be Careful!**  
The fact that every X “is” a Y in real life does not mean that the data structure representing an X should also represent a Y.
- Inheritance is useful only if you really want every X-object to be a Y-object.
- To make this decision, must look at theory of subtyping more closely.

```
class B implements BI {
    public void bmethod1() {...}
    public void bmethod2() { bmethod1(); }
    ...
}

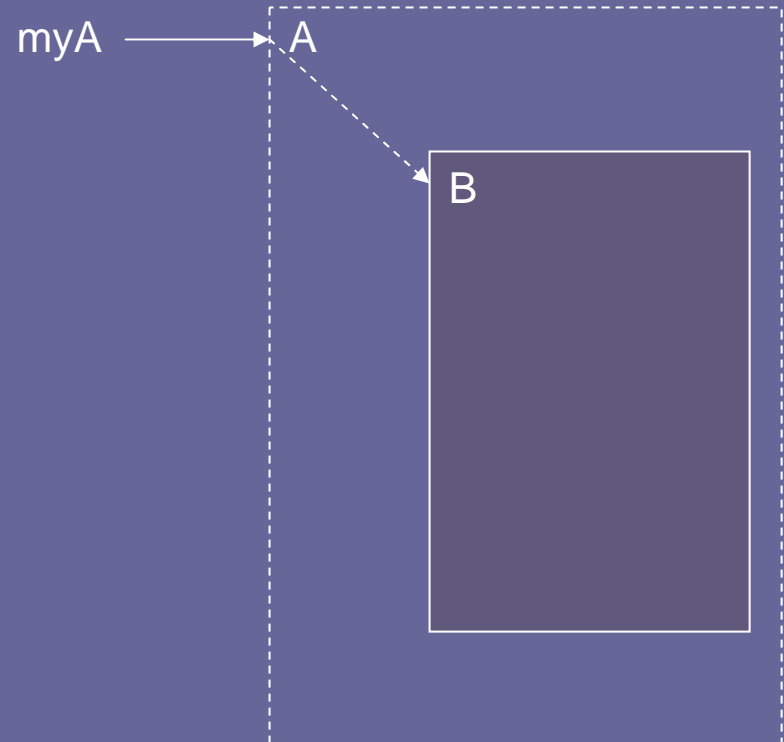
// Inheritance
class A extends B {
    public void bmethod1() { ... }
}

// Composition
class A implements BI {
    private B myB;
    public void bmethod1() { ... }
    public void bmethod2() {
        myB.bmethod2();
    }
}
```

# Subtyping as “Masquerading”

- Language considers every B-object to be an A-object.
- In other words, a B can “pretend” to be an A.

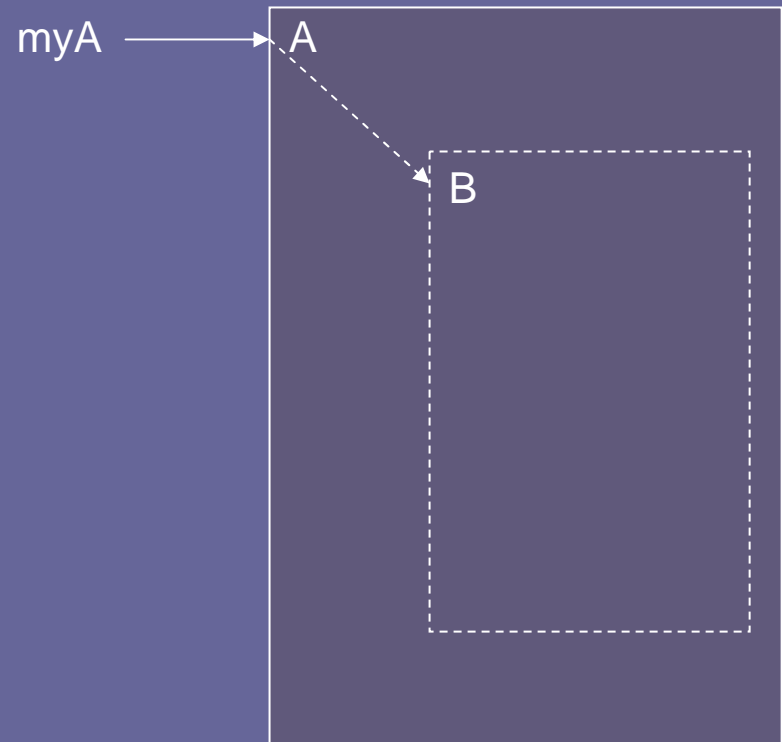
```
class B extends A {...}  
A myA = new B();
```



# Subtyping as “Masquerading”

- Language considers every B-object to be an A-object.
- In other words, a B can “pretend” to be an A.

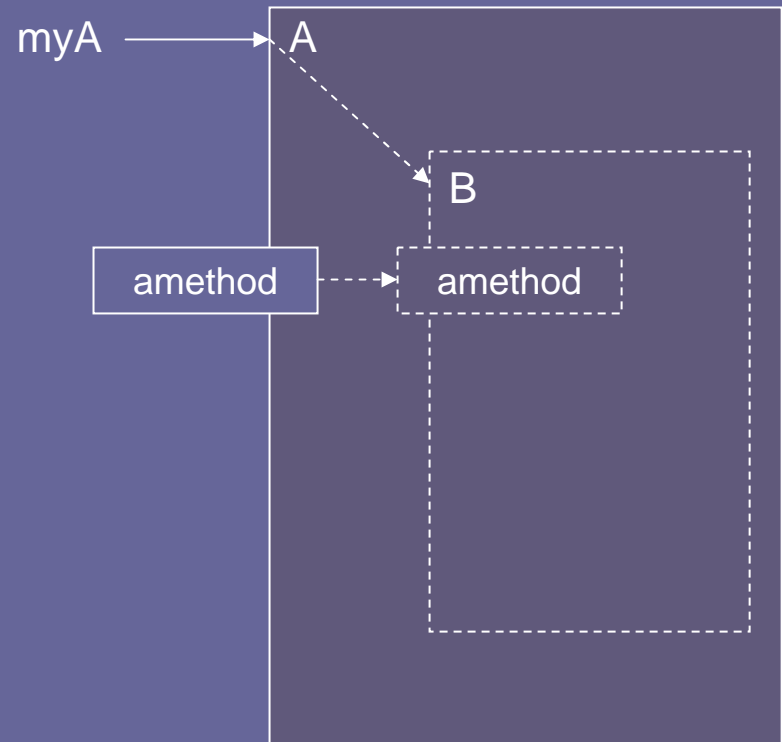
```
class B extends A {...}  
A myA = new B();
```



# Subtyping as “Masquerading”

- Language considers every B-object to be an A-object.
- In other words, a B can “pretend” to be an A.
- To maintain illusion, must be able to handle amethod.
  - B’s amethod must be able to handle x if A’s can.
  - (Return type must also be consistent.)
  - The Signature Rule – enforced by language/compiler.

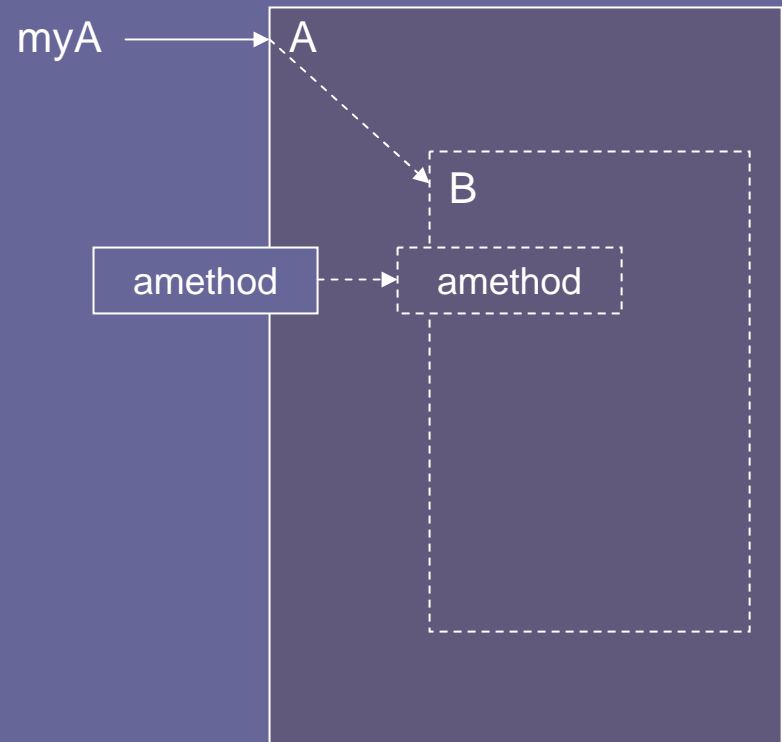
```
class B extends A {...}  
A myA = new B();  
myA.amethod(x);
```



# Subtyping as “Masquerading”

- Compiler can only enforce rules to do with **types**.
- Not enough to guarantee effective masquerading.
- What about the non-type part of a method specification?

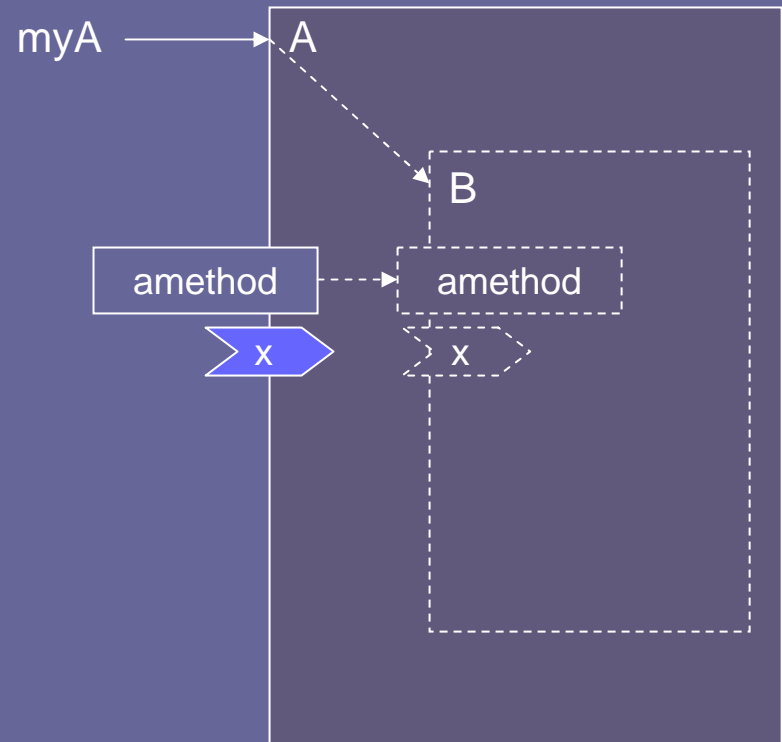
```
class B extends A {...}  
A myA = new B();  
myA.amethod(x);
```



# Subtyping as “Masquerading”

- Compiler can only enforce rules to do with **types**.
- Not enough to guarantee effective masquerading.
- Precondition Rule:  
If B.amethod has any requirements, A.amethod must have all the same requirements, maybe more.

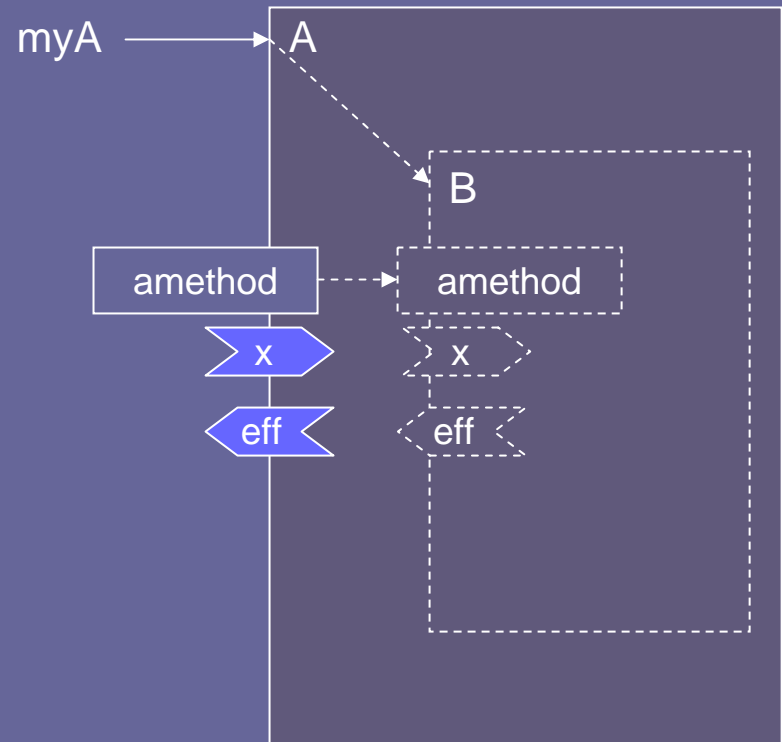
```
class B extends A {...}  
A myA = new B();  
myA.amethod(x);
```



# Subtyping as “Masquerading”

- Compiler can only enforce rules to do with **types**.
- Not enough to guarantee effective masquerading.
- Precondition Rule:  
If B.amethod has any requirements, A.amethod must have all the same requirements, maybe more.
- Postcondition Rule:  
B.amethod must have all the effects of A.amethod, perhaps more.

```
class B extends A {...}  
A myA = new B();  
myA.amethod(x);
```



# Properties Rule

- Paraphrase: subtype objects should never behave in a way that “surprises” supertype clients.
- Clients are allowed to try to “predict” object behavior based on specs.
  - Subtype objects must always obey supertype contracts.
  - Subtype objects must never change state in a way a supertype object can not.

# Counterintuitive Results

- Specialization does not (always) create subtypes.
  - A SushiChef **is a** Chef who can only accept SushiOrders?
  - A NonemptyList **is a** List that you can only remove from if it has size 2 or more?
  - A ShrinkingList **is a** List whose add method does nothing?
- Adding new capabilities does not (always) create subtypes.
  - A MutableMatrix **is a** Matrix supporting update operations?
  - A NetworkedAlgorithm **is an** Algorithm that can fail due to network errors?
- Most of these cases can be made to work if you anticipate the subtypes when defining the supertypes.
  - *E.g.*, Java’s “unmodifiable” collections.  
Collection interface explicitly states mutating operations may throw UnsupportedOperationException.

# Never Forget Subtyping!

- If you ask for an object of an extensible class, you never know what you'll get.
- If your class is extensible, and you call a non-final non-private method, you never know what will happen.
  - If you rely on information hiding for security purposes, watch out for rogue subclasses!
- Use “final” whenever possible.
  - Guarantees clients will get behavior from your class, not a random subclass.
  - Safeguard your reputation against impostors!