idbs Enabling Science

accu 2012

Variance in Generic Types in Java and C♯

Robert Stanforth

April 28, 2012

Variance in Generic Types in Java and C♯

List<Cat>

Variance in Generic Types in Java and C♯

```
List<? extends Cat>
```

```
List<? super Cat>
```

```
interface List<out T>
{
    …
}
```

```
interface Collector<in T>
{
    …
}
```

Variance in Generic Types in Java and C♯

```
package java.util;
public class Collections {

public static <T>
int  binarySearch(
        List<? extends Comparable<? super T>>  list,
        T  key)
{…}
```

Variance in Generic Types in Java and C♯

```
List<? extends Cat>
```

```
List<? super Cat>
```

- What do they mean?
- What problem do they solve?
- Why do they look so different in Java and C♯?

```
interface List<out T>
{
}
```

```
interface Collector<in T>
{
    …
}
```

Motivation for Covariance

- If an API expects a cat …

```
void writeToXml(Cat cat);
```

- … then we can give it a tiger.

```
Tiger tiger = …;
writeToXml(tiger);
```

Motivation for Covariance

- If an API expects a list of cats …

```
void writeToXml(List<Cat> cats);
```

- … then we’d expect it to accept a list of tigers …

```
List<Tiger> tigers = …;
writeToXml(tigers);
```

Motivation for Covariance

- If an API expects a list of cats …

```
void writeToXml(List<Cat> cats);
```

- … then we'd expect it to accept a list of tigers …

```
List<Tiger> tigers = …;
~~writeToXml(tigers);~~
```

- … but it doesn't.

Why not?

Motivation for Covariance

- Expect an implementation like this …

```
void writeToXml(List<Cat> cats)
{
    foreach (Cat cat in cats)
        process(cat);
}
```

List<Tiger> tigers = …;

~~writeToXml(tigers);~~

Motivation for Covariance

- … but it could be this …

```
void writeToXml(List<Cat> cats)
{
    cats.add(new Lion());
}
```

```
List<Tiger> tigers = …;
~~writeToXml(tigers);~~
```

Motivation for Covariance

- … but it could be this …

```
void writeToXml(List<Cat> cats)
{
    cats.add(new Lion());
}
```

- … which would cause a type violation.

```
List<Tiger> tigers = …;
~~writeToXml(tigers);~~
tigers.get(n).countStripes();
```

Motivation for Covariance

- We want to declare that List<Tiger> subtypes List<Cat>

```
void writeToXml(List<Cat> cats)
{
    foreach (Cat cat in cats)
        process(cat);
}
```

```
void writeToXml(List<Cat> cats)
{
    ~~cats.add(new Lion());~~
}
```

✔

```
List<Tiger> tigers = ...;
writeToXml(tigers);
```

Motivation for Contravariance

- We want to declare that Collector<Cat> subtypes Collector<Tiger>

```
void donateMyTigers(Collector<Tiger> tigerCollector)
{
   foreach (Tiger tiger in this.tigers)
      tigerCollector.accept(tiger);
}
```

```
void donateMyTigers(Collector<Tiger> tigerCollector)
{
   ~~Tiger tiger = tigerCollector.mostRecentItem();~~
   tiger.countStripes();
}
```

```
Collector<Cat> catCollector = …;
catCollector.accept(new Lion());

donateMyTigers(catCollector); ✓
```

Covariance in Traditional Java/C♯

Covariance in Traditional Java/C♯

- Tiger[] convertible to Cat[]

```
void writeToXml(Cat[] cats)
{
    for (int i = 0; i < cats.Length; i ++)
        process(cats[i]);
}
```

```
void writeToXml(Cat[] cats)
{
    cats[0] = new Lion();
}
```

```
Tiger[] tigers = …;
writeToXml(tigers);
```

✓

Covariance in Traditional Java/C♯

- Tiger[] convertible to Cat[]
- Not statically type-safe – check at run-time

```
void writeToXml(Cat[] cats)
{
    for (int i = 0; i < cats.Length; i ++)
        process(cats[i]);
}
```

```
void writeToXml(Cat[] cats)
{
    cats[0] = new Lion();
}
```

ArrayTypeMismatchException

```
Tiger[] tigers = …;
writeToXml(tigers);
```

✓

Covariance in Traditional Java/C♯

- Tiger[] convertible to Cat[]
- Not statically type-safe – check at run-time

```
void writeToXml(Cat[] cats)
{
    for (int i = 0; i < cats.Length; i ++)
        process(cats[i]);
}
```

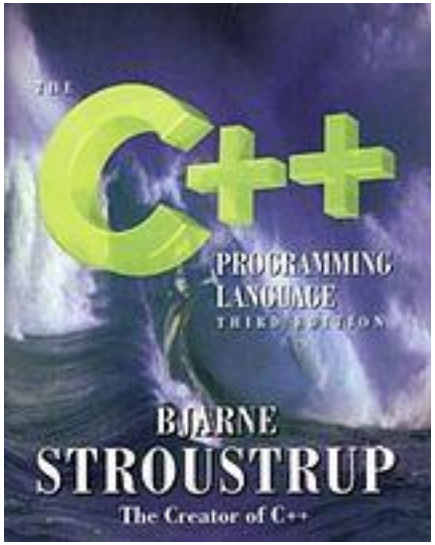
```
void writeToXml(Cat[] cats)
{
    cats[0] = new Lion();
}
```

ArrayTypeMismatchException

ArrayStoreException

```
Tiger[] tigers = …;
writeToXml(tigers);
```

✓



Covariance in C++

Covariance in C++

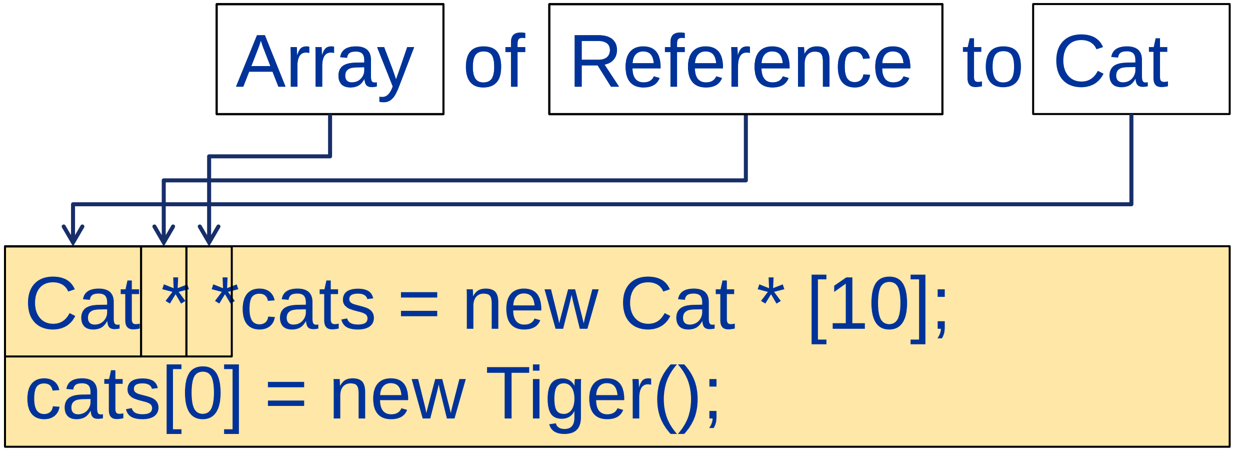
- C++ templates are always non-variant

```
std::list<Tiger *>
```

```
std::list<Cat *>
```

Covariance in C++

- Pointers to pointers



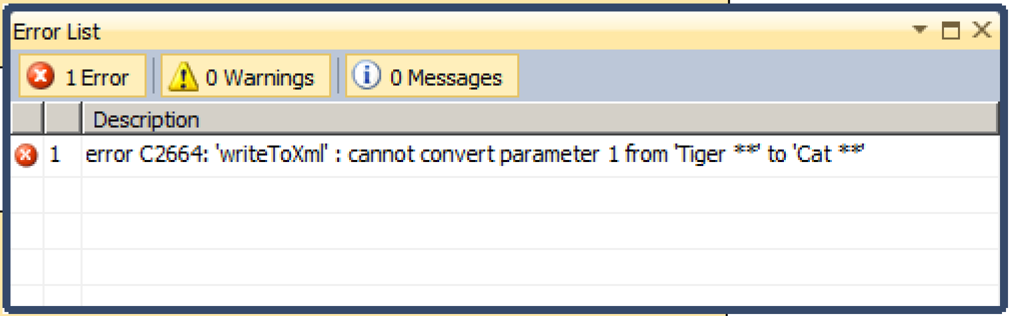
Array of Reference to Cat

```
Cat * *cats = new Cat * [10];
cats[0] = new Tiger();
```

Covariance in C++

- ‘T * *’ is not covariant

```
void writeToXml(Cat * *cats, size_t num)
{
    cats[0] = new Lion();
}
```



Error List

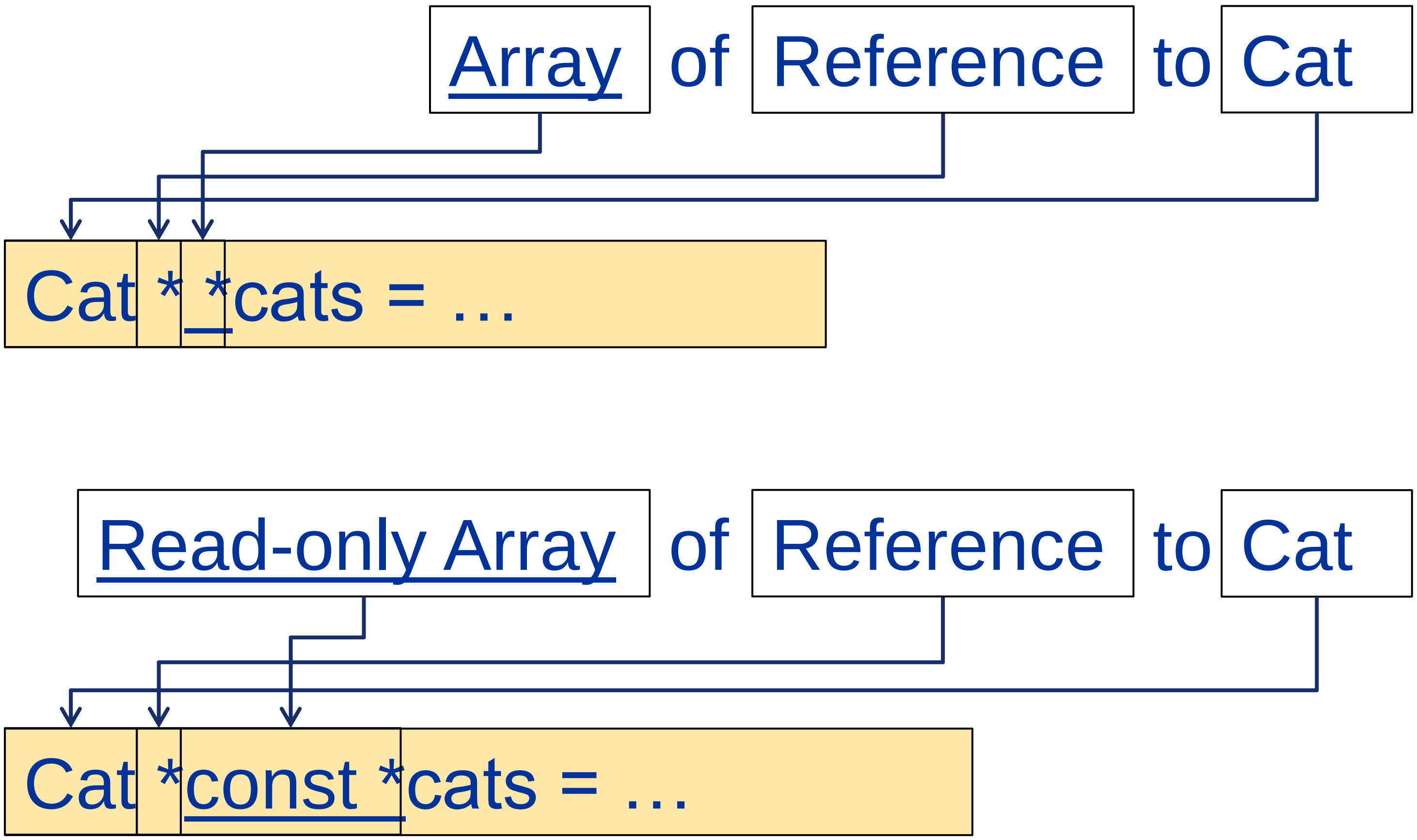
1 Error | 0 Warnings | 0 Messages

Description

1 error C2664: 'writeToXml' : cannot convert parameter 1 from 'Tiger **' to 'Cat **'

```
Tiger * *tigers = …;
~~writeToXml(tigers, numTigers);~~
tigers[0]->countStripes();
```

Covariance in C++



Array of Reference to Cat

```
Cat * *cats = …
```

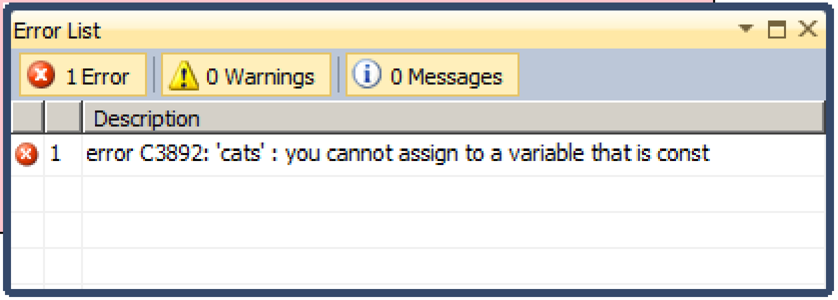
Read-only Array of Reference to Cat

```
Cat *const *cats = …
```

Covariance in C++

- ‘T *const *’ should be covariant …

```
void writeToXml(Cat *const *cats, size_t num)
{
    ~~cats[0] = new Lion();~~
}
```



```
1    error C3892: 'cats' : you cannot assign to a variable that is const
```

```
Tiger *const *tigers = …;
writeToXml(tigers, numTigers); ✓
tigers[0]->countStripes();
```

Covariance in C++

- ‘T *const *’ should be covariant …

```
void writeToXml(Cat *const *cats, size_t num)
{
    for (size_t  j = 0; j < num; j ++)
    {
        Cat *cat = cats[j];
        process(cat);
    }
```

```
Tiger *const *tigers = …;
writeToXml(tigers, numTigers);
```

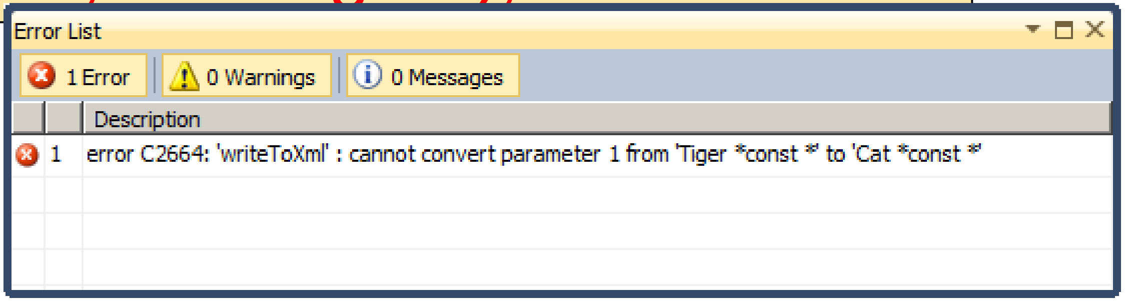
Covariance in C++

- ‘T *const *’ should be covariant …

```
void writeToXml(Cat *const *cats, size_t num);
```

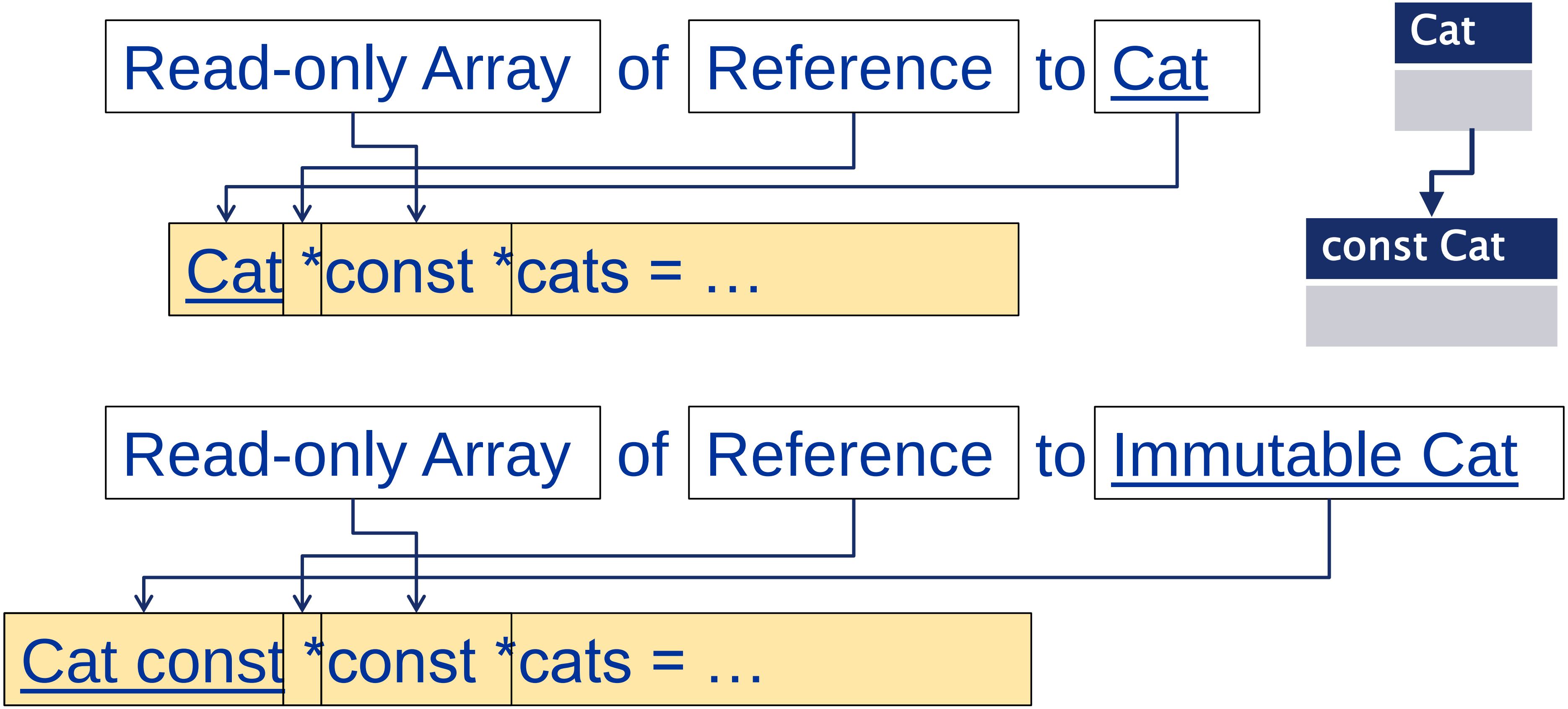
- … but it isn’t.

```
Tiger *const *tigers = …;
~~writeToXml(tigers, numTigers);~~
```



1 error C2664: 'writeToXml' : cannot convert parameter 1 from 'Tiger *const *' to 'Cat *const *'

Covariance in C++



Read-only Array of Reference to Cat

```
Cat *const *cats = …
```

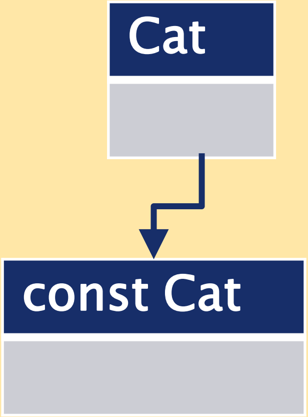
Read-only Array of Reference to Immutable Cat

```
Cat const *const *cats = …
```

Covariance in C++

- ‘T *const *’ is covariant w.r.t. const/volatile…

```
void writeToXml(Cat const *const *cats, size_t num)
{
    for (size_t  j = 0; j < num; j ++)
    {
        Cat const *cat = cats[j];
        cat.serialise(std::cout);
    }
```



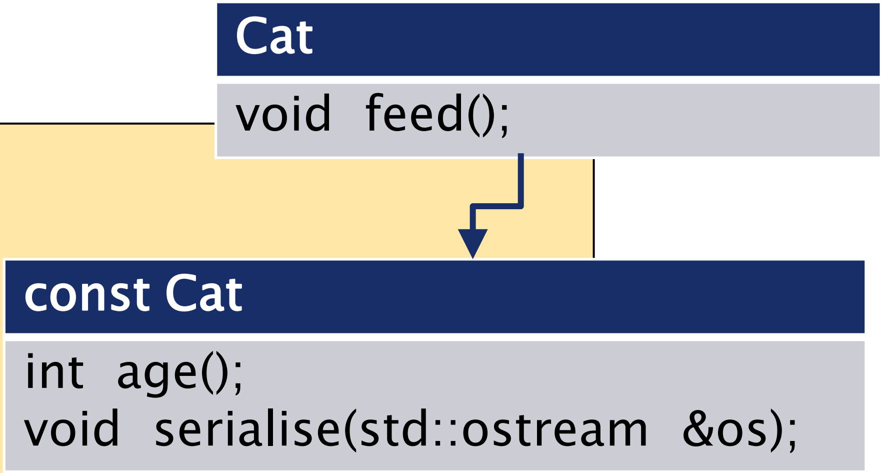
```
Cat *const *cats = …;
writeToXml(cats, num);
```

Covariance in C++

```
class Cat : public Animal
{
public:

    int  age()  const = 0;
    void  serialise(std::ostream &)  const = 0;

    void  feed() = 0;
};
```



Cat

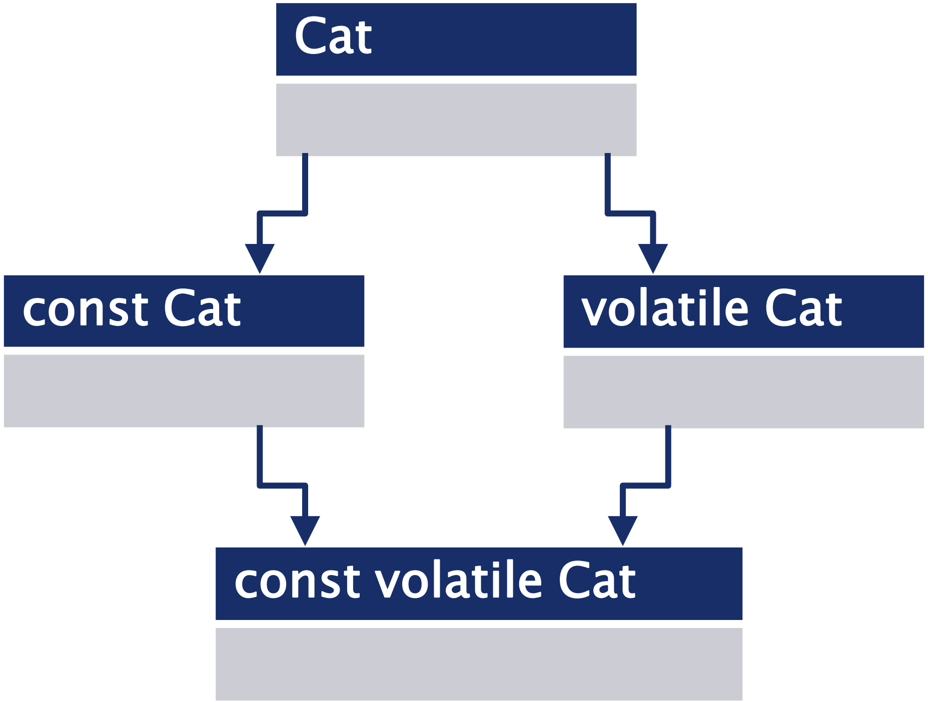
void feed();

const Cat

int age();
void serialise(std::ostream &os);

Covariance in C++

- The limit of its extensibility:



Cat

const Cat

volatile Cat

const volatile Cat

A Type System with Variance

A Type System with Variance

- Requirements:
    - List<Tiger> is a sub-type of List<Cat>
    - Type-safe

```
List<Cat> cats = …;

a = cats.f(b); // cats may be List<Tiger>.
```

- Deduce conditions on List, for this to be valid

Method Variance

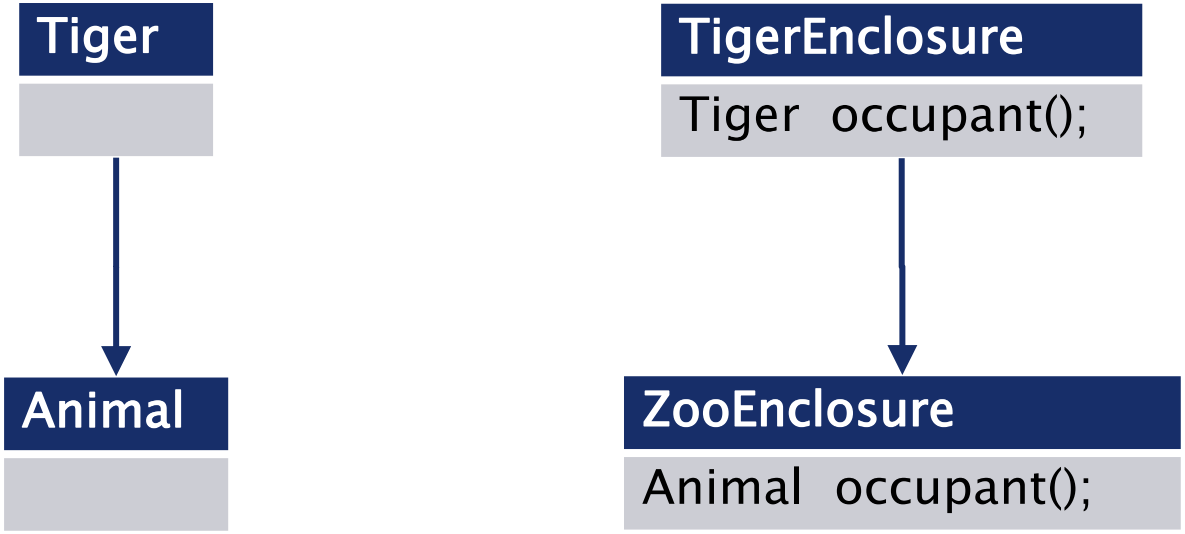
```
List<Cat> cats = …;

try {
    a = cats.f(b); // cats may be List<Tiger>
} catch (E) {…}
```

- Invoking a method without complete knowledge of its run-time implementation's:
  - Return type
  - Argument types
  - Checked exception types

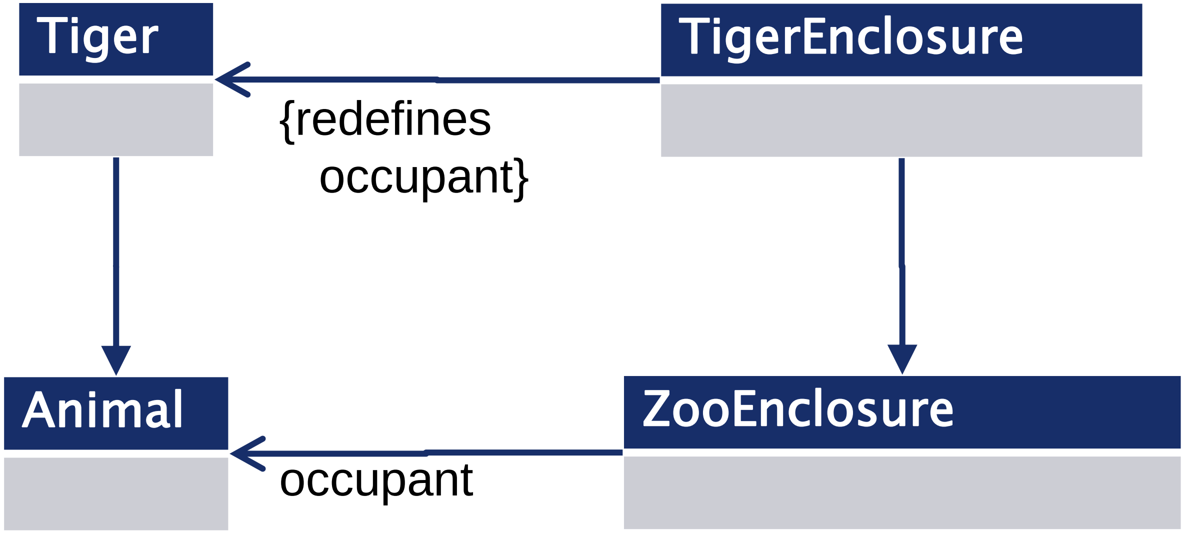
Method Variance

- Precedent: Covariant return type



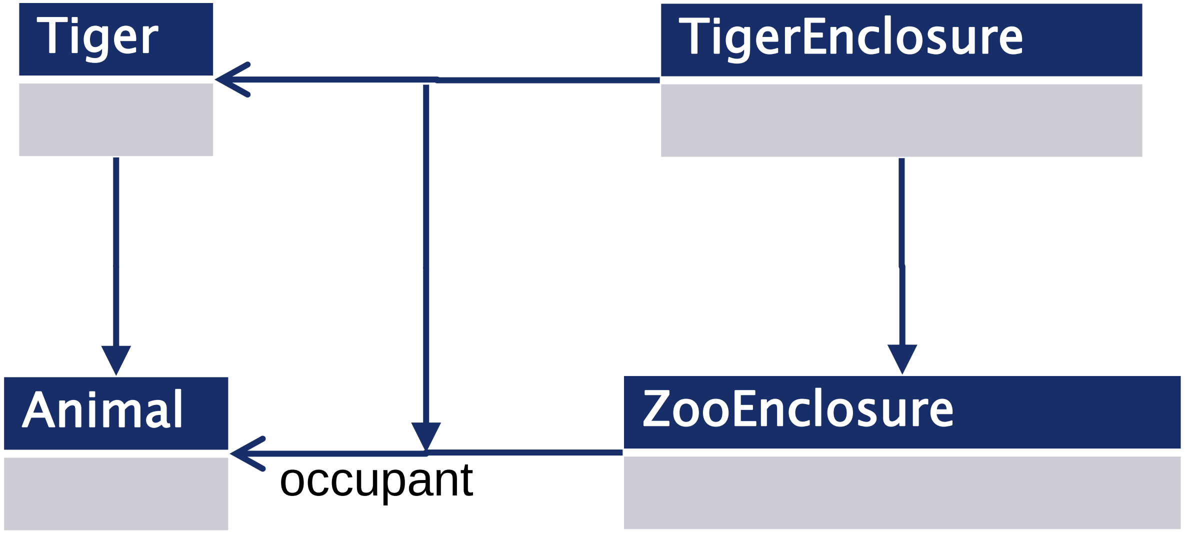
Method Variance

- Covariant return type in UML



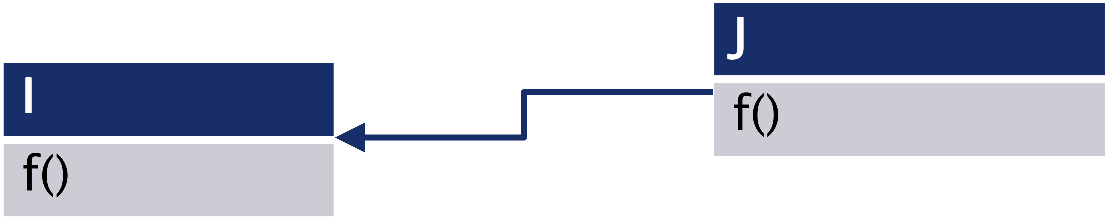
Method Variance

- Covariant return type in UML
  - Association specialisation



Method Variance

- Under-promise
- Over-deliver



J

f()

I

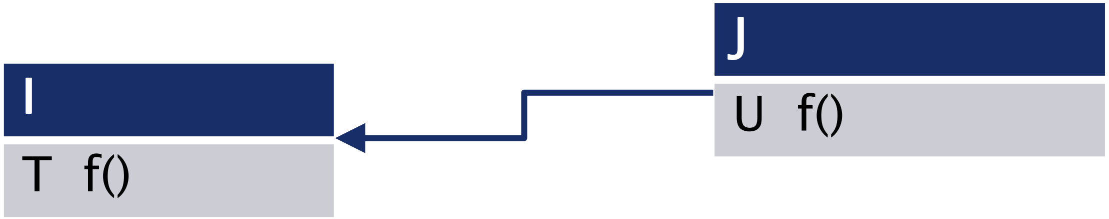
f()

```
I i = …;

try {
    a = i.f(b); // i may be a J.
} catch (E) {…}
```

Covariance of Return Type

- Under-promise
- Over-deliver



```
I i = ...;

T a = i.f();
```

if

J <: I

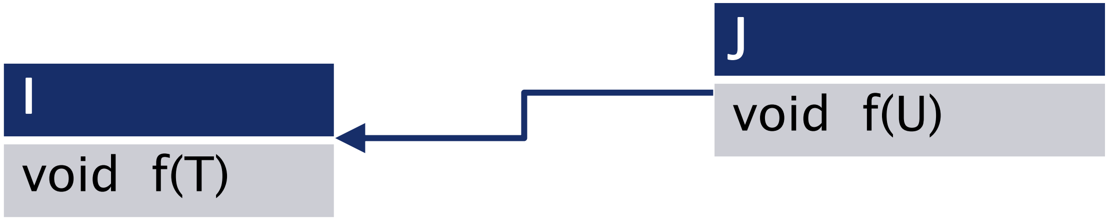
then

ret-type(J.f) <: ret-type(I.f)

ret-type(J.f) covariant w.r.t. J

Contravariance of Argument Type

- Under-promise
- Over-deliver



```
I i = …;

T b = …;
i.f(b);
```

if

$J <: I$

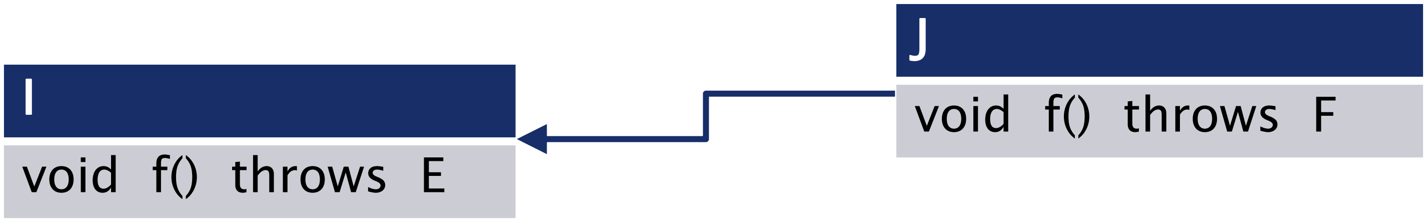
then

$\text{arg-type}_n(J.f) :> \text{arg-type}_n(I.f)$

$\text{arg-type}_n(J.f)$ contravariant w.r.t. J

Covariance of Checked Exception Type

- Under-promise
- Over-deliver



```
I i = …;

try {
    i.f();
} catch (E) {…}
```

if

  J <: I

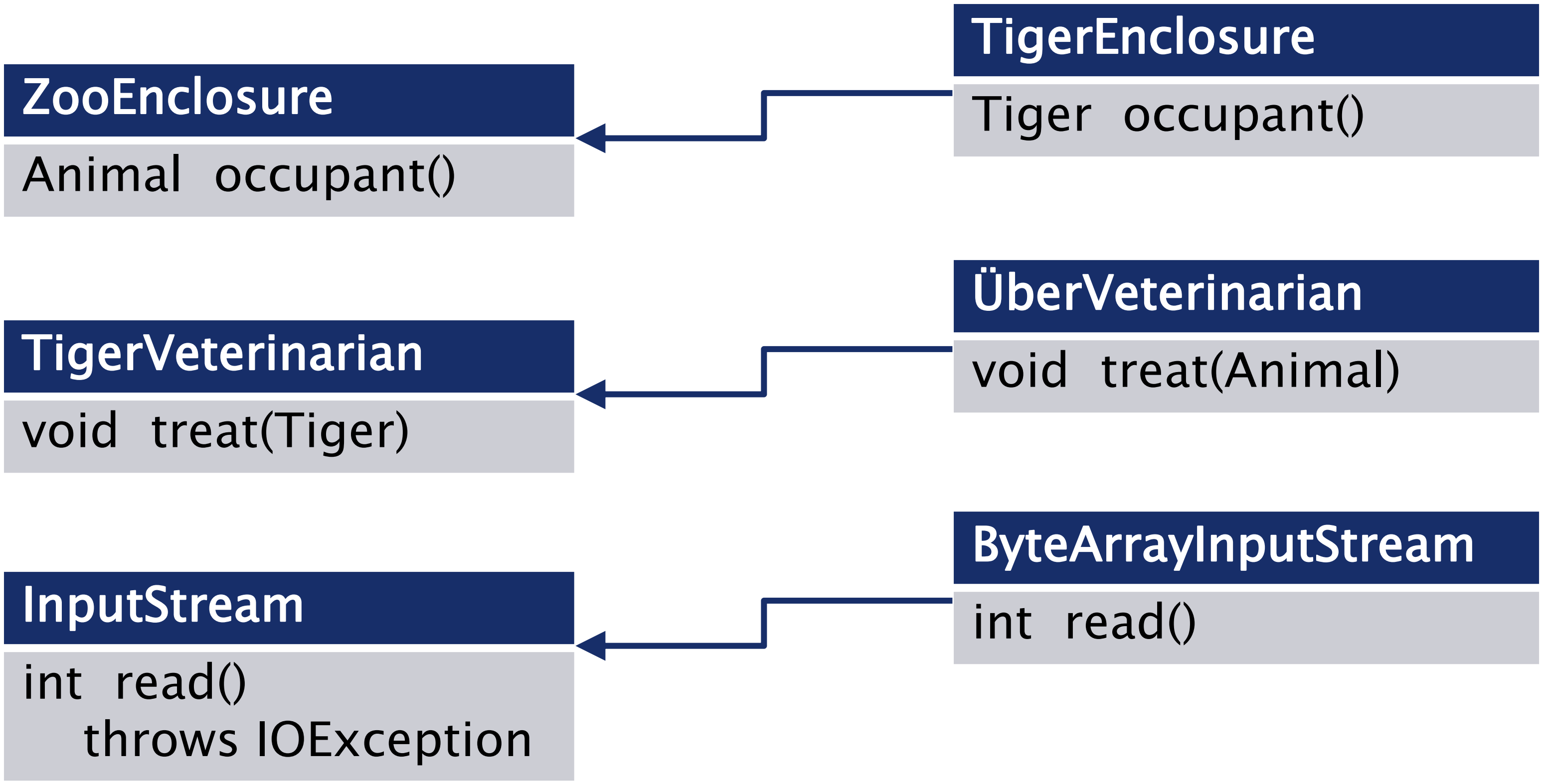
then

  throws-type(J.f) <: throws-type(I.f)

throws-type(J.f) covariant w.r.t. J

Method Variance Examples

- Under-promise
- Over-deliver



ZooEnclosure

Animal occupant()

TigerEnclosure

Tiger occupant()

TigerVeterinarian

void treat(Tiger)

ÜberVeterinarian

void treat(Animal)

InputStream

int read()
throws IOException

ByteArrayInputStream

int read()

Back to Generics …

Covariant Interface

- List<T> covariant w.r.t. T
- For each method f:

```
if
  U <: T
then
  List[U] <: List[T]
so
  ret-type(List[U].f) <: ret-type(List[T].f)
```

```
ret-type(List[T].f)  covariant  w.r.t.  T
```

Covariant Interface

- List<T> covariant w.r.t. T
- For each method f, for the *n*th argument:

```
if
  $U <: T$
then
  $List[U] <: List[T]$
so
  $\text{arg-type}_n(List[U].f) :> \text{arg-type}_n(List[T].f)$
```

$\text{arg-type}_n(\text{List}[T].f)$ contravariant w.r.t. T

Covariant Interface

```
When declaring List[T] to be covariant w.r.t. T:

for each method f in List
    ret-type(f)  covariant  w.r.t.  T
    arg-type$_n$(f)  contravariant  w.r.t.  T

for each immediate super-type J of List[T]
    J  covariant  w.r.t.  T
```

Covariant Interface

```
for each method f in List
    ret-type(f)  covariant  w.r.t.  T
    arg-type$_n$(f)  contravariant  w.r.t.  T

for each immediate super-type J of List[T]
    J  covariant  w.r.t.  T
```

```
interface  List<co  T>
{
    int  size();
    T  elementAt(int  index);
    List<T>  clone();
    List<List<T>>  chunks(int  chunkSize);
    void  sendTo(Collector<T>  collector);
    void  set(int  index,  ~~T~~  value);
    void  appendAll(~~List<T>~~  values);
    ~~Collector<T>~~  appender();
}
```

Contravariant Interface

- Collector<T> contravariant w.r.t. T
- For each method f:

if

U <: T

then

Collector[U] :> Collector[T]

so

ret-type(Collector[U].f) :> ret-type(Coll’r[T].f)

ret-type(Collector[T].f) contravariant w.r.t. T

Contravariant Interface

- Collector<T> contravariant w.r.t. T
- For each method f:

```
if
  U <: T
then
  Collector[U] :> Collector[T]
so
  arg-type_n(Collector[U].f) <: arg-type_n(Coll'r[T].f)
```

$\text{arg-type}_n(\text{Collector}[T].f)$ covariant w.r.t. T

Contravariant Interface

When declaring Collector[T] *to be contravariant w.r.t. T:*

for each method f *in* Collector
    ret-type(f) contravariant w.r.t. T
    arg-type$_n$(f) covariant w.r.t. T

for each immediate super-type J *of* Collector[T]
    J contravariant w.r.t. T

Contravariant Interface

```
interface  Collector<contra  T>
{
   void  accept(T  item);
   void  accept(List<T>  items);
   Collector<T>  clone();
   void  donate(~~Collector<T>~~  collector);
   ~~T~~  mostRecentItem();
   ~~List<T>~~  allItems();
}
```

for each method f *in* Collector
   ret-type(f) contravariant w.r.t. T
   arg-type$_n$(f) covariant w.r.t. T

for each immediate super-type J *of* Collector[T]
   J contravariant w.r.t. T

Implementation in C♯

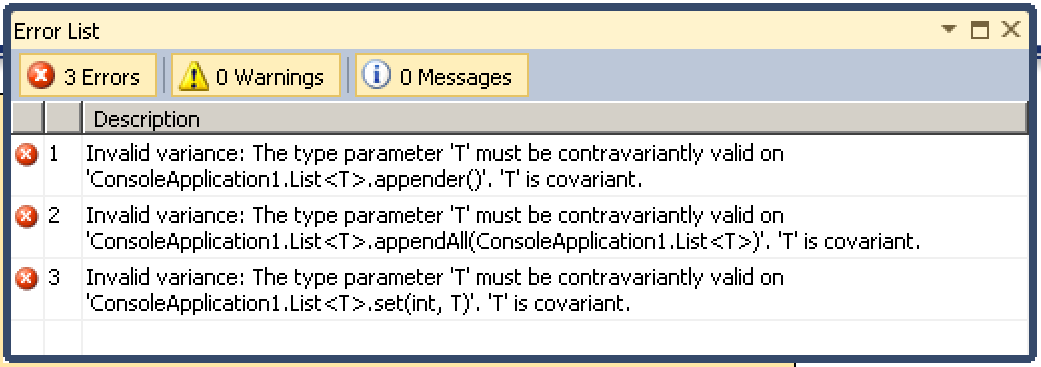
Implementation in C♯

```
interface  IEnumerator<out  T>
{
   T  Current  { get; }
   void  MoveNext();
}
```

```
interface  IComparable<in  T>
{
   int  CompareTo(T  other);
}
```

Implementation in C♯

```
interface List<out T>
{
    int size();
    T elementAt(int index);
    List<T> clone();
    List<List<T>> chunks(int chunkSize);
    void sendTo(Collector<T> collector);
    void set(int index, ~~T~~ value);
    void appendAll(~~List<T>~~ values);
    ~~Collector<T>~~ appender();
}
```



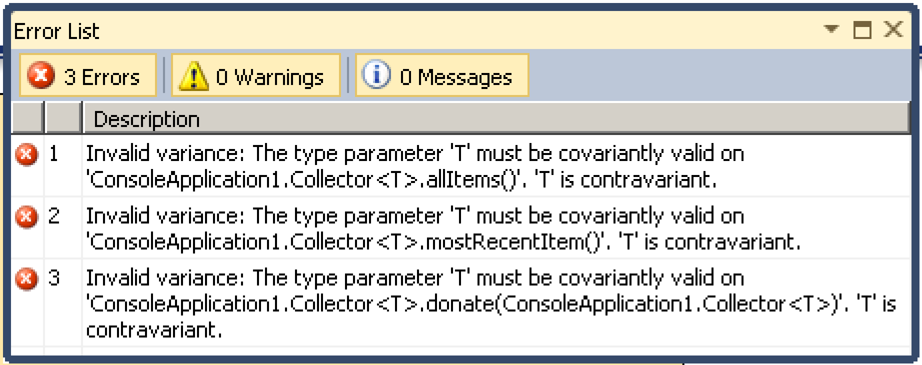
1 Invalid variance: The type parameter 'T' must be contravariantly valid on 'ConsoleApplication1.List<T>.appender()'. 'T' is covariant.

2 Invalid variance: The type parameter 'T' must be contravariantly valid on 'ConsoleApplication1.List<T>.appendAll(ConsoleApplication1.List<T>)'. 'T' is covariant.

3 Invalid variance: The type parameter 'T' must be contravariantly valid on 'ConsoleApplication1.List<T>.set(int, T)'. 'T' is covariant.

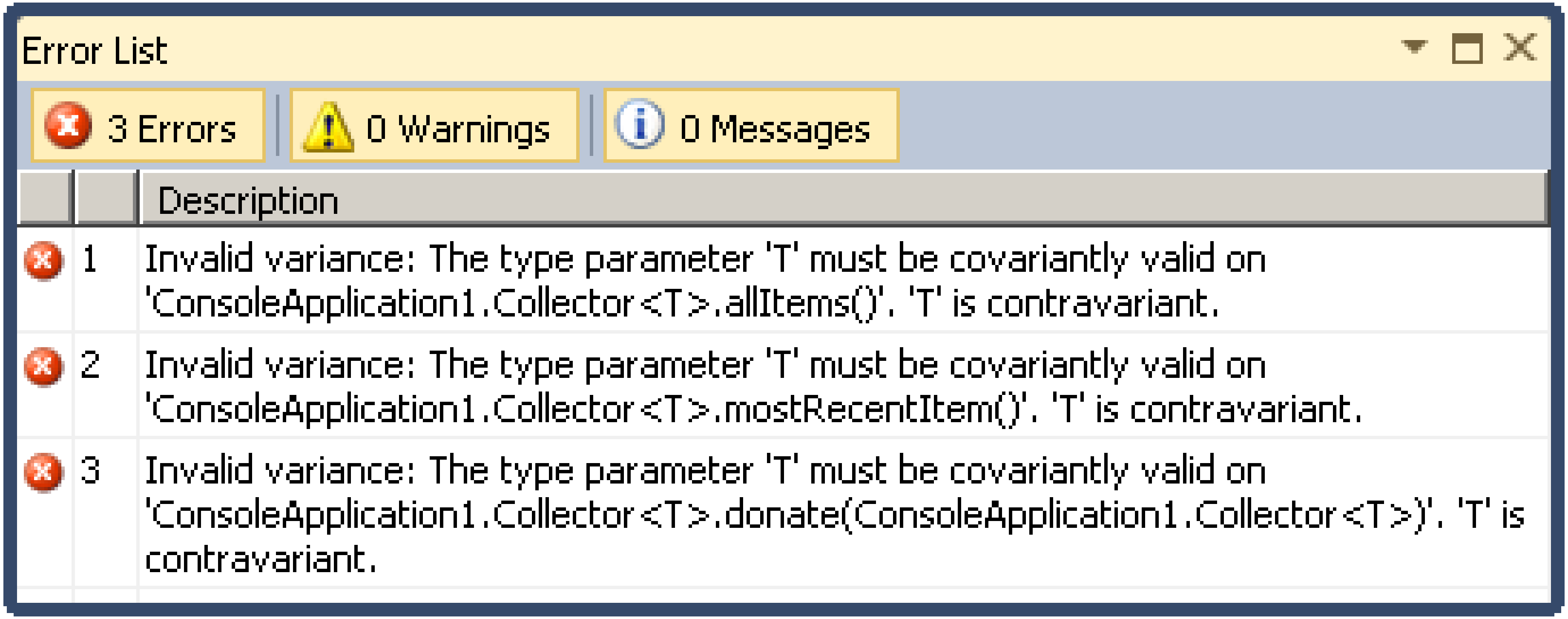
Implementation in C♯

```
interface Collector<in T>
{
    void accept(T item);
    void accept(List<T> items);
    Collector<T> clone();
    void donate(~~Collector<T>~~ collector);
    ~~T~~ mostRecentItem();
    ~~List<T>~~ allItems();
}
```



	Description
1	Invalid variance: The type parameter 'T' must be covariantly valid on 'ConsoleApplication1.Collector<T>.allItems()'. 'T' is contravariant.
2	Invalid variance: The type parameter 'T' must be covariantly valid on 'ConsoleApplication1.Collector<T>.mostRecentItem()'. 'T' is contravariant.
3	Invalid variance: The type parameter 'T' must be covariantly valid on 'ConsoleApplication1.Collector<T>.donate(ConsoleApplication1.Collector<T>)'. 'T' is contravariant.

Implementation in C♯



		Description
☒	1	Invalid variance: The type parameter 'T' must be covariantly valid on 'ConsoleApplication1.Collector<T>.allItems()'. 'T' is contravariant.
☒	2	Invalid variance: The type parameter 'T' must be covariantly valid on 'ConsoleApplication1.Collector<T>.mostRecentItem()'. 'T' is contravariant.
☒	3	Invalid variance: The type parameter 'T' must be covariantly valid on 'ConsoleApplication1.Collector<T>.donate(ConsoleApplication1.Collector<T>)'. 'T' is contravariant.

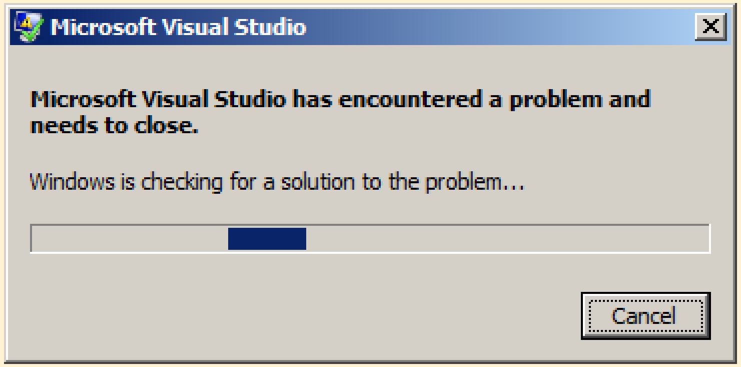
Determinism of Sub-Type

```
interface  N<in  Z>  { }
interface  C  :  N<N<C>>  { }
…
void  f(C  c)
{
   N<C>  nc  =  c;
}
```

Kennedy, A.J., Pierce, B.C. (2006). On Decidability of Nominal Subtyping with Variance, FOOL-WOOD ’07.

Determinism of Sub-Type

```
interface N<in Z> { }
interface C : N<N<C>> { }
…
void f(C c)
{
   N<C> nc = c;
}
```



Microsoft Visual Studio has encountered a problem and needs to close.

Windows is checking for a solution to the problem...

Kennedy, A.J., Pierce, B.C. (2006). On Decidability of Nominal Subtyping with Variance, FOOL-WOOD '07.

Determinism of Sub-Type

interface N[contra Z]
interface C <: N[N[C]]

C <: N[C] ?

Kennedy, A.J., Pierce, B.C. (2006). On Decidability of Nominal Subtyping with Variance, FOOL-WOOD ’07.

Determinism of Sub-Type

interface N[contra Z]
interface C <: N[N[C]]

C <: N[C]
is implied by
N[N[C]] <: N[C]

Kennedy, A.J., Pierce, B.C. (2006). On Decidability of Nominal Subtyping with Variance, FOOL-WOOD ’07.

Determinism of Sub-Type

interface N[contra Z]
interface C <: N[N[C]]

C <: N[C]
is implied by
N[N[C]] <: N[C]
is implied by
N[C] :> C

Kennedy, A.J., Pierce, B.C. (2006). On Decidability of Nominal Subtyping with Variance, FOOL-WOOD ’07.

Implementation in Java

Covariance in Java

- If an API expects a list of cats …

```
void writeToXml(List<Cat> cats);
```

- … then we’d expect it to accept a list of tigers …

```
List<Tiger> tigers = …;
writeToXml(tigers);
```

Covariance in Java

- How can we make the API general?

```
void writeToXml(List<Cat> cats);
```

Covariance in Java

- How can we make the API general?

```
void writeToXml(List<Cat> cats);
void writeToXml(List<Tiger> cats);
```

Covariance in Java

- How can we make the API general?

```
void writeToXml(List<Cat> cats);
void writeToXml(List<Tiger> cats);
void writeToXml(List<Lion> cats);
…
void writeToXml(List<Ocelot> cats);
```

Covariance in Java

- How can we make the API general?

```
<T extends Cat>
void writeToXml(List<T> cats);
```

Covariance in Java

- How can we make the API general?

```
<T extends Cat>
void writeToXml(List<T> cats);
```

- Unsatisfactory: we don’t do this:

```
<T extends Cat>
void writeToXml(T cat);
```

Covariance in Java

- How can we make the API general?

```
void writeToXml(List<? extends Cat> cats);
```

- List<? extends T> is covariant w.r.t. T

Wildcard Types (Covariant)

```
void writeToXml(List<? extends Cat> cats);
```

- The Java compiler:
  - *allows* us to obtain Cat instances from “cats”
    - This is safe even if “cats” is List<Tiger>

```
Cat firstCat = cats.get(0);
```

  - *forbids* us to give Cat instances to “cats”
    - This would be unsafe if “cats” is List<Tiger>

~~cats.add(new Lion());~~

Contravariance in Java

- How can we make the API general?

```
void donateMyTigers(List<Tiger> tigerCollector);
void donateMyTigers(List<Cat> tigerCollector);
void donateMyTigers(List<Animal> tigerCollector);
void donateMyTigers(List<HasStripes> tigerCollector);
void donateMyTigers(List<Object> tigerCollector);
```

Contravariance in Java

- How can we make the API general?

```
void donateMyTigers(List<? super Tiger> tigerCollector);
```

- List<? super T> is contravariant w.r.t. T

Wildcard Types (Contravariant)

```
void donateMyTigers(List<? super Tiger> tigerCollector);
```

- The Java compiler:
  - *allows* us to give Tiger instances to “tigerCollector”
    - This is safe even if “tigerCollector” is List<Cat>

```
tigerCollector.add(new Tiger());
```

  - *forbids* us to obtain Cat instances from “cats”
    - This would be unsafe if “cats” is List<Animal>

~~Tiger firstTiger = tigerCollector.get(0);~~

Available Methods (Covariant)

```
interface List<T>
              extends Collection<T>
{
   int size();
   T get(int index);
   void add(int index, T item);
   Iterator<T> iterator();
   List<List<T>> chunks(int chunkSize);
}
```

Available Methods (Covariant)

```
interface  List<? extends T>
                  extends  ~~Collection<T>~~
{
    int  size();
    T  get(int  index);
    void  add(int  index,  ~~T  item~~);
    ~~Iterator<T>~~  iterator();
    ~~List<List<T>>~~  chunks(int  chunkSize);
}
```

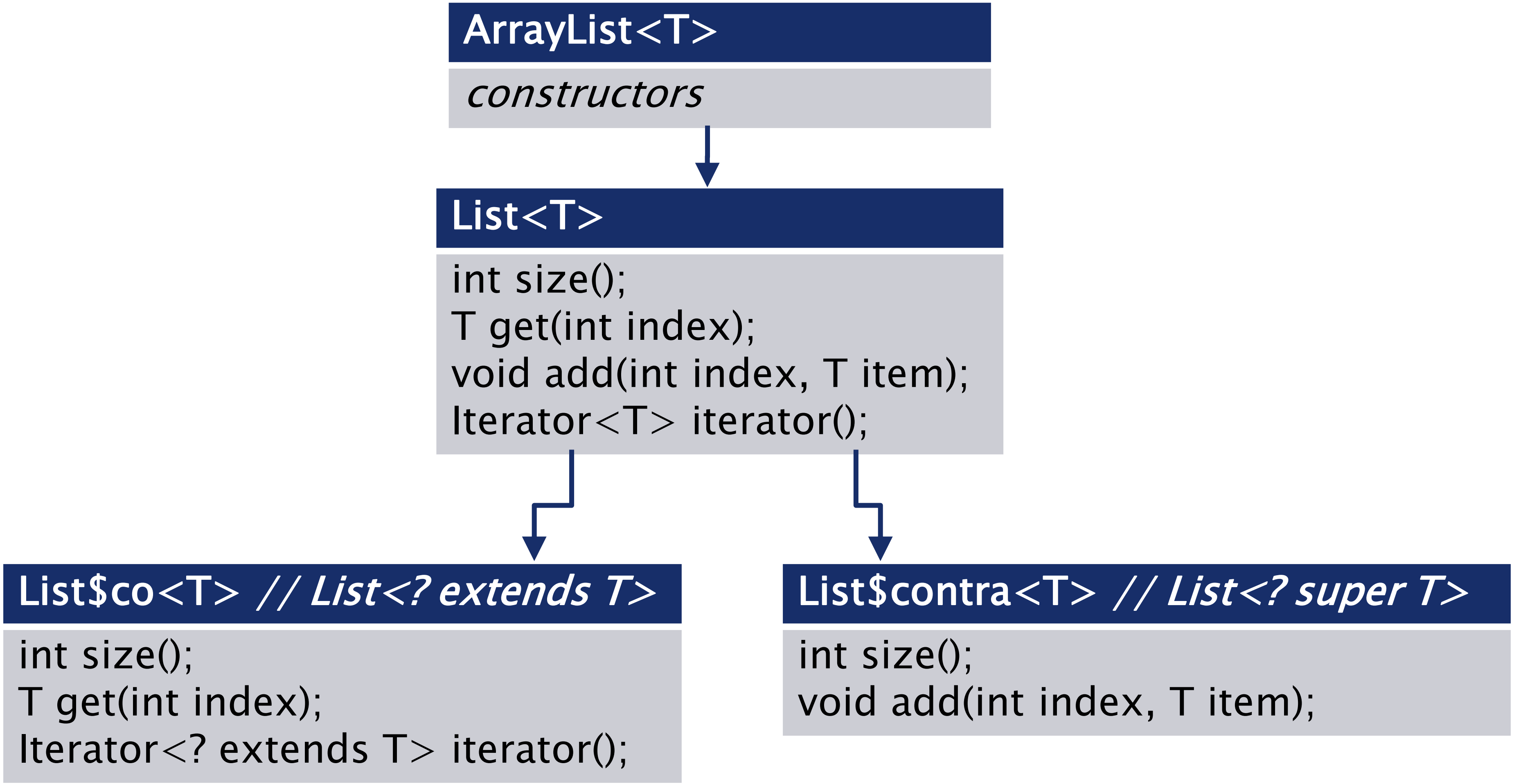
Available Methods (Covariant)

```
interface  List$co<T>  // List<? extends T>
                extends  Collection<? extends T>
{
   int  size();
   T  get(int  index);
   void  add(int  index,  null-type  item);
   Iterator<? extends T>  iterator();
   List<List<capture#1-of-? extends T>>
         chunks(int  chunkSize);
}
```

Available Methods (Contravariant)

```
interface  List$contra<T>  // List<? super T>
                  extends  Collection<? super T>
{
    int  size();
    Object  get(int  index);
    void  add(int  index,  T  item);
    Iterator<? super T>  iterator();
    List<List<capture#1-of-? super T>>
            chunks(int  chunkSize);
}
```

Implicit Interfaces



ArrayList<T>

constructors

List<T>

int size();
T get(int index);
void add(int index, T item);
Iterator<T> iterator();

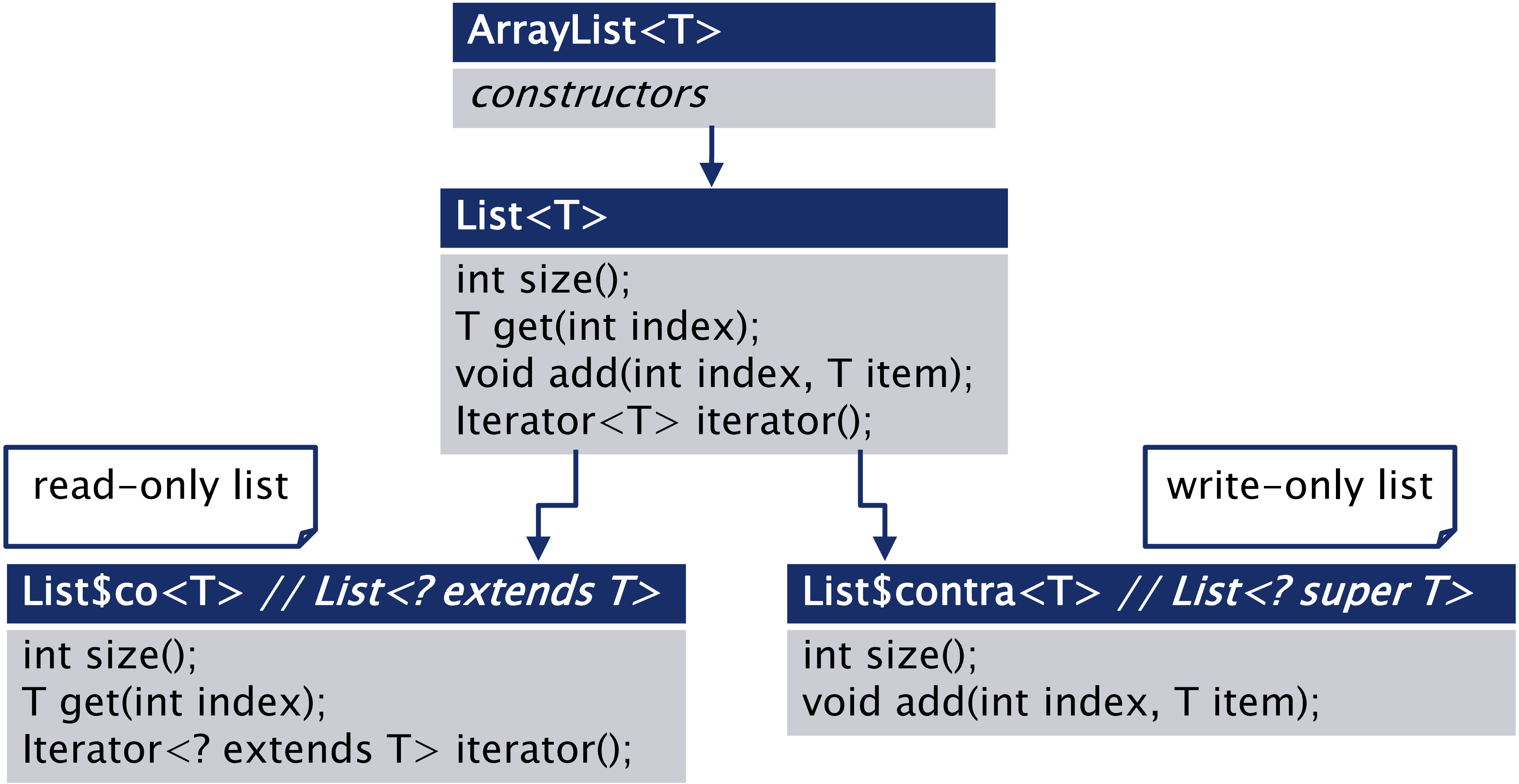
List$co<T> *// List<? extends T>*

int size();
T get(int index);
Iterator<? extends T> iterator();

List$contra<T> *// List<? super T>*

int size();
void add(int index, T item);

Implicit Interfaces



ArrayList<T>

constructors

List<T>

int size();
T get(int index);
void add(int index, T item);
Iterator<T> iterator();

read-only list

write-only list

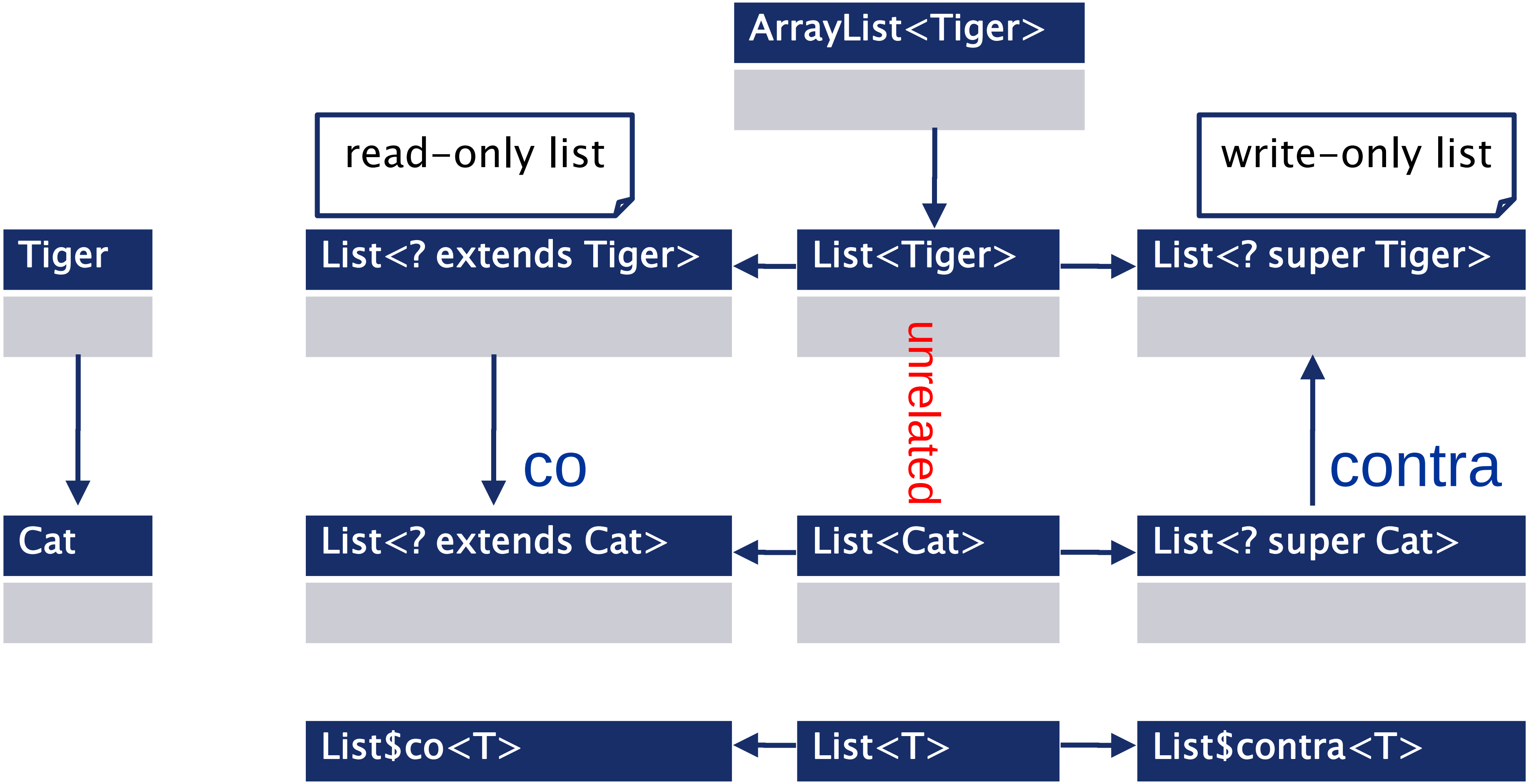
List$co<T> *// List<? extends T>*

int size();
T get(int index);
Iterator<? extends T> iterator();

List$contra<T> *// List<? super T>*

int size();
void add(int index, T item);

Covariance and Contravariance



ArrayList<Tiger>

read-only list

write-only list

Tiger

List<? extends Tiger>

List<Tiger>

List<? super Tiger>

unrelated

co

contra

Cat

List<? extends Cat>

List<Cat>

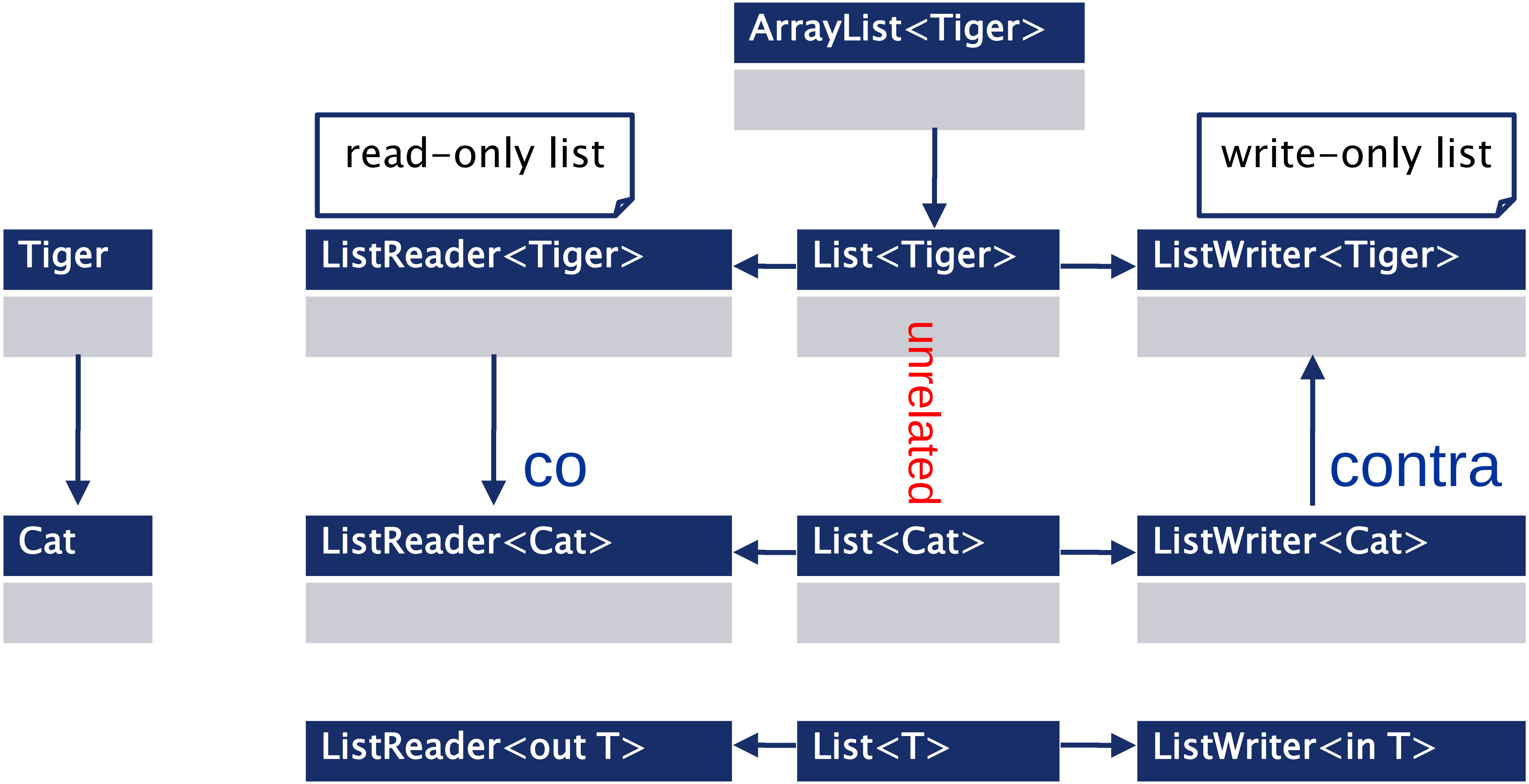
List<? super Cat>

List$co<T>

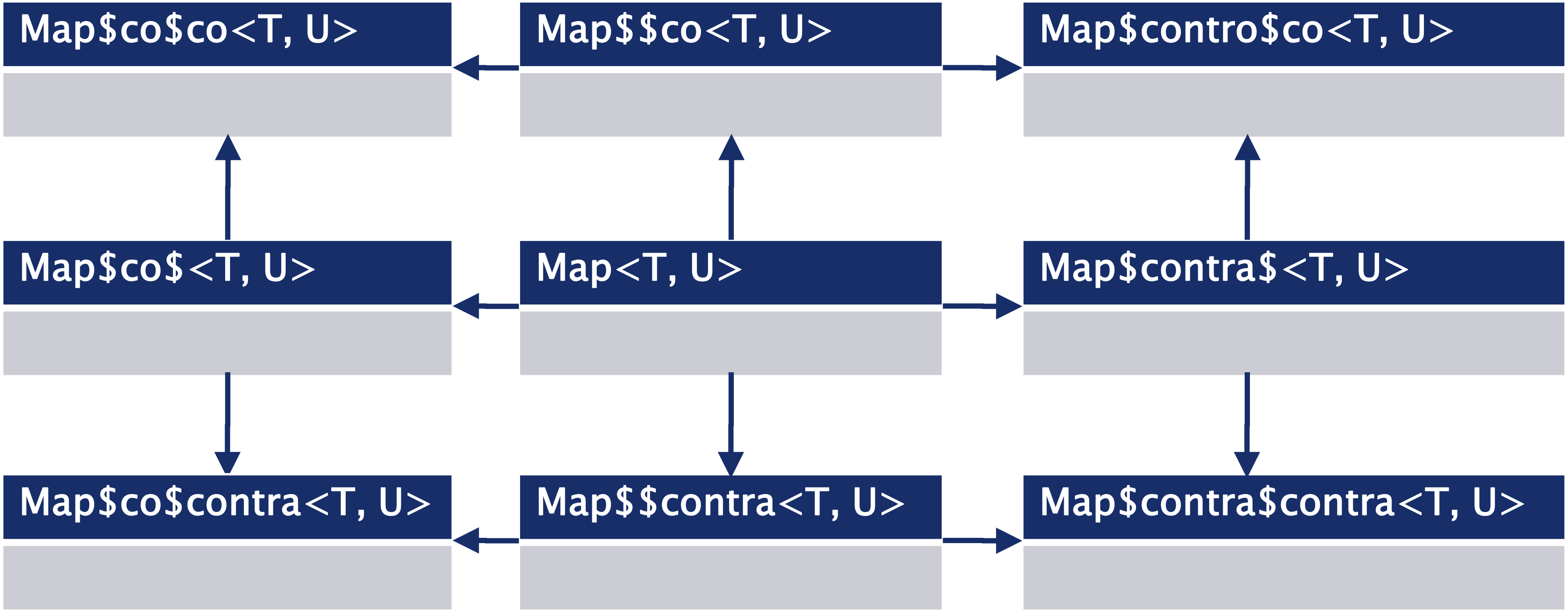
List<T>

List$contra<T>

Covariance and Contravariance in C♯



Multiple Type Parameters



Wildcard vs Variant Interface

- Wildcard perspective:

List < ? extends T >

- “List of some sub-type of T”

⇔

- Variant Interface perspective:

List<?extends T>
List$co<T>

- “Covariant list of T”

Implementation Hiding

```
interface Car
{
    ArrayList<Wheel> wheels();
}
```

Implementation Hiding

```
interface Car
{
    ArrayList<Wheel> wheels();
}
```

```
interface Car
{
    List<Wheel> wheels();
}
```

Implementation Hiding

```
interface Car
{
    ArrayList<Wheel> wheels();
}
```

```
interface Car
{
    List<Wheel> wheels();
}
```

```
interface Car
{
    List<? extends Wheel> wheels();
}
```

Documentation Aid

```
/**
 * @return           car’s wheels
 */
List<Wheel> wheels()
{
  return this.wheels;
}
```

Documentation Aid

```
/**
 * @return            car’s wheels;
 *                    please don’t modify
 */
List<Wheel> wheels()
{
  return this.wheels;
}
```

Documentation Aid

```
/**
 * @return              car’s wheels;
 *                      please don’t modify
 *                      pretty please
 */
List<Wheel> wheels()
{
  return this.wheels;
}
```

Documentation Aid

```
/**
 * @return          car’s wheels;
 *                  please don’t modify
 *                  (actually, don’t bother trying)
 */
List<Wheel> wheels()
{
  return Collections.unmodifiableList(
                this.wheels);
}
```

Documentation Aid

```
/**
 * @return            car’s wheels
 */
List<? extends Wheel> wheels()
{
  return this.wheels;
}
```

Declaration-Site vs Use-Site Variance

- Declaration-Site Variance

```
interface List<out T>
{

}
```

```
interface Collector<in T>
{
    …
}
```

- Use-Site Variance

```
List<? extends Cat>
```

```
List<? super Cat>
```

Limitations of Use-Site Variance

- Cannot inherit from a variant type

```
class ListDecorator<T>
                  implements ~~List<? extends T>~~
{
    private final List<? extends T> target;
…
    public Iterator<? extends T> iterator()
    {
        return target.iterator();
    }
}
```

Limitations of Use-Site Variance

- Cannot inherit from a variant type

```
class ListDecorator<T>
              implements List<T>
{
   private final List<? extends T> target;
…
   public Iterator<T> iterator()
   {
      return ~~target.iterator()~~;
   }
}
```

Reading Types

```
public static <T>
int  binarySearch(
        List<? extends Comparable<? super T>>  list,
        T  key)
{…}
```

```
public static <T>
int  binarySearch(
        List$co<Comparable$contra<T>>  list,
        T  key)
{…}
```

Questions

Questions

? extends