CSE 331

Type Polymorphism

James Wilcox and Kevin Zatloukal

Type Polymorphism

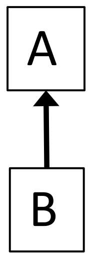
- Last design topic will be "**type polymorphism**"
  - allows code to work correctly with more than one type

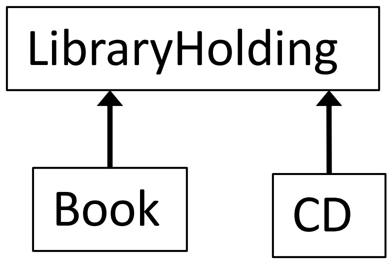
- We will look at two instances of this
  - **subtypes**: can be used in where supertype expected
  - **generics**: can use type with different instantiation of its type parameters

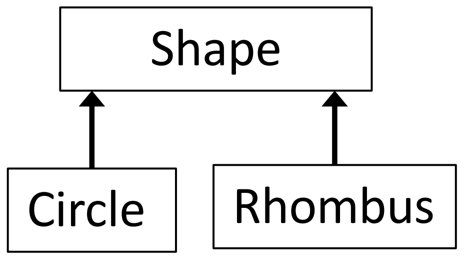
Subtypes

What Is a Subtype?

- Sometimes "every B is an A"
  - every book is a library holding
  - every circle is a shape
- Denote with an upward arrow
- Not always so clear
  - would like a formal definition







Recall: Comparing Specifications

- Would like a similar definition for specifications

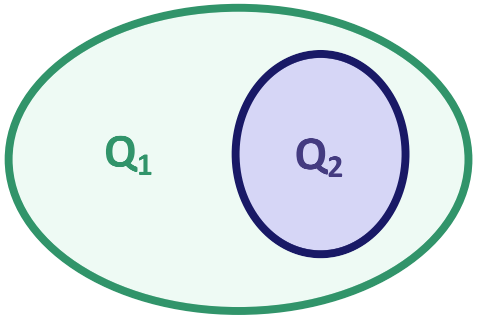
- Specification S_1 is stronger than S_2...
  - whenever is S_1 satisfied, S_2 is also satisfied
  - i.e., satisfying S_1 implies satisfying S_2

- Code written for S_2 also works with S_1

- But what does this mean?
  - specifications have a precondition and postcondition

Recall: Stronger Assertions

- Assertion is stronger iff it holds in a subset of states

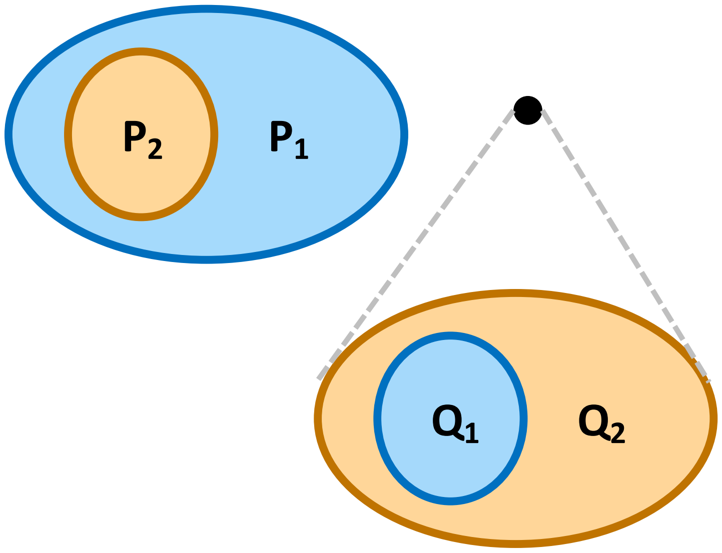


- Stronger assertion implies the weaker one
  - stronger is a synonym for “implies”
  - weaker is a synonym for “is implied by”

Recall: Comparing Function Specifications

- Specification S_1 is stronger than S_2 if it has…
  - a stronger postcondition and the same precondition
  - a weaker precondition and the same postcondition
  - (or both)

precondition

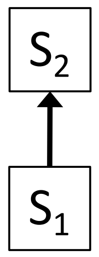


postcondition
(for a **fixed** input)

What Is a Subtype?

- Would like a similar definition for types

- ADT S_1 is a subtype (stronger) than S_2...
  - when is S_1 satisfied, S_2 is also satisfied
  - i.e., satisfying S_1 implies satisfying S_2

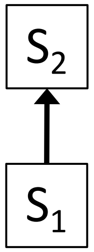


- Code written for S_2 also works with S_1

- Alternatively, subtypes are "substitutable"
  - called the "Liskov substitution principle?

What Is a Subtype?

- Would like a similar definition for types
- ADT S_1 is a subtype (stronger) than S_2…
  - when is S_1 satisfied, S_2 is also satisfied
  - i.e., satisfying S_1 implies satisfying S_2



$$\begin{tikzcd} \boxed{S_2} \\ \boxed{S_1} \arrow[u] \end{tikzcd}$$

- Code written for S_2 also works with S_1
- But what does this mean?

Comparing ADT Specifications

- ADT S_1 is a **subtype** (stronger) than S_2 if…
  - S_1 has all the methods of S_2
  - each of method of S_1 has a stronger specification than the corresponding method of S_2

    stronger or the same specification

- For example:

foo exists with same spec
bar exists with stronger spec

```
interface A {
  int foo(String s);
  Object bar();
}
```

```
interface B {
  int foo(String s);
  String bar();
  int baz();
}
```

okay to have an extra method

`B` is a subtype but Java will **not** let you substitute

Comparing ADT Specifications

- ADT S_1 is a subtype (stronger) than S_2 if…
  - S_1 has all the methods of S_2
  - each of method of S_1 has a stronger specification than the corresponding method of S_2

    stronger or the same specification

- For example:

In order to pass a B where an A is expected, Java requires B to **extend** A

```
interface A {
  int foo(String s);
  Object bar();
}
```

```
interface B extends A{
  int foo(String s);
  String bar();
  int baz();
}
```

Java mistakenly equates subtypes and subclasses

Recall: Subclasses

- Subclassing is a means of sharing code
  - subclass gets parent fields & methods (unless overridden)

```
class Product {
  private String name;
  private int price;
  public String getName() {return name; }
  public int getPrice() { return price; }
}

class SaleProduct extends Product {
  private float discount;
  public int getPrice() {
    return (1 - discount) * super.getPrice();
  }
}
```

Subclassing ≠ Subtyping

- **Subclassing is a means of code sharing**
  - **all fields of the superclass**
  - **all methods of superclass copied into subclass**

    unless overridden in the subclass
  - **ensures it has all the methods of the parent class**

- **Subtyping is about specifications**
  - **each method specification must be stronger**

    weaker **precondition** and/or stronger **postcondition**
  - **mostly in the comments!**

    which the Java compiler does not read

Example 1

```
// An integer value that represents …
interface NumberA {
    // @requires obj is present in A
    // @returns an index i such that A[i] = obj
    int indexOf(int[] A);
}

interface NumberB extends NumberA {
    // @requires obj is present in A
    // @returns the smallest index i such that A[i] = obj
    int indexOf(int[] A);
}
```

Would Java notice if we swapped these?

No! Compiler doesn't read the comments.

– **can see that `NumberB` is a subtype of `NumberA`**

  `NumberB` has a stronger postcondition than `NumberA`

Example 2

```
// An integer value that represents …
interface NumberA {
    // @returns the smallest i such that A[i] = obj
    //     or -1 if obj is not present in A
    int indexOf(int[] A);
}

interface NumberB extends NumberA {
    // @requires obj is present in A
    // @returns the smallest index i such that A[i] = obj
    int indexOf(int[] A);
}
```

- **can see that `NumberB` is not a subtype of `NumberA`**
  - `NumberB` allows *fewer* inputs (stronger precondition) than `NumberA`
- **but Java allows it anyway**

Subtyping in the Type Checker

```
class NumberA { … }
class NumberB extends NumberA { … }

    public void foo(NumberA n) { … }

    NumberB m = …
    foo(m);             // Java allows this call
```

- Java allows you to pass the subclass
  - it allows substitution of subclasses
  - it assumes that subclasses are subtypes
- Subtyping shows up when you make method calls
  - both the arguments passed in and the return value

Recall: Subclasses

- Subclassing is a surprisingly dangerous feature

- Subclassing tends to break modularity
  - creates tight coupling between super- and sub-class
  - often see the "fragile base class" problem

    changes to super class often break subclasses

- New: Java assumes subclasses are subtypes
  - Java will let you pass subclass where supertype expected
  - no way for it to check that it is really a subtype!
  - code will break in strange ways if it's not true

Subtyping in the Type Checker

```
class NumberA { … }
class NumberB extends NumberA { … }

    public void foo(NumberA n) {
     … in here …
    }
```

- **In the body of the method `foo`, variable `n` will be instance of `NumberA` or a subclass (e.g. `NumberB`)**

- **It will <u>have</u> all the methods of `NumberA`**
  - **any subclass gets those methods copied into it**
  - **rules out many bugs!**

Java Does Some Checks (Return Types)

- Java checks the return types:

```
interface A {
  String foo();
}

interface B extends A {
  Object foo();   // error!
}
```

  - subclass wants to return non-`String` values
  - checks the part of the postcondition visible in the types

Java Does Some Checks (Exceptions)

- Java checks most exceptions:

```
interface A {
  String foo() throws IOException;
}

interface B extends A {
  String foo() throws Exception;  // error!
}
```

  - subclass wants to throw non-`IOException` exceptions
  - checks the part of the postcondition visible in the types
  - only applies to "checked" exceptions

    does not check `RuntimeException` or `Error`

Java Cannot Handle More Inputs

- **Java checks most exceptions:**

```
interface A {
  int foo(String s);
}

interface B extends A {
  int foo(Object s);  // doesn't work!
}
```

Java (and C++) identify methods by their signature (name + argument types)

  - this is a strengthening (more allowed inputs)

    but it will not work properly in Java…

  - Java calls this *overloading* not overriding

    `B` has two methods named "`foo`", not one

TypeScript Can Handle More Inputs

- This works properly in TypeScript

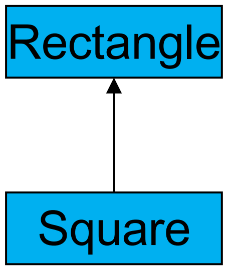
```
interface A {
  foo(s: string): number;
}

interface B extends A {
  foo(s: string | number): number;  // okay!
}
```

  - TypeScript has only one method with a given name
  - also not necessary to say "`extends`"

    TypeScript has a structural, not nominal, type system

Squares and Rectangles

- Is a Square a Rectangle?
  - yes



- Is a Square a subtype of Rectangle?
  - seems like it should be…

Squares and Rectangles

- Consider the following method of Rectangle:

```
// @modifies obj
// @effects obj.width = w and obj.height = h
void setSize(int w, int h)
```

- How do we implement this in Square?

```
// @requires w = h
// @modifies obj
// @effects obj.width = w and obj.height = h
void setSize(int w, int h)
```

This is a weakening (stronger precondition)!

Squares and Rectangles

- Consider the following method of Rectangle:

```
// @modifies obj
// @effects obj.width = w and obj.height = h
void setSize(int w, int h)
```

- How do we implement this in Square?

```
// @modifies obj
// @effects obj.width = w and obj.height = h
// @throws BadSize if w != h
void setSize(int w, int h)
```

This is an incomparable spec

Squares and Rectangles

- Consider the following method of Rectangle:

```
// @modifies obj
// @effects obj.width = w and obj.height = h
void setSize(int w, int h)
```

- How do we implement this in Square?

```
// @modifies obj
// @effects obj.width = w and obj.height = w
void setSize(int w, int h)
```

This is an incomparable spec

Squares and Rectangles

- Consider the following method of Rectangle:

```
// @modifies obj
// @effects obj.width = w and obj.height = h
void setSize(int w, int h)
```

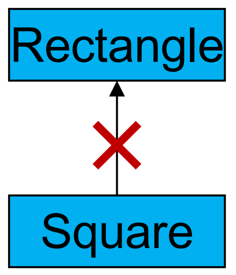
- How do we implement this in Square?

```
// @modifies obj
// @effects obj.width = obj.height = sideLength
void setSize(int sideLength)
```

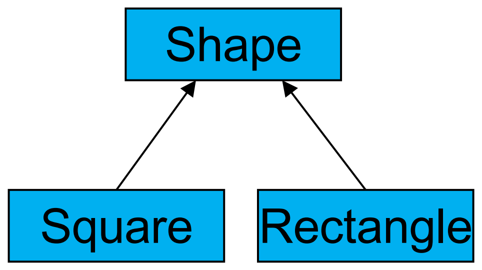
This isn't the same method
(overloading not overriding)

Squares and Rectangles

- Square is not a subtype of Rectangle
  - subtyping can be *unintuitive*

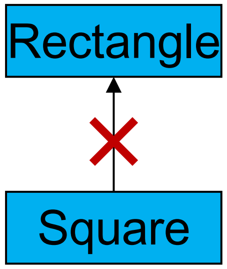


- Solution 1: make them siblings
  - common parts in the parent class `Shape`
  - cannot substitute `Square` for `Rectangle`



Squares and Rectangles

- Square is not a subtype of Rectangle
  - subtyping can be *unintuitive*



- Solution 2: make them immutable
  - problem in `setSize` because it mutates
  - would be no problems if we did not allow it

- Will see more examples of this later on...
  - reading and writing operations are different

Benefits of Immutability

- No worries about representation exposure
  - do not need to copy in & copy out
- No worries about key mutation errors
  - one of the worst bugs out there
- Subtyping usually works the way you expect
  - squares are subtypes of rectangles

Inappropriate Subtyping in the JDK

```
class Hashtable {
    public void put(Object key, Object value) { … }
    public Object get(Object key) { … }
}

class Properties extends Hashtable {
    public void setProperty(String key, String val) {
      put(key, val);
    }

    public String getProperty(String key) {
      return (String) get(key);
    }
}
```

– **can cast** `Properties` **to** `Hashtable` **(but it's not a good idea!)**

Inappropriate Subtyping in the JDK

```
class Hashtable {
    public void put(Object key, Object value) { … }
    public Object get(Object key) { … }
}

class Properties extends Hashtable {
    public void setProperty(String key, String val) {
      put(key, val);
    }

    public String getProperty(String key) {
      return (String) get(key);
    }
}
```

```
Properties p = new Properties();
Hashtable h = p;
h.put("One", 1);
p.getProperty("One");  // crash!
```

Inappropriate Subtyping in the JDK

- The documentation says not to do this:

  > "*Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. ... If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail.*"

- Problem solved?
  - no! someone will still mess this up
  - this is bad design

- What should they do?

Composition

```
class Properties {
    private Hashtable tbl;

    public void setProperty(String key, String val) {
      tbl.put(key, val);
    }

    public String getProperty(Object key) {
      return (String) tbl.get(key);
    }
}
```

- Can no longer be misused
  - has no `get` or `put` methods at all
- Solution you already know is called "composition"

Generic Types

The Many Varieties of Lists

type $\text{List} := \text{nil} \mid \text{cons}(x : \mathbb{Z},\ L : \text{List})$	list of integers
type $\text{BList} := \text{nil} \mid \text{cons}(f : \mathbb{B},\ L : \text{List})$	list of booleans
type $\text{PList} := \text{nil} \mid \text{cons}(p : \mathbb{Z} \times \mathbb{Z},\ L : \text{List})$	list of pairs

– **common pattern to all of these:**

all the same except what data we store in them

The Many Varieties of Lists

$\textbf{type}\ \mathrm{List}\ :=\ \mathrm{nil}\ \mid\ \mathrm{cons}(x : \mathbb{Z},\ L : \mathrm{List})$	list of integers
$\textbf{type}\ \mathrm{BList}\ :=\ \mathrm{nil}\ \mid\ \mathrm{cons}(f : \mathbb{B},\ L : \mathrm{List})$	list of booleans
$\textbf{type}\ \mathrm{PList}\ :=\ \mathrm{nil}\ \mid\ \mathrm{cons}(p : \mathbb{Z} \times \mathbb{Z},\ L : \mathrm{List})$	list of pairs

- **Can have one definition for all lists:**

$$\textbf{type}\ \mathrm{List}\langle \mathbf{A} \rangle\ :=\ \mathrm{nil}\ \mid\ \mathrm{cons}(x : \mathbf{A},\ L : \mathrm{List})$$

  - **those above are** $\mathrm{List}\langle\mathbb{Z}\rangle$, $\mathrm{List}\langle\mathbb{B}\rangle$, **and** $\mathrm{List}\langle\mathbb{Z} \times \mathbb{Z}\rangle$
  - **"**$\mathbf{A}$**" is a type argument**
  - **"**List**" is a function from types to types**

Java Before Generics

- Collections allowed any kind of object

```
Hashtable h = new Hashtable();
h.put("abc", new Integer(3));

Integer v = (Integer) h.get("abc");
```

  - only type checking occurs at run-time

    relying on your unit tests & code reviews to catch everything
  - programmers frequently make mistakes here

Java Generics

Declaration

```
class FastLastList<T> implements List<T> {
    // RI: last = last(this.list)
    private List<T> list;
    private T last;
}
```

Use

- "T" is an argument to the type declaration
  - also called a "type parameter"

Java Generics

```
class Name<A, B, C, D> { … }
interface Name<A, B, C, D> { … }
```

- Declarations can have any number of arguments
  - Java style is to use short names

    e.g. "E" for element, "K" for key, "V" for value

- Must fill in the arguments to use it

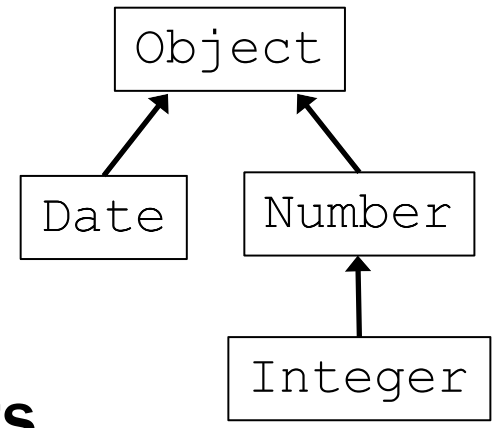
```
Name<Integer, Boolean, String, Object> n;
```

  - Java only *warns* if you leave off `<..>` but don't do it!

Type Constraints

- Type declarations constrain the values passed in

```
boolean add1(Object elt);
boolean add2(Number elt);

add1(new Date());   // okay
add2(new Date());   // error!
```

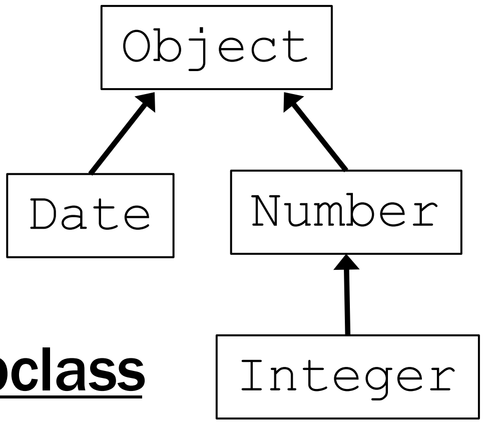


- Same idea applies to type arguments
  - here they are called "bounds" on the type

Type Bounds

- Type Bounds constrain the types passed in

```
interface List1<E extends Object> { … }
interface List2<E extends Number> { … }

List1<Date> L1;   // okay
List2<Date> L2;   // error!
```



  - these are called "upper bounds"
  - type argument can be that type or a <u>subclass</u>

Java Generics (Take Two)

```
class Name<T1 extends B1, ..., Tn extends Bn>
interface Name<T1 extends B1, ..., Tn extends Bn
```

- Declarations can have any number of arguments
  - each argument has an optional upper bound

    if not provided, it is "`Object`"

- Must fill in the arguments to use it

```
Name<Integer, Boolean, String, Object> n;
```

  - Java only *warns* if you leave off `<..>`
  - Java gives an *error* if the type does not meet the bound

Uses of Type Arguments

- Code can only use methods from the type bound

```
class Foo1<E extends Object> {
  public int m(E arg) {
    return arg.intValue();  // error!
  }
}

class Foo2<E extends Number> {
  public int m(E arg) {
    return arg.intValue();  // okay!
  }
}
```

  – can only call methods guaranteed to be there

More Examples of Generic Classes

- These can look pretty crazy at first:

```
class Graph<N> implements Iterable<N> {
  private Map<N, Set<N>> node2neighbors;
  public Graph(Set<N> nodes, Set<Pair<N, N>> edges) {
    …
  }
}

interface Path<N, P extends Path<N, P>>
    extends Iterable<N>, Comparable<Path<N, P>> {
  public Iterator<N> iterator();
}
```

More Examples of Generic Classes

- Type argument is in scope immediately after ","
- Often see variable used in its bound:

```
class TreeSet<T extends Comparable<T>>

interface Comparable<C> {
  int compareTo(C other);
}
```

  - so `Comparable<T>` will have:

```
 int compareTo(T other);   // compare two Ts
```

More Bounds

```
class Name<A extends B>
```

– "`B`" is an upper bound

```
class Name<A extends B & C & D>
```

– can include multiple upper bounds with "`&`"
– these can be classes or interfaces

Generic Methods

Example 3

```
class Utils {
  static double sum(List<Number> list) {
    double result = 0;
    for (Number n : list) {
      result += n.doubleValue();
    }
    return result;
  }
  …
}
```

– would like `sum` to work with any type of number

  e.g., want to pass `List<Double>` or `List<Integer>`

– that will not work for reasons we will see later

Example 4

```
class Utils {
  …
  static Object choose(List<Object> list) {
    int index = (int) (list.size() * Math.random());
    return list.get(index);
  }
}
```

- **would like** `choose` **to work with any element type**

  e.g., want to pass `List<Double>` or `List<String>`
- **would like** `choose(List<Double>)` **to return** `Double`

Generic Methods in Java

Declaration

```
class Utils {
  static <T extends Number> double sum(List<T> list) {
    double result = 0;
    for (T n : list) {
      result += n.doubleValue();
    }
    return result;
  }

  static <T> T choose(List<T> list) {
    int index = (int) (list.size() * Math.random());
    return list.get(index);
  }
}
```

Use

Use

Declaration

Example 4 (Updated)

```
class Utils {
  …
  static <T> T choose(List<T> list) {
    int index = (int) (list.size() * Math.random());
    return list.get(index);
  }
}
```

– **can now call like this:**

```
List<String> list = …;
String s = choose(list);  // result is a String
```

– **no need to fill in type parameters to the method**

they are always "inferred"

Example 3 (Updated)

```
class Utils {
  static <T extends Number> double sum(List<T> list) {
    double result = 0;
    for (T n : list) {
      result += n.doubleValue();
    }
    return result;
  }
  …

}
```

– since `T extends Number`, we can call `doubleValue`

Generic Classes and Methods

```
class Name<A> {
    public <B> int foo(A a, B b) {
      …
    }
}
```

– both the class and the method have type params
– all of those are in scope within the method

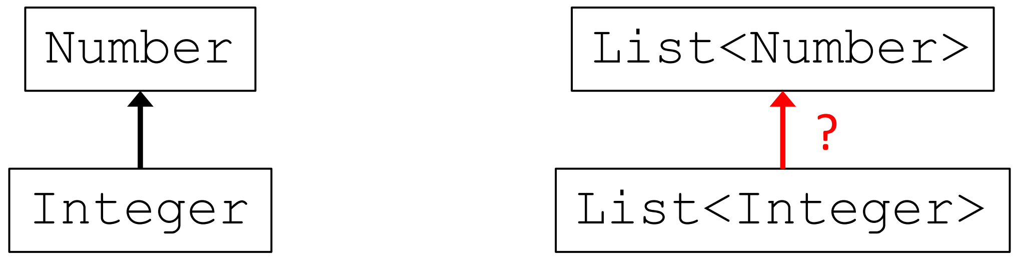
More Examples of Generic Mehods

```
<T extends Comparable<T>> T max(Collection<T> c) {
    …
}

<T extends Comparable<T>> void sort(List<T> list) {
    …  // can use List.get() and T.compareTo(T)
}

<T> void copyTo(List<T> dest, List<T> src) {
    for (T t : src)
      dest.add(t);
}
```

- last method can be improved further…

Generics & Subtyping

How Does Subtyping Work With Generics?



- `Integer` **is a subtype of** `Number`
  - **can safely be substituted where** `Number` **is expected**

    (note that both classes are immutable…)

- **Is** `List<Integer>` **a subtype of** `List<Number>`**?**
  - **can we safely substitute the latter for the former?**

How Does Subtyping Work With Generics?

$$\begin{tikzcd}
\texttt{Number} & & \texttt{List<Number>} \\
\texttt{Integer} \arrow[u] & & \texttt{List<Integer>} \arrow[u, "?"']
\end{tikzcd}$$

- **If true,** `List<T>` **would be covariant in its type arg** ↑
- **Would be contravariant if arrow was reversed** ↓

List Is Neither Covariant Nor Contravariant

```
List<Number> numList = new ArrayList<Number>();
List<Integer> intList = new ArrayList<Integer>();

// cannot substitute numList for intList
intList.add(new Integer(3));
```

  becomes `numList.add(new Integer(3));  // okay`

```
Integer n = intList.get(0);
```

  becomes `Integer n = numList.get(0);  // error!`

```
// cannot substitute intList for numList
Number n = numList.get(0);
```

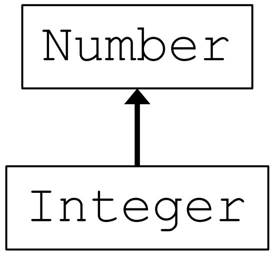
  becomes `Number n = intList.get(0);  // okay`

```
numList.add(new Double(3));
```

  becomes `intList.add(new Double(3));  // error!`

List Is Invariant

```
interface List<T> {
  void add(T elem);
  T get(int index);
}
```



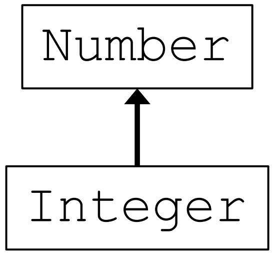
– becomes these two interfaces

```
interface List<Number> {     interface List<Integer> {
  void add(Number elem);       void add(Integer elem);
  Number get(int index);       Integer get(int index);
}                            }
```

– `add` prevents covariance, `get` prevents contravariance
– `List` is "invariant" with respect to its type arg
– Java assumes all generic types are invariant

Read-Only List is Covariant

```
interface List<T> {
  T get(int index);
}
```

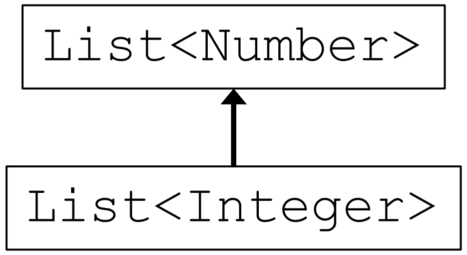


– becomes these two interfaces

```
interface List<Number> {
  Number get(int index);
}
```

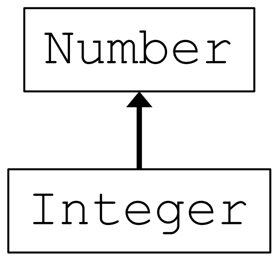
```
interface List<Integer> {
  Integer get(int index);
}
```

– this is covariant



Write-Only List Is Contravariant

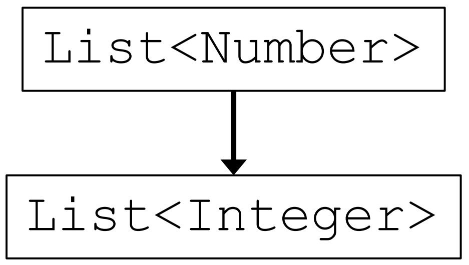
```
interface List<T> {
  void add(T elem);
}
```



– becomes these two interfaces

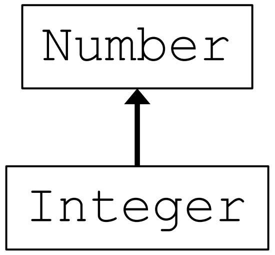
```
interface List<Number> {           interface List<Integer> {
  void add(Number elem);             void add(Integer elem);
}                                  }
```

– this is contravariant



List Is Invariant

```
interface List<T> {
  void add(T elem);
  T get(int index);
}
```



– would be covariant if `T` is only a *return value*
– would be contravariant if `T` is only an *argument*

- Java does not see these distinctions
  – all generic types are invariant
  – other languages do (e.g., C# and Scala)

Java Generic Invariance

- Cannot pass `List<Integer>` for `List<Number>` or vice versa!
  - saw this before with `sum` and `choose`
  - generic methods are needed to overcome invariance

- Still have subtyping on classes themselves

```
interface A<T> { … }
interface B<T> extends A<T> { … }
```

  - can pass `B<Integer>` where `A<Integer>` expected
  - cannot pass `B<Number>` where `A<Integer>` expected

    once a parameter changes, the classes are unrelated

Example 5

```
interface Set<E> {
  // @modifies obj
  // @effects obj = c ++ obj_0
  void addAll(_________ c)
}
```

– **what is best argument type?**

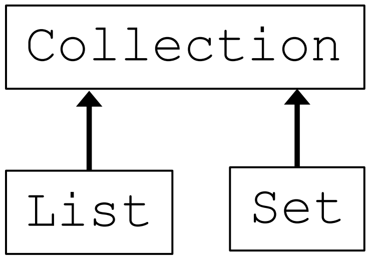
  try to make this as general as possible…

Example 5

```
interface Set<E> {
  // @modifies obj
  // @effects obj = c ++ obj_0
  void addAll(Set<E> c)
}
```

- how about this?
- it is too restrictive!

  cannot pass `List<E>` for example



Example 5

```
interface Set<E> {
  // @modifies obj
  // @effects obj = c ++ obj_0
  void addAll(Collection<E> c)
}
```

- how about this?
- still too restrictive!

  cannot pass `List<Integer>` to `addAll(Collection<Number>)`

  as we would have on `Set<Number>`
- prevented by invariance

Example 5

```
interface Set<E> {
  // @modifies obj
  // @effects obj = c ++ obj_0
  <T extends E> void addAll(Collection<T> c)
}
```

- this is the most general
  can pass `List<Integer>` to `addAll(Collection<T>)`
  allowed on `Set<Number>` since `Integer` extends `Number`
- generic methods work around invariance

Recall: More Examples of Generic Mehods

```
<T extends Comparable<T>> T max(Collection<T> c) {
    …
}

<T extends Comparable<T>> void sort(List<T> list) {
    …  // can use List.get() and T.compareTo(T)
}

<T> void copyTo(List<T> dest, List<T> src) {
    for (T t : src)
      dest.add(t);
}
```

– last method can be improved further…

More Examples (Updated)

```
<T, S extends T> void copyTo(List<T> dest, List<S> src) {
    for (S t : src)
      dest.add(t);
}
```

- any valid S is castable to T since it is a superclass
  - also works if S = T

Wildcards

Wildcard Example

- More concise way of writing some generics
  - this earlier example:

```
interface Set<E> {
  <T extends E> void addAll(Collection<T> c);
}
```

  - can be written equivalently as:

```
interface Set<E> {
  void addAll(Collection<? extends E> c);
}
```

  - wildcard is an anonymous type variable

    automatically transformed into above with some name like "`T`"

Wildcards

- **More concise way of writing some generics**
  - "`? extends E`" **is an anonymous subclass of** `E`
    or `E` itself
  - "`?`" **is an anonymous subclass of** `Object`
    or `Object` itself

? vs Object

- Do not confuse `List<?>` with `List<Object>`
  - former allows the latter
  - but also allows `List<Integer>`, `List<String>`, etc.

- Cannot pass `List<Integer>` as `List<Object>`
  - prevented by invariance

- Can pass `List<Integer>` as `List<?>`
  - allowed by generic methods

Example

```
void foo(List<?> list1, List<?> list2) {
    …
}
```

- each "?" is its own anonymous variable
- so this example becomes

```
<T1, T2> void foo(List<T1> list1, List<T2> list2) {
    …
}
```

- if you want both to be the same type, you need this

```
<T> void foo(List<T> list1, List<T> list2) {
    …
}
```

Non-Examples

```
<T> T choose(Collection<T> c) {
    …
}
```

- cannot be translated into a wildcard
- need both "T"s to be the same type
  - type returned is whatever was in the list

- another non-example:

```
<T extends Comparable<T>> T max(Collection<T> c) {
    …
}
```

Wildcards

- More concise way of writing some generics
  - "`? extends E`" is an anonymous subclass of `E` or `E` itself
  - "`?`" is an anonymous subclass of `Object` or `Object` itself
  - "`? super E`" is an anonymous superclass of `E` or `E` itself

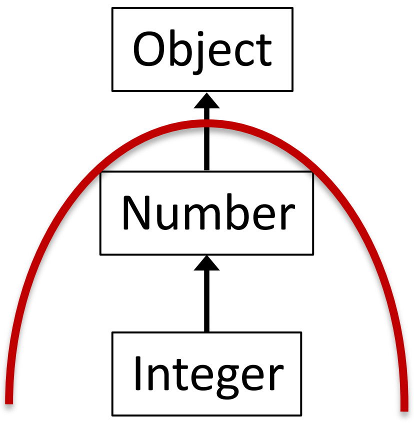
- No way to do this without wildcards!
  - no theoretical reason not to allow it

    presumably just cut due to lack of time

Type Bounds

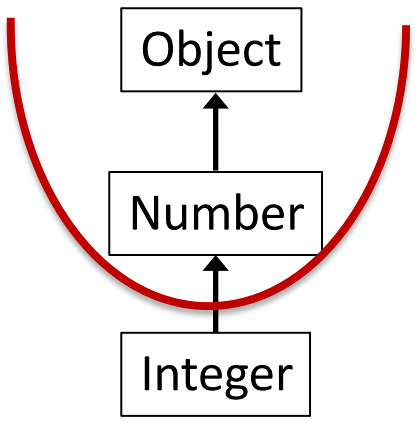
Upper Bound

`? extends Number`



Lower bound

`? super Number`



Recall: More Examples (Updated)

```
<T, S extends T> void copyTo(List<T> dest, List<S> src) {
    for (S t : src)
      dest.add(t);
}
```

- any valid `S` is castable to `T` since it is a superclass
  - also works if `S` = `T`

More Examples (Updated More)

```
<T, S extends T> void copyTo(List<T> dest, List<S> src)
```

- can write this with some wildcards

```
<T> void copyTo(List<? super T> dest,
                List<? extends T> src) {
    for (T t : src)
      dest.add(t);
}
```

- still need one variable to connect the two types
- `dest` is anything that can accept (consume) "`T`"s
- `src` is anything that can give out (produce) "`T`"s

Producer Extends, Consumer Super (PECS)

```
<T> void copyTo(List<? super T> dest,
                List<? extends T> src) {
    for (T t : src)
      dest.add(t);
}
```

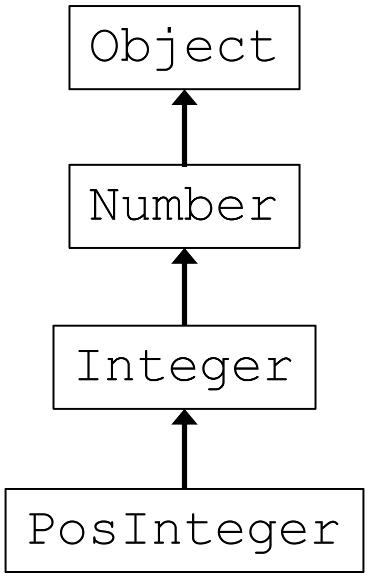
- **Should you use "`extends`" or "`super`"?**
  - **use "`? extends T`" when you get values (it's a producer)**

    fine if it gives you a subclass [covariant case]
  - **use "`? super T`" when you put values (it's a consumer)**

    fine if it accepts a superclass [contravariant case]

Legal Operations With Wildcards

```
Object o;
Number n;
Integer i;
PosInteger p;   // just pretend
List<? extends Integer> list;
```



- **Which of these lines is legal?**

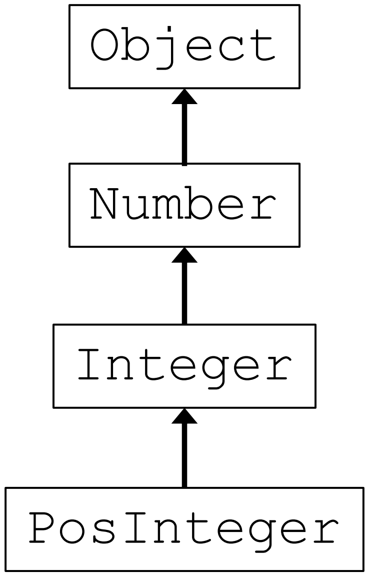
~~list = new ArrayList<Object>();~~

~~list = new ArrayList<Number>();~~

```
list = new ArrayList<Integer>();
list = new ArrayList<PosInteger>();
list = new ArrayList<NegInteger>();
```

Legal Operations With Wildcards

```
Object o;
Number n;
Integer i;
PosInteger p;   // just pretend
List<? extends Integer> list;
```

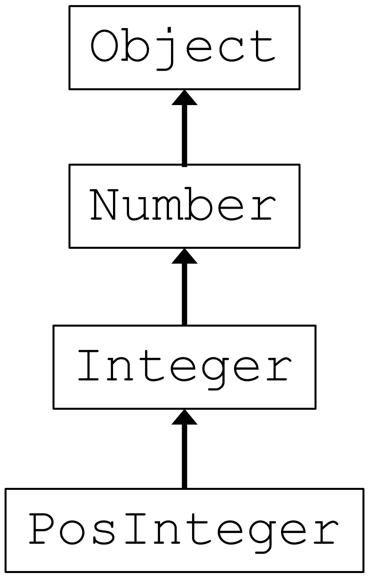


- **Which of these lines is legal?**

```
o = list.get(0);
n = list.get(0);
i = list.get(0);
```
~~p = list.get(0);~~

Legal Operations With Wildcards

```
Object o;
Number n;
Integer i;
PosInteger p;  // just pretend
List<? extends Integer> list;
```



- Which of these lines is legal?

~~list.add(o);~~

~~list.add(n);~~

~~list.add(i);~~

~~list.add(p);~~

```
list.add(null);
```

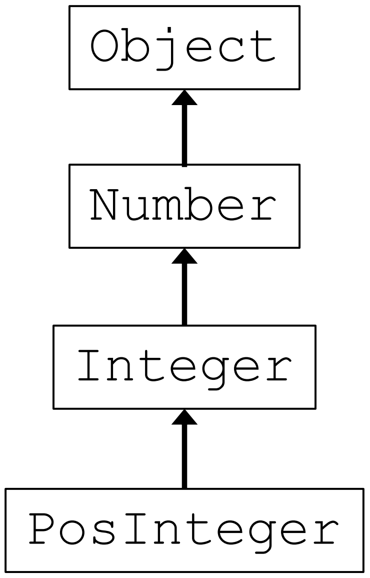
extends is for producers
almost no consuming is legal

Legal Operations With Wildcards

```
Object o;
Number n;
Integer i;
PosInteger p;   // just pretend
List<? super Integer> list;
```

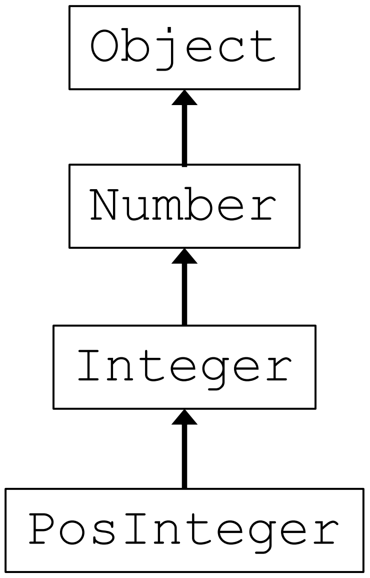
- **Which of these lines is legal?**

```
list = new ArrayList<Object>();
list = new ArrayList<Number>();
list = new ArrayList<Integer>();
```
~~list = new ArrayList<PosInteger>();~~
~~list = new ArrayList<NegInteger>();~~



Legal Operations With Wildcards

```
Object o;
Number n;
Integer i;
PosInteger p;   // just pretend
List<? super Integer> list;
```



- **Which of these lines is <u>legal</u>?**

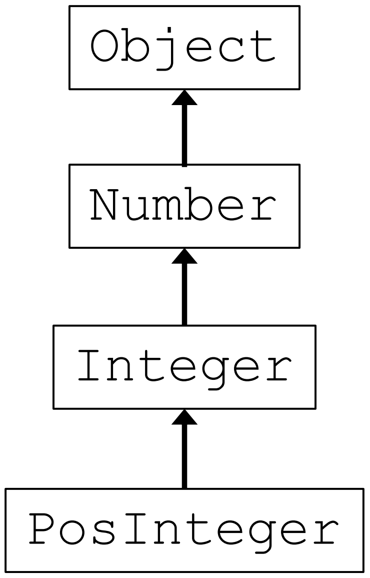
~~list.add(o);~~

~~list.add(n);~~

```
list.add(i);
list.add(p);
list.add(null);
```

Legal Operations With Wildcards

```
Object o;
Number n;
Integer i;
PosInteger p;   // just pretend
List<? super Integer> list;
```



- **Which of these lines is legal?**

```
o = list.get(0);
```
~~`n = list.get(0);`~~
~~`i = list.get(0);`~~
~~`p = list.get(0);`~~

`super` is for consumers
almost no producing is legal

The Depths of Java Generics

Arrays

- **Arrays are conceptually like `ArrayList`:**

```
Integer[] vals = …;              ArrayList<Integer> vals = …;
println(vals[0]);                println(vals.get(0));
vals[0] = 10;                    vals.set(0, 10);
```

  - **operation to get a value by index**
  - **operation to set a value by index**

Arrays

- **Arrays are conceptually like `ArrayList`:**

```
Integer[] vals = …;
println(vals[0]);
vals[0] = 10;
```

```
ArrayList<Integer> vals = …;
println(vals.get(0));
vals.set(0, 10);
```

- **How does subtyping work?**
  - **saw `ArrayList` is invariant**

$$\begin{tikzcd} \boxed{\texttt{List<Number>}} \arrow[d, shift right=2, "\times"'] \\ \boxed{\texttt{List<Integer>}} \arrow[u, shift right=2, "\times"'] \end{tikzcd}$$

Arrays

- **Arrays are conceptually like `ArrayList`:**

```
Integer[] vals = …;
println(vals[0]);
vals[0] = 10;
```

```
ArrayList<Integer> vals = …;
println(vals.get(0));
vals.set(0, 10);
```

- **How does subtyping work?**
  - **saw `ArrayList` is invariant**
  - **arrays are covariant!**

    but… how?

$$\begin{tikzcd}
\texttt{List<Number>} \arrow[d, shift right, "\times" marking] \\
\texttt{List<Integer>} \arrow[u, shift right, "\times" marking]
\end{tikzcd}$$

$$\begin{tikzcd}
\texttt{Number[]} \\
\texttt{Integer[]} \arrow[u]
\end{tikzcd}$$

Arrays Covariance Is Useful

- Necessary for things like sorting
  - one example:

```
void swap(LibraryHolding[] arr, int i, int j) {
  LibraryHolding temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}

Book[] books = …
maybeSwap(books, 0, 10);  // should work & does
```

Arrays Aren't Really Covariant

- Somewhere, this code must fail

```
void replace17(LibraryHolding[] arr,
               LibraryHolding h) {
  arr[17] = h;
}

Book[] books = …
LibraryHolding theWall = new CD("Pink Floyd", …);

replace17(books, theWall);
Book b = books[17];    // not really a book…
b.getChapters();       // doesn't have this method
```

  - the last line can't work, so where does it fail?

Arrays Aren't Really Covariant

- Somewhere, this code must fail

```
void replace17(LibraryHolding[] arr,
               LibraryHolding h) {
  arr[17] = h;
}
```

  – fails on this line, attempting to write a `CD` into `Book[]`

- Java checks this at runtime, not compile time
  – every array remember its element type
  – all writes are type checked at runtime
    pay a performance penalty for this

Type Erasure

- Type parameters become `Object` when compiled
  - e.g., the following declaration

```
List<String> list = new ArrayList<String>();
```

  - becomes

```
List<Object> list = new ArrayList<Object>();
```

- Generics are purely a compiler feature
  - Java cannot double-check type information at runtime…

Checking Type Parameters at Runtime

- Java cannot check type information at runtime:

```
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) {
  …
}
```

  - Java can check that it is a `Collection<?>`
  - but not that it is a `Collection<String>`

- As a result, this check is illegal!

Checking Type Parameters at Runtime

- Java cannot check type information at runtime:

```
List<?> list = new ArrayList<String>();
List<String> list2 = (List<String>) list;
```

  - Java can check that it is a `List<?>`
  - but not that it is a `List<String>`

- This should be illegal, but instead it's a warning
  - compiler flag will make this an error
  - these should always be treated as errors...

Checking Type Parameters at Runtime

- Ignore "unchecked cast" warnings at your peril
- Can seriously break the type system

```
public static <T> magicCast(T t, Object o) {
  return (T) o;
}

String s = "abc";
Integer n = magicCast(3, s);  // why not
```

  - can turn any type into any other type!
  - will result in incredibly painful debugging

Example: Equals in Java

```
// Represents an amount of time measured in seconds
class Duration {

  // RI: 0 <= sec < 60
  // AF: obj = 60 * this.min + this.sec
  private int min;
  private int sec;

  public boolean equals(Object o) {
    if (!(o instanceof Duration))
      return false;

    Duration d = (Duration) o;
    return this.min == d.min && this.sec == d.sec;
  }
```

- Correct and idiomatic Java

Generics Causes Problems in Equals

```
class Node<E> {

  private E data;

  public boolean equals(Object o) {
    if (!(o instanceof Node<E>))
      return false;

    Node<E> n = (Node<E>) o;
    return this.data.equals(n.data);
  }
```

- **This does not compile!**
  - cannot perform these type checks at runtime

- **So how do we fix it?**

Generics Causes Problems in Equals

```
class Node<E> {

  private E data;

  public boolean equals(Object o) {
    if (!(o instanceof Node<?>))
      return false;

    Node<?> n = (Node<?>) o;
    return this.data.equals(n.data);
  }
```

- The call to `this.data.equals` will check types
  - not necessary for us to do it again here

Type Erasure and Arrays

```
class Foo<E> {
  private E[] data;
  public Foo() {
    this.data = new E[10];
  }
```

- This is illegal!
  - "`E`" becomes `Object` when compiled but...
  - arrays need to know the element type to check writes
- What should you do?
  - just use `ArrayList` instead