Parametric Polymorphism and Generics

Polymorphism

- Subtype polymorphism
  - What we discussed… Code can use a subclass **`B`** where a superclass **`A`** is expected
  - E.g., Code **`A a; … a.m()`** is "polymorphic", because **`a`** can be of many different types at runtime: it can be a **`A`** object or an **`B`** object. Code works with **`A`** and with **`B`** (with some caveats!)
  - Standard in object-oriented languages

Liskov Substitution Principle Rules for Subtype Polymorphism

http://www.ckode.dk/programming/solid-principles-part-3-liskovs-substitution-principle/

- Contravariance of method arguments in the subtype. (Parameter types of A.m may be replaced by supertypes in subclass B.m.)
  - Java doesn't allow this in overrides only in overloads
  - Java override method arguments must be invariant
- Covariance of return types in the subtype. (Return type of A.m may be replaced by subtype in subclass B.m)
- No new exceptions should be thrown, unless the exceptions are subtypes of exceptions thrown by the parent.
- Preconditions cannot be strengthened in the subtype. (You cannot require more than the parent)
- Postconditions cannot be weakened in the subtype. (You cannot guarantee less than the parent)
- Invariants must be preserved in the subtype.
- History Constraint – the subtype must not be mutable in a way the supertype wasn't. For instance MutablePoint cannot be a subtype of ImmutablePoint without violating the History Constraint (as it allows mutations which its supertype didn't)

How does a Method Call Execute?

- For example, x.foo(5);
- Compile time
  - Determine what class to look in – compile time class
  - Determine the method signature (method family)
  - Find all methods in the class with the right name
    - Includes *inherited* methods
      - Look for overrides
      - Return type may be a subtype
        - The types of the actual arguments (e.g. 5 has type int above) must be the same or subtypes of the corresponding formal parameter type
    - Keep only methods that are accessible
      - E.g. a private method is not accessible to calls from outside the class
    - Keep track of the method's signature (argument types) for run-time

How does a Method Call Execute?

- Run time
  - Determine the run-time type of the *receiver*
    - x in this case
    - Look at the object in the heap to find out what its run-time type is
  - Locate the method to invoke
    - Starting at the run-time type, look for a method with the right name and argument types found statically, i.e. method family
      - The types of the actual arguments may be the same or subtypes of the corresponding formal parameter type
    - If it is found in the run-time type, invoke it.
    - Otherwise, continue the search in the superclass of the run-time type
    - Look only at family members
    - This procedure will always find a method to invoke, due to the checks done during static type checking

Parametric Polymorphism

- Overloading is typically referred to as “ad-hoc” polymorphism

- Parametric polymorphism
  - Code takes a type as a parameter
  - Implicit parametric polymorphism
  - Explicit parametric polymorphism

Implicit Parametric Polymorphism

- Python - There is no explicit type parameter(s).

Code is "polymorphic" because it works with
many different types. E.g.:

```
def intersect(sequence1, sequence2):
  result = [ ]
  for x in sequence1:
    if x in sequence2:
      result.append(x)
  return result
```

- As long as sequence1 and sequence2 are of some iterable type, **`intersect`** works!

Implicit Parametric Polymorphism

- How does this differ from subtype polymorphism?
  - Subtype polymorphism is static, this is dynamic.
  - Subtype polymorphism requires declared types for seq1 and seq2, which are iterable types.
  - Subtype polymorphism guarantees that every subclass of the declared type will be iterable!
  - Java subtyping guarantees that when intersect is called, the runtime seq1 and seq2 will implement iteration!
  - With subtype polymorphism the compiler prevents calling intersect(2,2) for example.
- In contrast, Python is dynamic.
  - There is no “common iterable supertype” for the types that seq1 and seq2 may take at runtime. seq1 and seq2 can be anything.
  - They can be completely unrelated types.
  - Errors are dynamic.
  - If someone calls intersect(2,2), intersect will start to execute, and bomb at “for x in sequence1” with a runtime error.

Implicit Parametric Polymorphism

- In Python, Perl, Scheme, other dynamic languages
- There is no explicit type parameter(s); the code works with many different types

- Usually, there is a single copy of the code, and all type checking is delayed until runtime
  - If the arguments are of type as expected by the code, code works
  - If not, code issues a type error at runtime

Explicit Parametric Polymorphism

- C++, Java, C#
  - Explicit parametric polymorphism
- There is an explicit type parameter(s)
- Explicit parametric polymorphism is also known as genericity
- E.g. in C++ we have templates:

```
template<class V>
class list_node {
  list_node<V>* prev;
  …
}
```

```
template<class V>
class list {
  list_node<V> header;
  …
}
```

Explicit Parametric Polymorphism

- Instantiating classes from previous slide with **`int`**:

```
typedef list_node<int> int_list_node;
typedef list<int> int_list;
```

- Usually templates are implemented by creating multiple copies of the generic code, one for each concrete type argument, then compiling
- Problem: if you instantiate with the "wrong" type argument, C++ compiler gives us long, cryptic error messages referring to the generic (templated) code in the STL :)

Explicit Parametric Polymorphism

- Java generics work differently from C++ templates: more type checking on generic code
- OO languages usually have both: subtype polymorphism
  - through inheritance: A extends B or A implements B
- and explicit parametric polymorphism
  - referred to as generics or templates

Using Java Generics

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

AType is the type argument. We instantiated generic (templated) class **ArrayList** with concrete type argument **AType**

```
List<String> names = new ArrayList<String>();
names.add("Ana");
names.add("Katarina");
String s = names.get(0);  // what happens here?
Point p = names.get(0); // what happens here?
Point p = (Point) names.get(0);  // what happens?
```

Defining a Generic Class

```
class MySet<T> {
  // rep invariant: non-null,
  // contains no duplicates
  List<T> theRep;
  T lastLookedUp;
}
```

Declaration of type parameter

Uses of type parameter

```
MySet<String> s;
MySet<Integer> intSet;
MySet<int> i; // compiler error, doesn't autobox
```

Defining a Generic Class

```
// generic (templated, parameterized) class
public class Name<TypeVar, … TypeVar> {
```

- Convention: TypeVar is 1-letter name such as **`T`** for Type,
  `E` for Element, **`N`** for Number, **`K`** for Key, **`V`** for Value
- Class code refers to the type parameter
  - E.g., **`E`**
- To instantiate a generic class, client supplies type arguments
  - E.g., **`String`** as in **`List<String> name;`**
  - Think of it as invoking a "constructor" for the generic class

Example: a Generic Interface

```
// Represents a list of values
public interface List<E> {
  public void add(E value);
  public void add(int index, E value);
  public E get(int index);
  public int indexOf(E value);
  public boolean isEmpty();
  public void remove(int index);
  public void set(int index, E value);
  public int size();
}

public class ArrayList<E> implements List<E> {

public class LinkedList<E> implements List<E> {
```

Generics Clarify Your Code

- Without generics
  - This is known as “pseudo-generic containers”

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

Client code:

```
Map nodes2neighbors = new HashMap();
String key = …
HashSet value = …
nodes2neigbors.put(key, value);
HashSet neighbors = (HashSet) nodes2neighbors.get(key);
```

Casts in client code. Clumsy. If client mistakenly puts non-HashSet value in map,

ClassCastException at this point.

Generics Clarify Your Code

- With generics

```
interface Map<K,V> {
    V put(K key, V value);
    V get(K key);
}
```

Client code:

```
Map<String,HashSet<String>> nodes2neighbors =
     new HashMap<String,HashSet<String>>();
String key = …
nodes2neigbors.put(key,value);
HashSet<String> neighbors =
                        nodes2neighbors.get(key);
```

No casts. Compile-time checks prevent client from using non-HashSet value.

Aside: Why not <int>?

- Java generics are implemented by using type erasure for backward compatibility
- Anything used as a generic must be convertible to Object

```
public class Container<T> {
    private T data;
    public T getData() {
        return data;
    }
}
```

will be seen at runtime as,

```
public class Container {
    private Object data;
    public Object getData() {
        return data;
    }
}
```

Aside: Why not <int>?

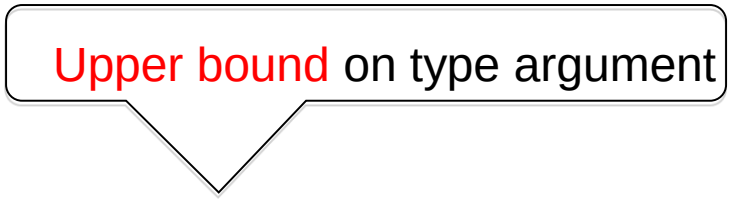
- Compiler provides proper casts to ensure type safety.

```
Container<Integer> val = new Container<Integer>();
Integer data = val.getData()

will become

Container val = new Container();
Integer data = (Integer) val.getData()
```

Type Erasure

- Replace all type parameters in generic types with their bounds (extends or super) or Object if the type parameters are unbounded.
  - The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary, to preserve type safety.
- Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.
  - Contrast with C++ where a new type is created for each instantiation of generic type

Bounded Types Restrict Instantiation by Client




```
interface MyList2<E extends Number> { … }
```

MyList2 can be instantiated only with type arguments that are **Number** or subtype of **Number**

```
MyList2<Number> // OK
MyList2<Integer> // OK
MyList2<Date>    // what happens here?
```

Why Bounded Types?

- Generic code can perform operations permitted by the bound

```
class MyList1<E extends Object>
  void m(E arg) {
    arg.intValue();//compile-time error; Object
                   //does not have intValue()
  }
}

class MyList2<E extends Number>
  void m(E arg) {
    arg.intValue();//OK. Number has intValue()
  }
}
```

Upper Bounded Type Parameters

<Type extends SomeType>

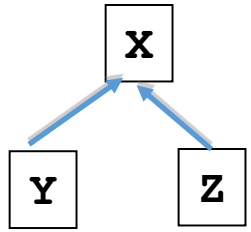
- An upper bound, type argument can be **SomeType** or any of its subtypes

- The upper bound on a type parameter has three effects:
  - ***Restricted Instantiation.*** The upper bound restricts the set of types that can be used for instantiation of the generic type.
    - <T extends Number> means T can be instantiated as a Number or an Integer
  - ***Access To Non-Static Members.*** The upper bound gives access to all public non-static methods and fields of the upper bound.
    - class Box<T extends Number> {...} we can invoke all public non-static methods defined in class Number , such as intValue()
  - ***Type Erasure.*** The leftmost upper bound is used for type erasure and replaces the type parameter in the byte code.
    - class Box<T extends Number> {...} all occurrences of T would be replaced by the upper bound Number

Lower Bounded Type Parameters?

- Why doesn't Java allow
  - class Box<T **super** Number> {...} ?
- ***Access To Non-Static Members***. A lower type parameter bound does not give access to any particular methods beyond those inherited from class Object .
  - `Box<T super Number>` the supertypes of Number have nothing in common, except that they are reference types and therefore subtypes of Object.
- ***Type Erasure*** would replace all occurrences of the type variable T by type Object , and not by its lower bound.
  - The lower bound would have the same effect as "no bound".
- Bottom line: <T super SomeType> doesn't buy much and leads to confusion

Exercise

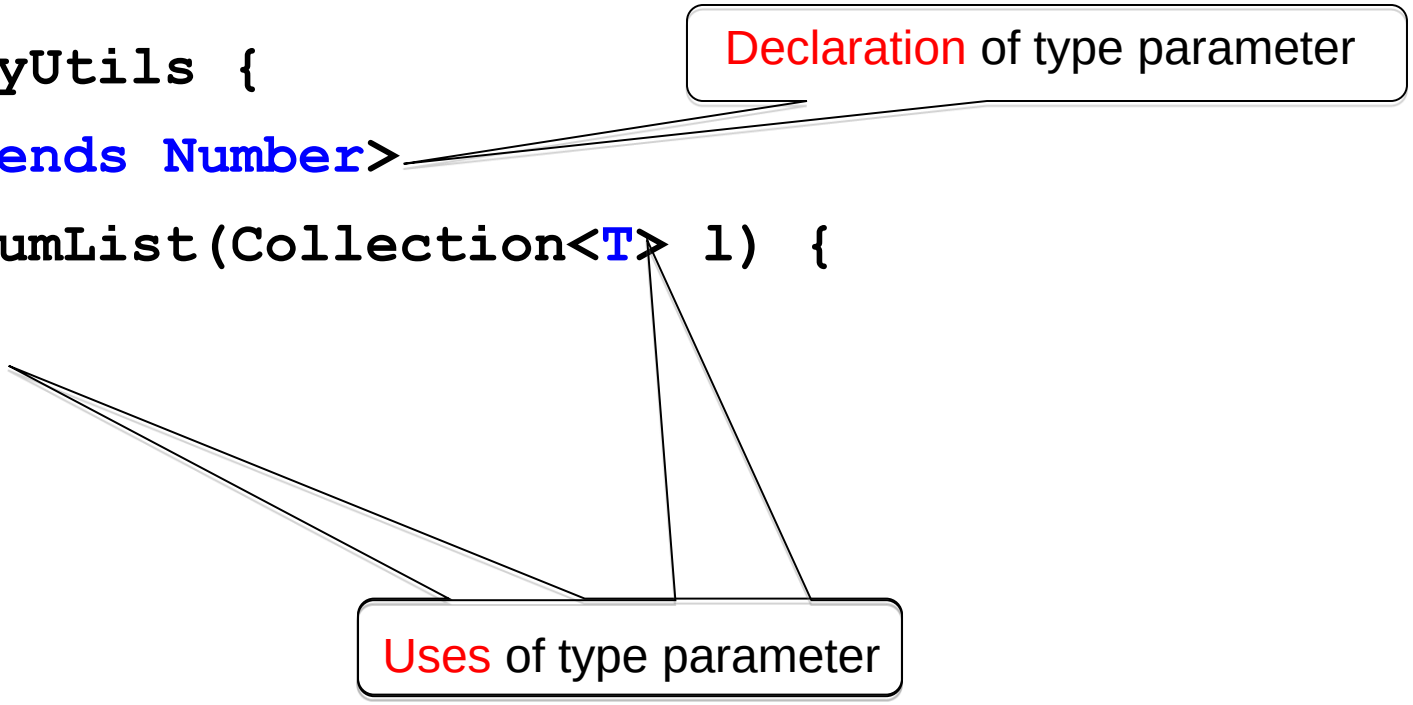


- Given this hierarchy with X, Y and Z:
- What are valid instantiations of generics

```
class A<T extends X> { … } ?
  A<Y>, A<X>, A<Z>
class A<T extends Z> { … } ?
  A<Z>
```

Declaring a Generic Method

```
class MyUtils {
 <T extends Number>
    T sumList(Collection<T> l) {
    …
  }
}
```



Generic Methods

- Generic methods can be called with arguments of different types
- Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately

  - All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).

  - Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

  - The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.

  - A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

```java
public class GenericMethodDemo {
// generic method printArray
        public static < E > void printArray( E[] inputArray ) {
                // Display array elements
                for(E element : inputArray) {
                        System.out.print(element + " ");
                }
                System.out.println();
        }

        public static void main(String args[]) {
        // Create arrays of Integer, Double and Character
                Integer[] intArray = { 1, 2, 3, 4, 5 };
                Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
                Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

                System.out.println("Array integerArray contains:");
                printArray(intArray);   // pass an Integer array

                System.out.println("\nArray doubleArray contains:");
                printArray(doubleArray);   // pass a Double array

                System.out.println("\nArray characterArray contains:");
                printArray(charArray);   // pass a Character array
        }
}
```

Generic Method Example: Sorting

```
public static
 <T extends Comparable<T>>
    void sort(List<T> list) {
     // use of get & T.compareTo<T>
     // T e1 = list.get(…);
     // T e2 = list.get(…);
     // e1.compareTo(e2);
     …
    }
```

We can use T.compareTo<T> because T is bounded by Comparable<T>!

T extends Comparable<T> means that the type parameter must support comparison with other instances of its own type, via the Comparable interface

Comparable is an interface. Many classes implement it.
It has one method, compareTo().

```java
public class GenericMax {
	// determines the largest of three Comparable objects
	public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
		T max = x;   // assume x is initially the largest

		if(y.compareTo(max) > 0) {
			max = y;   // y is the largest so far
		}

		if(z.compareTo(max) > 0) {
			max = z;   // z is the largest now
		}
		return max;   // returns the largest object
	}

	public static void main(String args[]) {
		System.out.printf("Max of %d, %d and %d is %d\n\n",
			3, 4, 5, maximum( 3, 4, 5 ));

		System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
			6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

		System.out.printf("Max of %s, %s and %s is %s\n","pear",
			"apple", "orange",
			maximum("pear", "apple", "orange"));
	}
}
```

Another Generic Method Example

```
public class Collections {
  …
  public static
  <T> void copy(List<T> dst, List<T> src)
  {
    for (T t : src) {
          dst.add(t);
    }
  }
}
```

When you want to make a single (often static) method generic in a class, precede its return type by type parameter(s).

Compiler Ensures Type Compatibility

public static <T> void copy(List<T> dst, List<T> src) {…}

```
List<Number> l1;
List<String> l2;
copy(l1,l2); // compile-time error.
```
Informally, we cannot store Strings into a list of Numbers!.

```
Another example:
List<String> l1;
List<Object> l2;
copy(l1,l2); // compile-time error.
```
We cannot store Objects into a list of Strings.

```
Another example.
List<Object> l1;
List<String> l2;
copy(l1,l2); // Can't do this either.

List<String> l7;
List<String> l8;
copy(l7, l8); // this one will work
```

Bounded Types?

We would like to do this, but it won't compile

```
<T> void copy(List<T2 super T> dst,
              List<T3 extends T> src);
```

Why? We would like:

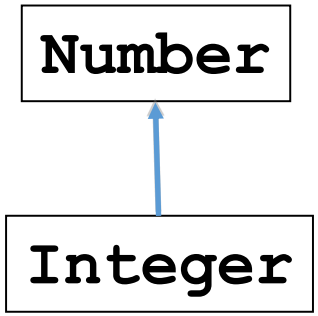
T2 super T means T2 is a supertype of T. T3 extends T means that T3 is a subtype of T. Thus, T3 is a subtype of T2! This makes sense: the destination list should be able to store all elements stored in the source list.

(actually, we can use wildcards ? --- more on this later:

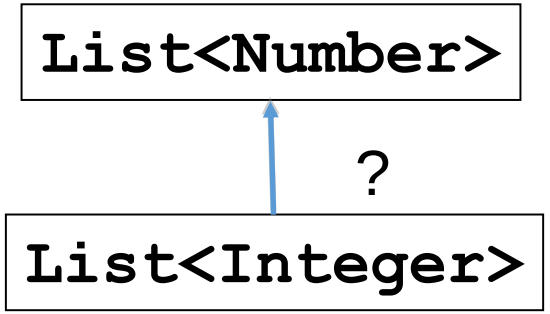
```
<T> void copy(List<? super T> dst,
              List<? extends T> src); )
```

Generics and Subtyping

- **`Integer`** is a subtype of **`Number`**

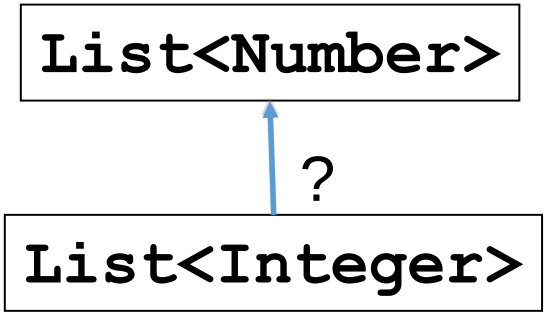


- Is **`List<Integer>`** a subtype of **`List<Number>`**?



Use Function Subtyping Rules to Find Out!

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



- Function subtyping: true subtype must have supertype parameters and subtype return!

What is the Subtyping Relationship Between List<Number> and List<Integer>

- Thus, **`List<Number>`** and **`List<Integer>`** are unrelated through subtyping!

`List<Number>`

`List<Integer>`

Why not?

- Because if Java generics were implicitly polymorphic, you could do this

```
// Cat and Dog are subtypes of Animal

List<Dog> dogs = new ArrayList<Dog>(); // ArrayList implements List
List<Animal> animals = dogs;
animals.add(new Cat());
Dog dog = dogs.get(0); // We just assigned a Cat to a Dog. No one is happy.
```

- We end up trying to assign a Cat to a Dog
- Wildcards restrict access in a type-safe manner

Java Wildcards

- A wildcard is essentially an anonymous type variable
  - Use ? if you'd use a type variable exactly once
- ? appears at the instantiation site of the generic (also called use site)
  - As opposed to declaration site (also called definition site: where type parameter is declared)
  - You can say <? extends E> but not <E extends ?>
- Purpose of the wildcard is to make a library more flexible and easier to use by allowing limited subtyping
  - Wildcards limit the kind of operations that instances of a class can perform

Using Wildcards

This is instantiation of the generic Collection

```
class HashSet<E> implements Set<E> {
    void addAll(Collection<? extends E> c) {
      // What does this give us about c?
      // i.e., what can code assume about c?
      // What operations can code invoke on c?
    }
}
```

- Wildcard appears at instantiations (uses) of generics
- There is also **<? super E>**
- Intuitively, why **<? extends E>** makes sense here
  - The syntax "? extends E" means "some type that either is E or a subtype of E"

Covariance and Wildcards

```
List<Apple> apples = new ArrayList<Apple>();
List<? extends Fruit> fruits = apples;
```

- "? extends" introduces covariant subtyping
- Apple is a subtype of Fruit
- Strawberry is a subtype of Fruit
- List<Apple> is a subtype of List<? extends Fruit>

```
List<Apple> apples = new ArrayList<Apple>();
List<? extends Fruit> fruits = apples;
fruits.add(new Strawberry());
```

- Won't compile
- fruits.add(new Fruit()); won't compile either

Covariance

- the *? extends T* wildcard tells the compiler that we're dealing with a T or a subtype of the type T
  - but *the compiler cannot know which one*.
    - Can lead to problems, if this were allowed
      - fruits.add(new Strawberry()); Apple a = fruits.get(0);
      - This is like the Cat and Dog example
      - We could turn a Strawberry into an Apple
  - Since there's no way to tell, and it needs to guarantee type safety *you won't be allowed to put anything inside such a structure*.
  - Since we know that whichever type it might be, it will be a subtype of T, we can get data out of the structure with the *guarantee* that it will be a T instance
    - Fruit f = fruits.get(0);

Legal Operations on Wildcards

```
Object o;
Number n;
Integer i;
PositiveInteger p; // extends Integer
```

```
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?

~~`lei.add(o);`~~

~~`lei.add(n);`~~

~~`lei.add(i);`~~

~~`lei.add(p);`~~

`lei.add(null);`

`o = lei.get(0);`

`n = lei.get(0);`

`i = lei.get(0);`

~~`p = lei.get(0);`~~

Legal Operations on Wildcards

- The purpose of the wildcard is to allow for subtyping!
- The type declaration List<? extends Integer> means that every List<Type> such that Type extends Integer, is a subtype of List<? extends Integer> (and thus can be used where List<? extends Integer> is expected)
- Covariant subtyping must be restricted to immutable lists. lei can be read but can't be written into (because writing is not safe).
- p = lei.get(0) fails because lei.get(0) can return an Integer. We need a supertype on the left of the '='.
  - Note that PositiveInteger is a fiction.
- The problem with lei.add() is that the compiler doesn't know what type of list lei is at runtime
  - The compiler doesn't know what you will pass at runtime, so it disallows these adds().

Contravariance and Wildcards

- List<? super Apple> fruits;
- fruits is a reference to a List of *something* that is a *supertype* of Apple.
- Apple and any of its subtypes will be assignment compatible with fruits
- These will compile
  - fruits.add(new Apple());
  - fruits.add(new GreenApple());
- This will not
  - fruits.add(new Object());

Contravariance

- Since we cannot know which supertype it is, we aren't allowed to add instances of any supertype.
- What about *getting* data out of such a type?
  - The only thing you can get out of it will be Object instances since we cannot know which supertype it is
  - the compiler can only guarantee that it will be a reference to an Object
    - since Object is the supertype of any Java type.

Legal Operations on Wildcards

```
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>();
```

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);
```

Aside: Covariant vs Contravariant vs Bivariant vs Invariant

A programming language can have features which may support the following subtyping rules:

Covariant

A feature which allows a subtype to replace a supertype

E.g., Java covariant return type

Contravariant

A feature which allows a supertype to replace a subtype

E.g., contravariant argument types

Overloads in Java not overrides

Bivariant

A feature which is both covariant and contravariant.

Invariant

A feature which does not allow any of the above substitutions.

How to Use Wildcards

- Use **`<? extends T>`** when you *get* (read) values from a *producer*
- Use **`<? super T>`** when you *add* (write) values into a *consumer*
- E.g.:

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

- **PECS**: **P**roducer **E**xtends, **C**onsumer **S**uper
- Use neither, just **`<T>`**, if both *add* and *get*

PECS

- Use the ? extends wildcard if you need to retrieve object from a data structure.
- Use the ? super wildcard if you need to put objects in a data structure.
- If you need to do both things, don't use any wildcard.

Type Erasure

- All type arguments become Object or type bounds when compiled
  - Reason: backward compatibility with old bytecode
  - At runtime all generic instantiations have same type

```
List<String> lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass() // works
```

- Cannot use **`instanceof`** to find type argument

```
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) {
  // compile-time error
```

- What happens to **`equals()`** on elements of generic type

Equals for a Generic Class

What happens to **`equals()`** on elements of generic type

At runtime, JVM has no knowledge of type argument. Node<String> is same as Node<Elephant>. instanceof is a compile-time error.

```
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());
  }
}
```

Equals for a Generic Class

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

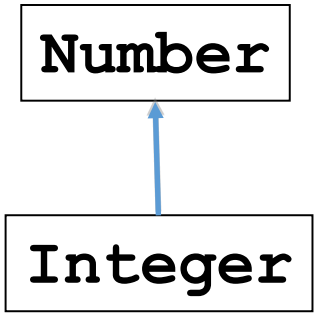
Same here. JVM has no knowledge of type argument. Node<String> will cast to Node<Elephant>. Casting results in a compile-time warning, but not error.

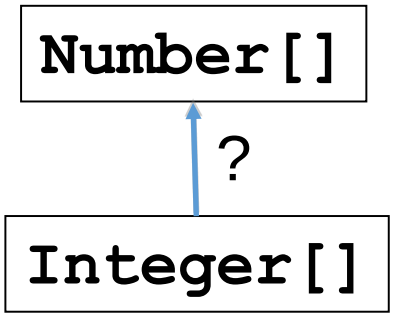
Equals for a Generic Class

Generics for the fix

```
class Node<E> {

    …
        @Override
        public boolean equals(Object obj) {
                If (!(obj instanceof Node<?>))
                        return false;
                Node<E> n = (Node<E>) obj;  // gets a warning
                // Node<?> n = (Node<?>) obj; // compiles
                return this.label.equals(n.label);
        }

}
```

Arrays and Subtyping





- **`Integer`** is subtype of **`Number`**
- Is **`Integer[]`** a subtype of **`Number[]`**
- Use our subtyping rules to find out

(Just like with List<Integer> and List<Number>)

- Again, the answer is NO!
- Different answer in Java: in Java **`Integer[]`** is a Java subtype of **`Number[]`**!
  - The Java subtype is not a true subtype!
  - Known as "problem with Java's covariant arrays"
  - Here, covariant simply means if X is subtype of Y then X[] will also be sub type of Y[]. Arrays are covariant
  - String is subtype of Object So String[] is subtype of Object[]

Why?

- Early versions of Java did not include generics (a.k.a. parametric polymorphism).
  - C# also
- If arrays were not covariant something like
  - boolean equalArrays (Object[] a1, Object[] a2);
  - Would only work with Objects
- You would have to write a separate method for every type combination

Why?

- When generics were introduced, they were purposefully not made covariant to prevent this sort of thing:
  - Cat and Dog are subtypes of animal

```
// Illegal code - because otherwise life would be Bad
List<Dog> dogs = new List<Dog>();
List<Animal> animals = dogs; // Trouble ahead, List<Dog> is not really a subtype of List<Animal>
animals.add(new Cat());
Dog dog = dogs.get(0); // This should be safe, right?
```

- We just assigned a cat to a dog.
- https://stackoverflow.com/questions/18666710/why-are-arrays-covariant-but-generics-are-invariant

Integer[] is a Java subtype of Number[]

```
Number n;
Number[] na;
Integer i;
Integer[] ia;
na[0] = n;
na[1] = i;
n = na[0];
i = na[1]; //what happens?
ia[0] = n; //what happens?
ia[1] = i;
n = ia[0];
i = ia[1];
```

```
ia = na; //what
// happens here?
Double d = 3.14;
na = ia; //OK!
na[2] = d;
i = ia[2]; //what
// happens here?
```

Writing a Generic Class

- Start by writing a concrete class
- Make sure it is correct
  - Reasoning
  - Testing
- Think about writing a second concrete class with different types
    - How would you have to change original class
- Generalize by adding type parameters (generics)
  - Which are the same, which differ
  - Compiler will find errors
- With practice, it gets easier to start with generics

Advice for Generics

- Don’t use raw types
  - Eclipse complains
  - Don’t do this:
    - private final Collection stamps = ... ;
  - don’t use raw iterators
    - for (Iterator i = stamps.iterator(); i.hasNext();)
  - You lose type safety with raw types
- Eliminate Warnings
  - Java compiler is quite good at checking types
  - Always try to eliminate warnings
  - If you can’t, use SupressWarnings on the smallest block possible
  - Add comments
- Prefer Lists to Arrays when possible
  - More type safety

Advice for Generics

- Use generic types rather than fixed types
  - Makes code more flexible
  - Usually easy to make classes generic without affecting client code
  - Use generic methods
- Use bounded wildcards to enhance flexibility