

CSE 331

Software Design & Implementation

Topic: Generics

 **Discussion:** Would you ever quit your job to protest a decision?

Reminders

- If you don't know where to start, read [answers-hw6.txt](#)!
- Don't add generics to HW6

Upcoming Deadlines

- Prep. Quiz: HW6 due Monday (7/24)
- HW6 due Thursday (7/27)

Last Time...

- True Subtypes vs. Subclasses
- Designing for Inheritance
- Ethics I

Today's Agenda

- Intro to Generics
- Generic Methods
- Generics and Subtyping
- Arrays
- Type Bounds
- Wildcards
- Type Erasure

(over next two lectures)

Where are we?

- Now:
 - basics of generic types for classes and interfaces
 - basics of *bounding* generics
 - generic *methods* [not just using type parameters of class]
 - generics and *subtyping*
 - related digression: Java's *array subtyping*
 - using *bounds* for more flexible subtyping
 - using *wildcards* for more convenient bounds
 - Java realities: type erasure
 - unchecked casts
 - **equals** interactions
 - creating generic arrays

Preface

- This lecture will get into the gritty details of generics
- In practice:
 - you will constantly need to **use** generic classes
 - e.g., the collections library
 - but you will rarely need to **write** generic classes
 - (generic methods are a little more common)
 - unless you are writing a container class, you are probably making a mistake by making it generic
- We will go through all the details so that you have seen it once

Pre-generic Collection Use

Checking types were correct was done at run-time.

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

No compiler help to ensure type constraints are satisfied

Alternative: Many, Many Classes

```
interface ListOfNumbers {  
    boolean add(Number elt);  
    Number get(int index);  
}
```

```
interface ListOfIntegers {  
    boolean add(Integer elt);  
    Integer get(int index);  
}
```

... and many, many more

Why we *love* abstraction

Hide details

- avoid getting lost in details (readability)
- permit details to change later on (changeability)

Give a *meaningful name* to a concept (readability)

Permit *reuse* in new contexts

- avoid duplication: error-prone, confusing, less changeable
- save reimplementing effort

Varieties of abstraction

Abstraction over *computation*: procedures (methods)

```
int x1, y1, x2, y2;  
Math.sqrt(x1*x1 + y1*y1);  
Math.sqrt(x2*x2 + y2*y2);
```

Abstraction over *data*: ADTs (classes, interfaces)

```
Point p1, p2;
```

Abstraction over *types*: polymorphism (generics)

```
Point<Integer>, Point<Double>
```

Related abstractions

```
interface ListOfNumbers {  
    boolean add(Number elt);  
    Number get(int index);  
}
```

```
interface ListOfIntegers {  
    boolean add(Integer elt);  
    Integer get(int index);  
}
```

... and many, many more

// abstracts over element type

```
interface List<E> {  
    boolean add(E n);  
    E get(int index);  
}
```

Lets us use types:

List<Integer>

List<Number>

List<String>

List<List<String>>

...

An analogous parameter

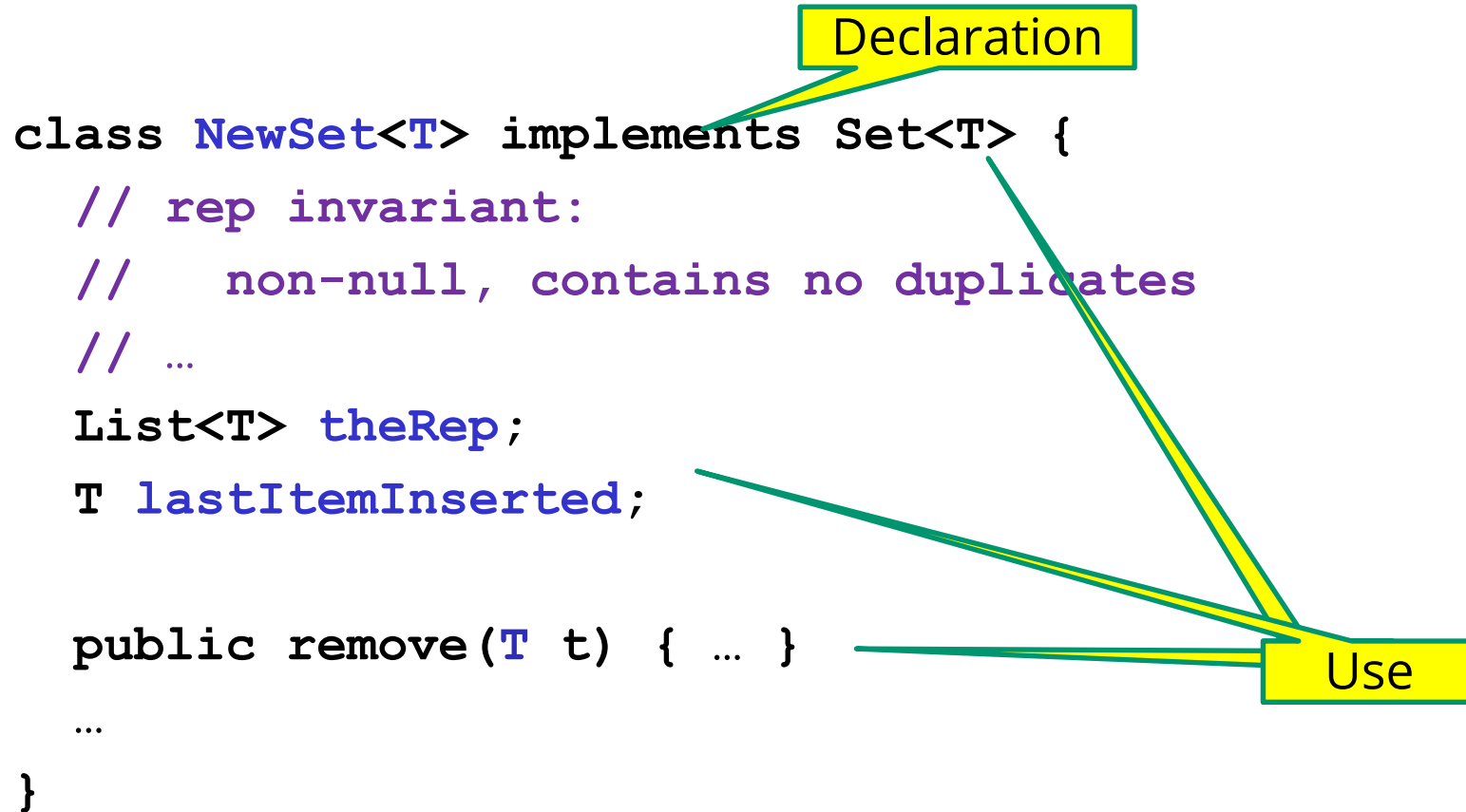
```
interface ListOfIntegers {  
    boolean add(Integer elt);  
    Integer get(int index);  
}
```

- Declares a new **variable**, called a **parameter**
- Instantiate with any expression of the right type
 - e.g., `lst.add(7)`
- Type of `add` is *Integer* -> *boolean*

```
interface List<E> {  
    boolean add(E n);  
    E get(int index);  
}
```

- Declares a new **type variable**, called a **type parameter**
- Instantiate with any (reference) type
 - e.g., `List<String>`
- **"Type"** of `List` is *Type* -> *List<Type>*
 - never just use `List` (allowed for backward-compatibility only)

Type variables are types



Declaring and instantiating generics

```
class Name<TypeVar1, ..., TypeVarN> {...}
```

```
interface Name<TypeVar1, ..., TypeVarN> {...}
```

- often one-letter name such as:
 T for **Type**, **E** for **Element**,
 K for **Key**, **V** for **Value**, ...

To instantiate a generic class/interface, supply type arguments:

```
Name<Type1, ..., TypeN>
```

Restricting instantiations by clients

```
boolean add1(Object elt);  
boolean add2(Number elt);  
add1(new Date()); // OK  
add2(new Date()); // compile-time error
```

```
interface List1<E extends Object> {...}  
interface List2<E extends Number> {...}  
  
List1<Date> // OK, Date is a subtype of Object  
List2<Date> // compile-time error, Date is not a  
            // subtype of Number
```

Revised definition

```
class Name<TypeVar1 extends Type1,  
        ...,  
        TypeVarN extends TypeN> {...}
```

- (same for interface definitions)
- (default upper bound is **Object**)

To instantiate a generic class/interface, supply type arguments:

```
Name<Type1, ..., TypeN>
```

Compile-time error if type is not a subtype of the upper bound

Using type variables

Code can perform any operation permitted by the bound

- because we know all instantiations will be subtypes!
- an enforced precondition on type instantiations

```
class Foo1<E extends Object> {  
    void m(E arg) {  
        arg.intValue(); // compiler error, E might not  
                        // support intValue  
    }  
}
```

```
class Foo2<E extends Number> {  
    void m(E arg) {  
        arg.intValue(); // OK, since Number and its  
                        // subtypes support intValue  
    }  
}
```


More examples

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

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

(Note: you don't want to use this code in your HW6 – we'll do it in HW7.)

More bounds

`<TypeVar extends SuperType>`

- an *upper bound*; accepts given supertype or any of its subtypes

`<TypeVar extends ClassA & InterfaceB & InterfaceC & ...>`

- *multiple* upper bounds (superclass/interfaces) with `&`

Example:

```
// tree set works for any comparable type
public class TreeSet<T extends Comparable<T>> {
    ...
}
```

Where are we?

- Done:
 - basics of generic types for classes and interfaces
 - basics of *bounding* generics
- Now:
 - *generic methods* [not just using type parameters of class]
 - generics and *subtyping*
 - related digression: Java's *array subtyping*
 - using *bounds* for more flexible subtyping
 - using *wildcards* for more convenient bounds
 - Java realities: type erasure
 - unchecked casts
 - **equals** interactions
 - creating generic arrays

Not all generics are for collections

```
class Utils {
    static double sumList(List<Number> lst) {
        double result = 0.0;
        for (Number n : lst) {
            result += n.doubleValue();
        }
        return result;
    }
    static Number choose(List<Number> lst) {
        int i = ... // random number < lst.size()
        return lst.get(i);
    }
}
```

Weaknesses

- Would like to use `sumList` for any subtype of `Number`
 - for example, `Double` or `Integer`
 - but as we will see, `List<Double>` is not a subtype of `List<Number>`
- Would like to use `choose` for any element type
 - i.e., any subclass of `Object`
 - no need to restrict to subclasses of `Number`
 - want to tell clients more about return type than `Object`
- Class `Utils` is not generic, but the *methods* should be generic

Much better

Have to declare
type parameter(s)

```
class Utils {  
    public static <T extends Number> double sumList(List<T> lst) {  
        double result = 0.0;  
        for (Number n : lst) { // T also works  
            result += n.doubleValue();  
        }  
        return result;  
    }  
    public static <T> T choose(List<T> lst) {  
        int i = ... // random number < lst.size  
        return lst.get(i);  
    }  
}
```

Have to declare
type parameter(s)

Using generics in methods

- Instance methods can use type parameters of the class
- Instance methods can have their own type parameters
 - generic methods
- Callers to generic methods need not explicitly instantiate the methods' type parameters
 - compiler just figures it out for you
 - example of *type inference*

More examples

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

```
<T extends Comparable<T>> void sort(List<T> list) {  
    // ... use list.get() and T's compareTo  
}
```

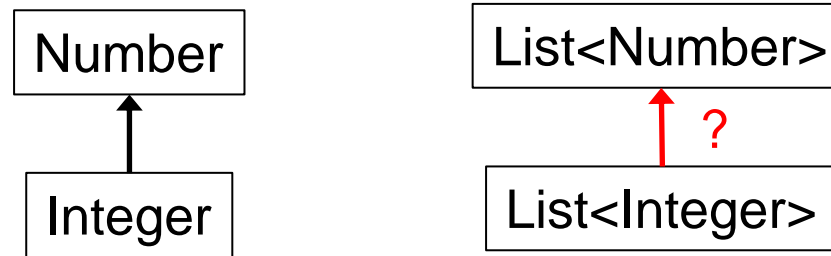
(This works but will be even more useful later with more bounds)

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


Where are we?

- Done:
 - basics of generic types for classes and interfaces
 - basics of *bounding* generics
- Now:
 - generic *methods* [not just using type parameters of class]
 - *generics and subtyping*
 - related digression: Java's *array subtyping*
 - using *bounds* for more flexible subtyping
 - using *wildcards* for more convenient bounds
 - Java realities: type erasure
 - unchecked casts
 - **equals** interactions
 - creating generic arrays

Generics and subtyping



- **Integer** can be used wherever **Number** is expected
 - this is the notion of a subtype
 - (specifically, the Liskov substitutability principle)
 - i.e, **Integer** satisfies a *stronger spec* than **Number**
 - only adds methods and strengthens existing methods
- Can you safely substitute **List<Integer>** wherever a **List<Number>** is used without possibility of error?

Generics and subtyping

```
List<Number> numList = new List<Number>();  
List<Integer> intList = new List<Integer>();  
  
intList.add(new Integer(3));  
    -> numList.add(new Integer(3));           // okay  
numList.add(new Double(3.0));  
    -> intList.add(new Double(3.0));         // not legal  
  
Number n = numList.get(0);  
    -> Number n = intList.get(0);           // okay  
Integer n = intList.get(0);  
    -> Integer n = numList.get(0);         // illegal
```

Neither type can be substituted for the other legally in all situations!

List<Number> and List<Integer>

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

So type List<Number> has:

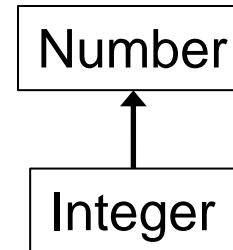
```
boolean add(Number elt);  
Number get(int index);
```

So type List<Integer> has:

```
boolean add(Integer elt);  
Integer get(int index);
```

Java subtyping is *invariant* with respect to generics

- Not covariant and not contravariant
- Neither List<Number> nor List<Integer> subtype of other



Hard to remember?

If **Type2** and **Type3** are different,
then **Type1<Type2>** is *not* a subtype of **Type1<Type3>**

Previous example shows why:

- Observer method prevents “one direction”
- Mutator/producer method prevents “the other direction”

If our types have only observers or only mutators, then one direction of subtyping would be sound

- But Java’s type system does not “notice this” so such subtyping is never allowed in Java

Read-only allows covariance

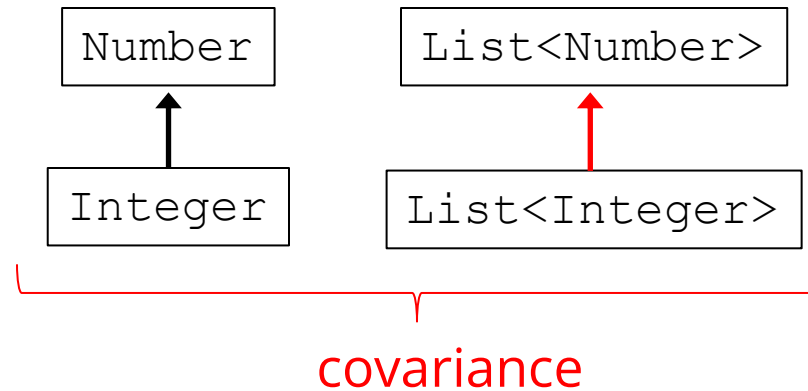
```
interface List<T> {  
    T get(int index);  
}
```

So type `List<Number>` has:
`Number get(int index);`

So type `List<Integer>` has:
`Integer get(int index);`

So *covariant* subtyping would be correct:
– `List<Integer>` a subtype of `List<Number>`

But Java does not analyze interface definitions like this
– conservatively disallows this subtyping



Write-only allows contravariance

```
interface List<T> {  
    boolean add(T elt);  
}
```

So type `List<Number>` has:

```
boolean add(Number elt);
```

So type `List<Integer>` has:

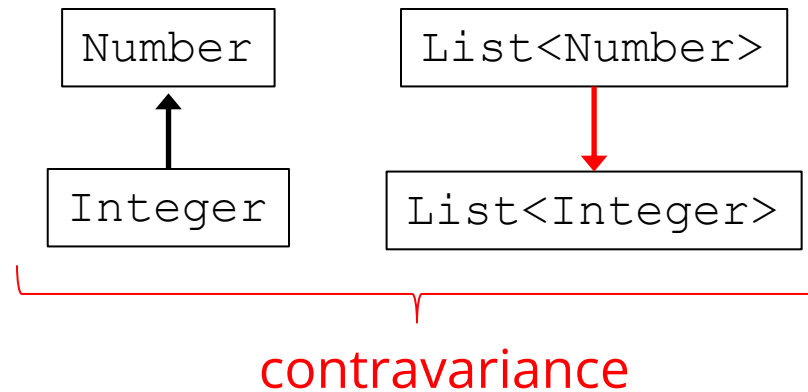
```
boolean add(Integer elt);
```

So *contravariant* subtyping would be correct:

- `List<Number>` a subtype of `List<Integer>`

But Java does not analyze interface definitions like this

- conservatively disallows this subtyping



Co- and Contra-variance

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

In general, `List<T>` should be

- covariant if `T` only appears as a return value
- contravariant if `T` only appears as an argument

Some languages (e.g., Scala and C#) allow this

Java does not:

- cannot substitute `List<T1>` for `List<T2>` unless `T1 = T2`

About the parameters

- So we have seen `List<Integer>` and `List<Number>` are not subtype-related
- There is “as expected” subtyping on the generic types themselves
- Example: If `HeftyBag` extends `Bag`, then
 - `HeftyBag<Integer>` is a subtype of `Bag<Integer>`
 - `HeftyBag<Number>` is a subtype of `Bag<Number>`
 - `HeftyBag<String>` is a subtype of `Bag<String>`
 - ...

Where are we?

- Done:
 - basics of generic types for classes and interfaces
 - basics of *bounding* generics
- Now:
 - generic *methods* [not just using type parameters of class]
 - generics and *subtyping*
 - related digression: *Java's array subtyping*
 - using *bounds* for more flexible subtyping
 - using *wildcards* for more convenient bounds
 - Java realities: type erasure
 - unchecked casts
 - **equals** interactions
 - creating generic arrays



Java arrays

We know how to use arrays:

- declare an array holding **Type** elements: **Type []**
- get an element: **x[i]**
- set an element **x[i] = e;**

Java included the syntax above because it's common and concise

But can reason about how it should work the same as this:

```
class Array<T> {  
    public T get(int i) { ... "magic" ... }  
    public T set(T newVal, int i) {... "magic" ...}  
}
```

So: If **Type1** is a subtype of **Type2**, how should **Type1 []** and **Type2 []** be related??

Java Arrays

- Given everything we have learned, if **Type1** is a subtype of **Type2**, then **Type1 []** and **Type2 []** should be unrelated
 - invariant subtyping for generics
 - because arrays are mutable



Surprise!

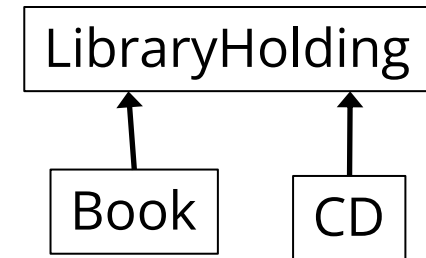
- Given everything we have learned, if **Type1** is a subtype of **Type2**, then **Type1 []** and **Type2 []** should be unrelated
 - invariant subtyping for generics
 - because arrays are mutable
- But in Java, if **Type1** is a subtype of **Type2**, then **Type1 []** *is a subtype* of **Type2 []** (covariant subtyping)
 - not true subtyping: the subtype does not support setting an array element to hold a **Type2** (spoiler: throws an exception)
 - Java (and C#) made this decision in pre-generics days
 - needed to write reusable sorting routines, etc.
 - also `_(ツ)_/`

What can happen: the good

Programmers can use this subtyping to “do okay stuff”

```
void maybeSwap(LibraryHolding[] arr) {  
    if(arr[17].dueDate() < arr[34].dueDate())  
        // ... swap arr[17] and arr[34]  
}
```

```
// client with subtype  
Book[] books = ...;  
maybeSwap(books); // relies on covariant  
                  // array subtyping
```



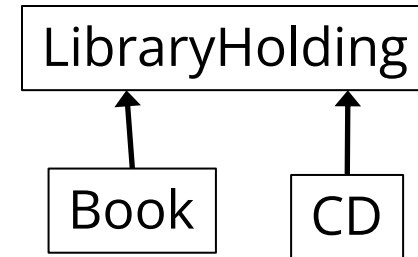
What can happen: the bad

Something in here must go wrong!

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

// client with subtype
Book[] books = ...;
LibraryHolding theWall = new CD("Pink Floyd",
                                "The Wall", ...);

replace17(books, theWall);
Book b = books[17]; // would hold a CD
b.getChapters();   // so this would fail
```



Java's choice

- Java normally guarantees run-time type is a subtype of the compile-time type
 - this was violated for the **Book b** variable
- To preserve the guarantee, Java must never get that far:
 - each array “knows” its actual run-time type (e.g., **Book []**)
 - storing a supertype into an index causes **ArrayStoreException** (at run time)
- So the body of **replace17** would raise an exception
 - even though **replace17** is entirely reasonable
 - and fine for plenty of “careful” clients
 - *every Java array-update includes this run-time check*
 - (array-reads never fail this way – why?)
 - **beware careful with array subtyping**

Where are we?

- Done:
 - basics of generic types for classes and interfaces
 - basics of *bounding* generics
- Now:
 - generic *methods* [not just using type parameters of class]
 - generics and *subtyping*
 - related digression: Java's *array subtyping*
 - using *bounds* for more flexible subtyping
 - using *wildcards* for more convenient bounds
 - Java realities: type erasure
 - unchecked casts
 - **equals** interactions
 - creating generic arrays

Best type for addAll

```
interface Set<E> {  
    // Adds all elements in c to this set  
    // (that are not already present)  
    void addAll(_____ c);  
}
```

What is the best type for `addAll`'s parameter?

- Allow as many clients as possible...
- ... while allowing correct implementations

Best type for `addAll`

```
interface Set<E> {  
    // Adds all elements in c to this set  
    // (that are not already present)  
    void addAll(_____ c);  
}
```

```
void addAll(Set<E> c);
```

Too restrictive:

- does not let clients pass other collections, like `List<E>`
- better: use a supertype interface with just what `addAll` needs

Best type for addAll

```
interface Set<E> {  
    // Adds all elements in c to this set  
    // (that are not already present)  
    void addAll(_____ c);  
}
```

```
void addAll(Collection<E> c);
```

Still too restrictive:

- cannot pass a **List<Integer>** to **addAll** for a **Set<Number>**
- that should be okay because **addAll** implementations only need to read from **c**, not put elements in it
- but Java does not allow it
 - this is the invariant-subtyping limitation

Best type for addAll

```
interface Set<E> {  
    // Adds all elements in c to this set  
    // (that are not already present)  
    void addAll(_____ c);  
}
```

```
<T extends E> void addAll(Collection<T> c);
```

The fix: bounded generic type parameter

- *can* pass a **List<Integer>** to **addAll** for a **Set<Number>**
- **addAll** implementations won't know what element type **T** is, but will know it is a subtype of **E**
 - it cannot add anything to collection **c** refers to
 - but this is enough to implement **addAll**

Generic methods get around invariance

You cannot pass `List<Integer>` to method expecting `List<Number>`

- Java subtyping is invariant with respect to type parameters

Get around it by making your **method** generic:

```
<T extends Number> void sumList(List<T> nums) {  
    double s = 0;  
    for (T t : nums)  
        s += t.doubleValue();  
    return s;  
}
```

Compiler will automatically infer type argument

Revisit copy method

Earlier we saw this:

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

Now we can do this (which is more general):

```
<T1, T2 extends T1> void copyTo(List<T1> dst,  
                                List<T2> src) {  
    for (T2 t : src)  
        dst.add(t);  
}
```

Where are we?

- Done:
 - basics of generic types for classes and interfaces
 - basics of *bounding* generics
- Now:
 - generic *methods* [not just using type parameters of class]
 - generics and *subtyping*
 - related digression: Java's *array subtyping*
 - using *bounds* for more flexible subtyping
 - using *wildcards* for more convenient bounds
 - Java realities: type erasure
 - unchecked casts
 - **equals** interactions
 - creating generic arrays

Examples

[Compare to earlier version]

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

- More idiomatic (but equally powerful) compared to
 <T extends E> void addAll(Collection<T> c);
- More powerful than void addAll(Collection<E> c);

Wildcards

Syntax: for a type-parameter instantiation (inside the <...>), can write:

- ? **extends Type**, some unspecified subtype of **Type**
- ? is shorthand for ? **extends Object**

A wildcard is essentially an ***anonymous** type variable*

- each ? stands for some possibly-different unknown type

? versus Object

? indicates a particular but unknown type

```
void printAll(List<?> lst) {...}
```

Difference between `List<?>` and `List<Object>`:

- can instantiate ? with any type: `Object`, `String`, ...
- `List<Object>` much more restrictive:
 - e.g., wouldn't take a `List<String>`

Difference between `List<Number>` and `List<? extends Number>`:

- can instantiate ? with `Number`, `Integer`, `Double`, ...
- first version is much more restrictive

Non-example

```
<T extends Comparable<T>> T max(Collection<T> c);
```

No change because **T** used *more than once*

- must choose a name to say that two types must match

Wildcards

Syntax: for a type-parameter instantiation (inside the <...>), can write:

- **? extends Type**, some unspecified subtype of **Type**
- **?** is shorthand for **? extends Object**

A wildcard is essentially an ***anonymous** type variable*

- each **?** stands for some possibly-different unknown type
- use a wildcard when you would use a type variable only once (no need to give it a name)
- communicates to readers of your code that the type's "identity" is not needed anywhere else

Wildcards

Syntax: for a type-parameter instantiation (inside the <...>), can write:

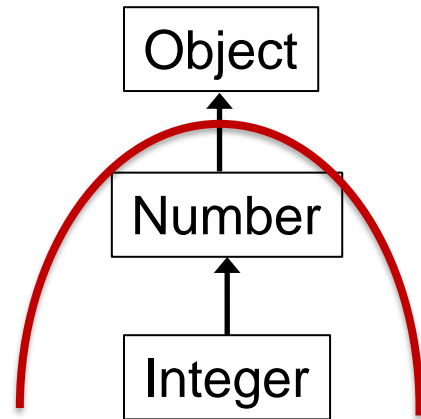
- ? **extends Type**, some unspecified subtype of **Type**
- ? is shorthand for ? **extends Object**
- ? **super Type**, some unspecified superclass of **Type**

Wildcard can have lower bounds instead of upper bounds!

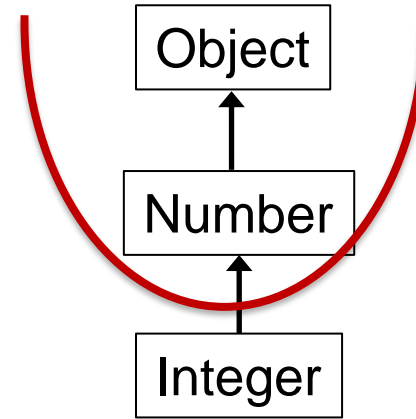
- says that ? must be **Type** or a superclass of **Type**

Type Bounds

Upper Bound
? **extends Number**



Lower bound
? **super Number**



Revisit copy method

First version:

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

More general version:

```
<T1, T2 extends T1> void copyTo(List<T1> dst,  
                                List<T2> src) {  
    for (T2 t : src)  
        dst.add(t);  
}
```


More examples

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

Why this works:

- lower bound of **T** for where callee puts values
- upper bound of **T** for where callee gets values
- callers get the subtyping they want
 - Example: `copy(numberList, integerList)`
 - Example: `copy(stringList, stringList)`

PECS: Producer Extends, Consumer Super

Should you use **extends** or **super** or neither?

- use ? **extends** **T** when you *get* values (from a *producer*)
 - no problem if it's a subtype
 - (the co-variant subtyping case)
- use ? **super** **T** when you *put* values (into a *consumer*)
 - no problem if it's a supertype
 - (the contra-variant subtyping case)
- use neither (just **T**, not ?) if you both *get* and *put*
 - can't be as flexible here

```
<T> void copyTo(List<? super T> dst,  
               List<? extends T> src);
```

More on lower bounds

- As we've seen, lower-bound ? **super T** is useful for “consumers”
- Upper-bound ? **extends T** could be rewritten without wildcards, but wildcards preferred style where they suffice
- But lower-bound is *only* available for wildcards in Java
 - this does not parse:

```
<T super Foo> void m(Bar<T> x) ;
```
 - no good reason for Java not to support such lower bounds except designers decided it wasn't useful enough to bother
 - ͇_(ツ)_/͇

Legal operations on wildcard types

```
Object o;
```

```
Number n;
```

```
Integer i;
```

```
PositiveInteger p;
```

```
List<? extends Integer> lei;
```

First, which of these is legal?

```
lei = new ArrayList<Object>();
```

```
lei = new ArrayList<Number>();
```

```
lei = new ArrayList<Integer>();
```

```
lei = new ArrayList<PositiveInteger>();
```

```
lei = new ArrayList<NegativeInteger>();
```

Legal operations on wildcard types

```
Object o;
```

```
Number n;
```

```
Integer i;
```

```
PositiveInteger p;
```

```
List<? extends Integer> lei;
```

First, which of these is legal?

```
lei = new ArrayList<Object>();
```

```
lei = new ArrayList<Number>();
```

```
lei = new ArrayList<Integer>();
```

```
lei = new ArrayList<PositiveInteger>();
```

```
lei = new ArrayList<NegativeInteger>();
```

Legal operations on wildcard types

```
Object o;  
Number n;  
Integer i;  
PositiveInteger p;  
  
List<? extends Integer> lei;
```

Which of these is legal?

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

First, which of these is legal?

```
lei = new ArrayList<Object>();  
lei = new ArrayList<Number>();  
lei = new ArrayList<Integer>();  
lei = new ArrayList<PositiveInteger>();  
lei = new ArrayList<NegativeInteger>();
```

Legal operations on wildcard types

```
Object o;  
Number n;  
Integer i;  
PositiveInteger p;  
  
List<? extends Integer> lei;
```

Which of these is legal?

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

First, which of these is legal?

```
lei = new ArrayList<Object>();  
lei = new ArrayList<Number>();  
lei = new ArrayList<Integer>();  
lei = new ArrayList<PositiveInteger>();  
lei = new ArrayList<NegativeInteger>();
```

Legal operations on wildcard types

```
Object o;  
Number n;  
Integer i;  
PositiveInteger p;  
  
List<? extends Integer> lei;
```

First, which of these is legal?

```
lei = new ArrayList<Object>();  
lei = new ArrayList<Number>();  
lei = new ArrayList<Integer>();  
lei = new ArrayList<PositiveInteger>();  
lei = new ArrayList<NegativeInteger>();
```

Which of these is legal?

```
o = lei.get(0);  
n = lei.get(0);  
i = lei.get(0);  
p = lei.get(0);  
lei.add(o);  
lei.add(n);  
lei.add(i);  
lei.add(p);  
lei.add(null);
```


Legal operations on wildcard types

```
Object o;  
Number n;  
Integer i;  
PositiveInteger p;  
  
List<? extends Integer> lei;
```

First, which of these is legal?

```
lei = new ArrayList<Object>();  
lei = new ArrayList<Number>();  
lei = new ArrayList<Integer>();  
lei = new ArrayList<PositiveInteger>();  
lei = new ArrayList<NegativeInteger>();
```

Which of these is legal?

```
o = lei.get(0);  
n = lei.get(0);  
i = lei.get(0);  
p = lei.get(0);  
lei.add(o);  
lei.add(n);  
lei.add(i);  
lei.add(p);  
lei.add(null);
```

Legal operations on wildcard types

```
Object o;
```

```
Number n;
```

```
Integer i;
```

```
PositiveInteger p;
```

```
List<? super Integer> lsi;
```

First, which of these is legal?

```
lsi = new ArrayList<Object>;
```

```
lsi = new ArrayList<Number>;
```

```
lsi = new ArrayList<Integer>;
```

```
lsi = new ArrayList<PositiveInteger>;
```

```
lsi = new ArrayList<NegativeInteger>;
```

Legal operations on wildcard types

```
Object o;
```

```
Number n;
```

```
Integer i;
```

```
PositiveInteger p;
```

```
List<? super Integer> lsi;
```

First, which of these is legal?

```
lsi = new ArrayList<Object>;
```

```
lsi = new ArrayList<Number>;
```

```
lsi = new ArrayList<Integer>;
```

```
lsi = new ArrayList<PositiveInteger>;
```

```
lsi = new ArrayList<NegativeInteger>;
```

Legal operations on wildcard types

```
Object o;  
Number n;  
Integer i;  
PositiveInteger p;  
  
List<? super Integer> lsi;
```

Which of these is legal?

```
lsi.add(o);  
lsi.add(n);  
lsi.add(i);  
lsi.add(p);  
lsi.add(null);
```

First, which of these is legal?

```
lsi = new ArrayList<Object>;  
lsi = new ArrayList<Number>;  
lsi = new ArrayList<Integer>;  
lsi = new ArrayList<PositiveInteger>;  
lsi = new ArrayList<NegativeInteger>;
```

Legal operations on wildcard types

```
Object o;  
Number n;  
Integer i;  
PositiveInteger p;  
  
List<? super Integer> lsi;
```

Which of these is legal?

```
lsi.add(o);  
lsi.add(n);  
lsi.add(i);  
lsi.add(p);  
lsi.add(null);
```

First, which of these is legal?

```
lsi = new ArrayList<Object>;  
lsi = new ArrayList<Number>;  
lsi = new ArrayList<Integer>;  
lsi = new ArrayList<PositiveInteger>;  
lsi = new ArrayList<NegativeInteger>;
```

Legal operations on wildcard types

```
Object o;  
Number n;  
Integer i;  
PositiveInteger p;  
  
List<? super Integer> lsi;
```

First, which of these is legal?

```
lsi = new ArrayList<Object>;  
lsi = new ArrayList<Number>;  
lsi = new ArrayList<Integer>;  
lsi = new ArrayList<PositiveInteger>;  
lsi = new ArrayList<NegativeInteger>;
```

Which of these is legal?

```
lsi.add(o);  
lsi.add(n);  
lsi.add(i);  
lsi.add(p);  
lsi.add(null);  
o = lsi.get(0);  
n = lsi.get(0);  
i = lsi.get(0);  
p = lsi.get(0);
```

Legal operations on wildcard types

```
Object o;  
Number n;  
Integer i;  
PositiveInteger p;  
  
List<? super Integer> lsi;
```

First, which of these is legal?

```
lsi = new ArrayList<Object>;  
lsi = new ArrayList<Number>;  
lsi = new ArrayList<Integer>;  
lsi = new ArrayList<PositiveInteger>;  
lsi = new ArrayList<NegativeInteger>;
```

Which of these is legal?

```
lsi.add(o);  
lsi.add(n);  
lsi.add(i);  
lsi.add(p);  
lsi.add(null);  
o = lsi.get(0);  
n = lsi.get(0);  
i = lsi.get(0);  
p = lsi.get(0);
```

Where are we?

- Done:
 - basics of generic types for classes and interfaces
 - basics of *bounding* generics
- Now:
 - generic *methods* [not just using type parameters of class]
 - generics and *subtyping*
 - related digression: Java's *array subtyping*
 - using *bounds* for more flexible subtyping
 - using *wildcards* for more convenient bounds
 - Java realities: type erasure
 - unchecked casts
 - **equals** interactions
 - creating generic arrays



Type erasure

All generic types become type `Object` once compiled

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

at runtime, becomes

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

Generics are purely a *compiler* feature!

Type erasure example

```
import java.util.*;

public class Erasure {

    public static void foo() {
        List<String> lst = new ArrayList<String>();
        lst.add("abc");
        lst.add("def");
    }
}
```

Type erasure example

Compile-time signature is `add(String)` but the bytecodes say...

```
public static void foo();
Code:
  0: new          #7          // class java/util/ArrayList
  3: dup
  4: invokespecial #9          // Method java/util/ArrayList."<init>":()V
  7: astore_0
  8: aload_0
  9: ldc          #10         // String abc
 11: invokeinterface #12, 2    // InterfaceMethod java/util/List.add:(Ljava/lang/Object;)V
 16: pop
 17: aload_0
 18: ldc          #18         // String def
 20: invokeinterface #12, 2    // InterfaceMethod java/util/List.add:(Ljava/lang/Object;)V
 25: pop
 26: return
```

Type erasure

All generic types become type **Object** once compiled

- gives backward compatibility (a selling point at time of adoption)
- at run-time, all generic instantiations have the same type

Cannot use **instanceof** to discover a type parameter

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

Generics and casting

Casting to generic type results in an important warning

```
List<?> lg = new ArrayList<String>(); // ok
List<String> ls = (List<String>) lg; // warn
```

Compiler gives a warning because the runtime system *will not check for you*

Usually, if you think you need to do this, you're wrong

- a real need to do this is extremely rare

`Object` can also be cast to any generic type ☹

```
public static <T> T badCast(T t, Object o) {
    return (T) o; // unchecked warning
}
```

The bottom-line

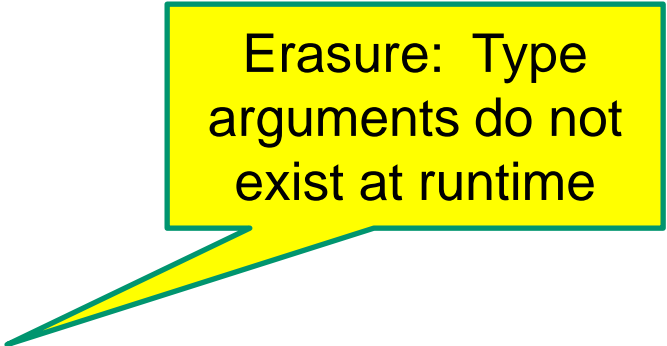
- Java guarantees a `List<String>` variable always holds a (subtype of) the *raw type* `List`
- Java does not guarantee a `List<String>` variable always has only `String` elements at run-time
 - will be true if no unchecked cast warnings are shown
 - compiler inserts casts to/from `Object` for generics
 - if these casts fail, ***hard-to-debug errors result:***
often far from where conceptual mistake occurred
- So, two reasons not to ignore warnings:
 1. You're violating good style/design/subtyping/generics
 2. You're risking difficult debugging

Recall equals

```
class Node {  
    ...  
    @Override  
    public boolean equals(Object obj) {  
        if (!(obj instanceof Node)) {  
            return false;  
        }  
        Node n = (Node) obj;  
        return this.data.equals(n.data);  
    }  
    ...  
}
```

equals for a parameterized class

```
class Node<E> {  
    ...  
    @Override  
    public boolean equals(Object obj) {  
        if (!(obj instanceof Node<E>)) {  
            return false;  
        }  
        Node<E> n = (Node<E>) obj;  
        return this.data.equals(n.data);  
    }  
    ...  
}
```



Erasure: Type arguments do not exist at runtime

equals for a parameterized class

```
class Node<E> {  
    ...  
    @Override  
    public boolean equals(Object obj) {  
        if (!(obj instanceof Node<E>)) {  
            return false;  
        }  
        Node<E> n = (Node<E>) obj;  
        return this.data.equals(n.data);  
    }  
    ...  
}
```

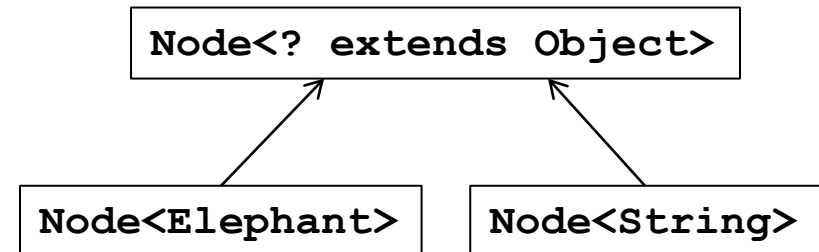
More erasure: At run time, do not know what **E** is and will not be checked, so don't indicate otherwise

equals for a parameterized class

```
class Node<E> {  
    ...  
    @Override  
    public boolean equals(Object obj) {  
        if (!(obj instanceof Node<?>)) {  
            return false;  
        }  
        Node<?> n = (Node<?>) obj;  
        return this.data.equals(n.data);  
    }  
    ...  
}
```

Works if the type of obj
is Node<Elephant>
or Node<String> or
...

Leave it to here to “do the
right thing” if **this** and **n**
differ on element type



Generics and arrays

```
public class Foo<T> {  
    private T aField;           // ok  
    private T[] anArray;       // ok  
  
    public Foo() {  
        aField = new T();      // compile-time error  
        anArray = new T[10];   // compile-time error  
    }  
}
```

- You cannot create objects or arrays of a parameterized type
 - type info is not available at runtime

Necessary array cast

```
public class Foo<T> {
    private T aField;
    private T[] anArray;

    @SuppressWarnings("unchecked")
    public Foo(T param) {
        aField = param;
        anArray = (T[]) new Object[10];
    }
}
```

You *can* declare variables of type **T**, accept them as parameters, return them, or create arrays by casting **Object[]**

- casting to generic types is not type-safe (hence the warning)
- Effective Java: use **ArrayList** instead

Generics clarify your code

```
interface Map {  
    Object put(Object key, Object value);  
  
    ...  
}
```

plus casts in client code
→ possibility of run-time errors

```
interface Map<Key, Value> {  
    Value put(Key key, Value value);  
  
    ...  
}
```

- Generics always make the client code prettier and safer
- Generics usually clarify the *implementation*
 - (but sometimes uglify: wildcards, arrays, instantiation)

Tips when writing a generic class

- Think through whether you **really need** to make it generic
 - if it's not really a container, most likely a *mistake*
- Start by writing a concrete instantiation
 - get it correct (testing, reasoning, etc.)
 - consider writing a second concrete version
- Generalize it by adding type parameters
 - think about which types are the same or different
 - the compiler will help you find errors
- It will become easier with practice to write generic from the start

Before next class...

1. Start on [Prep. Quiz: HW6](#)
 - Review of the concepts we've seen this quarter
 - A bit longer than what we normally give you
2. Read over spec for [HW6](#) and do [answers-hw6.txt](#) early
 - Implement your specification from HW5
 - Can be tricky!
 - Probably shouldn't use generics yet (we do this in HW7)