Generics in C#

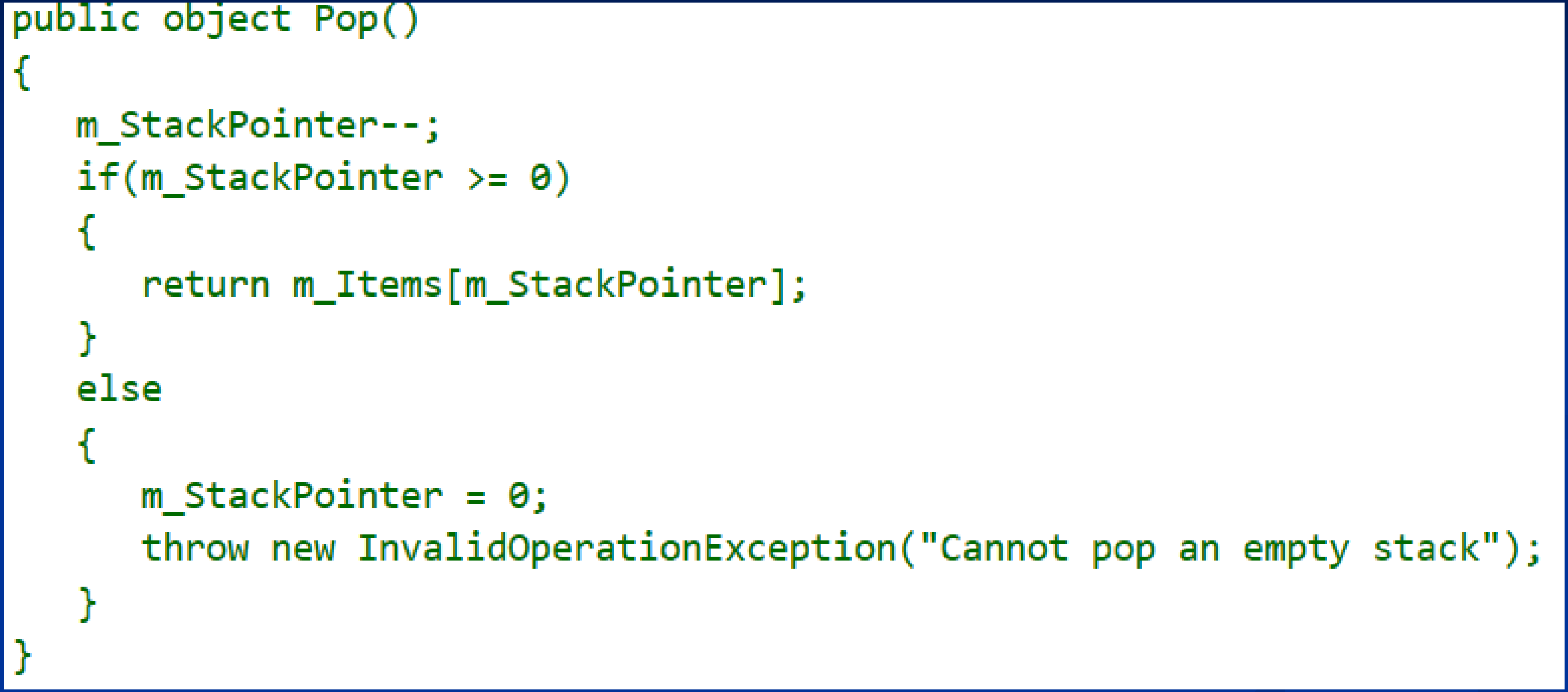
Introduction

- Generics allow you to define type-safe data structures, without committing to actual data types
  - Higher quality code (Code Re-Use)
  - Similar to generics in Java/C++ (although a bit different from the C++ implementation)

Generics w/o Help

```
public class Stack
{
   readonly int m_Size;
   int m_StackPointer = 0;
   object[] m_Items;
   public Stack():this(100)
   {}
   public Stack(int size)
   {
      m_Size = size;
      m_Items = new object[m_Size];
   }
   public void Push(object item)
   {
      if(m_StackPointer >= m_Size)
         throw new StackOverflowException();
      m_Items[m_StackPointer] = item;
      m_StackPointer++;
   }
```

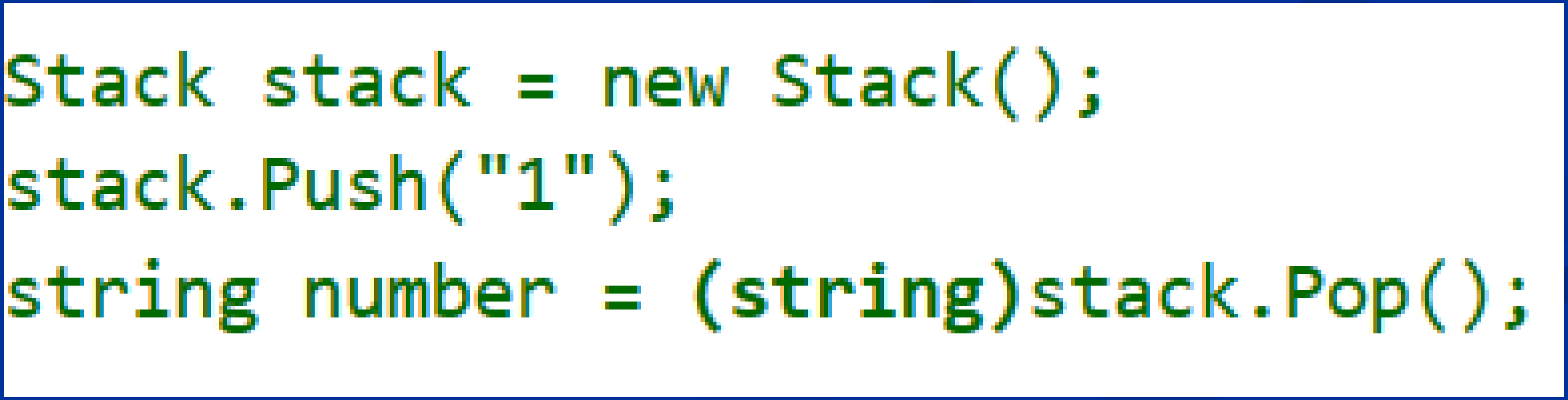
Generics w/o Help



```
public object Pop()
{
   m_StackPointer--;
   if(m_StackPointer >= 0)
   {
      return m_Items[m_StackPointer];
   }
   else
   {
      m_StackPointer = 0;
      throw new InvalidOperationException("Cannot pop an empty stack");
   }
}
```

Generics w/o Help

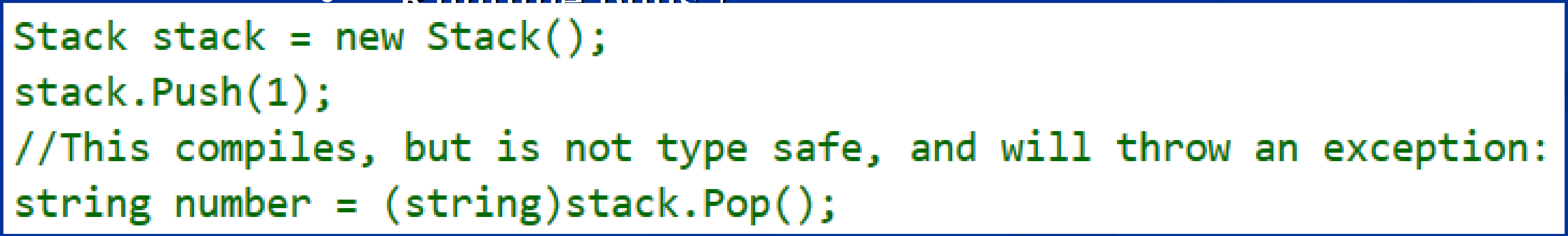
- Drawbacks
  - Performance issues
    - Boxing and unboxing for value types
    - Casting cost for references



```
Stack stack = new Stack();
stack.Push("1");
string number = (string)stack.Pop();
```

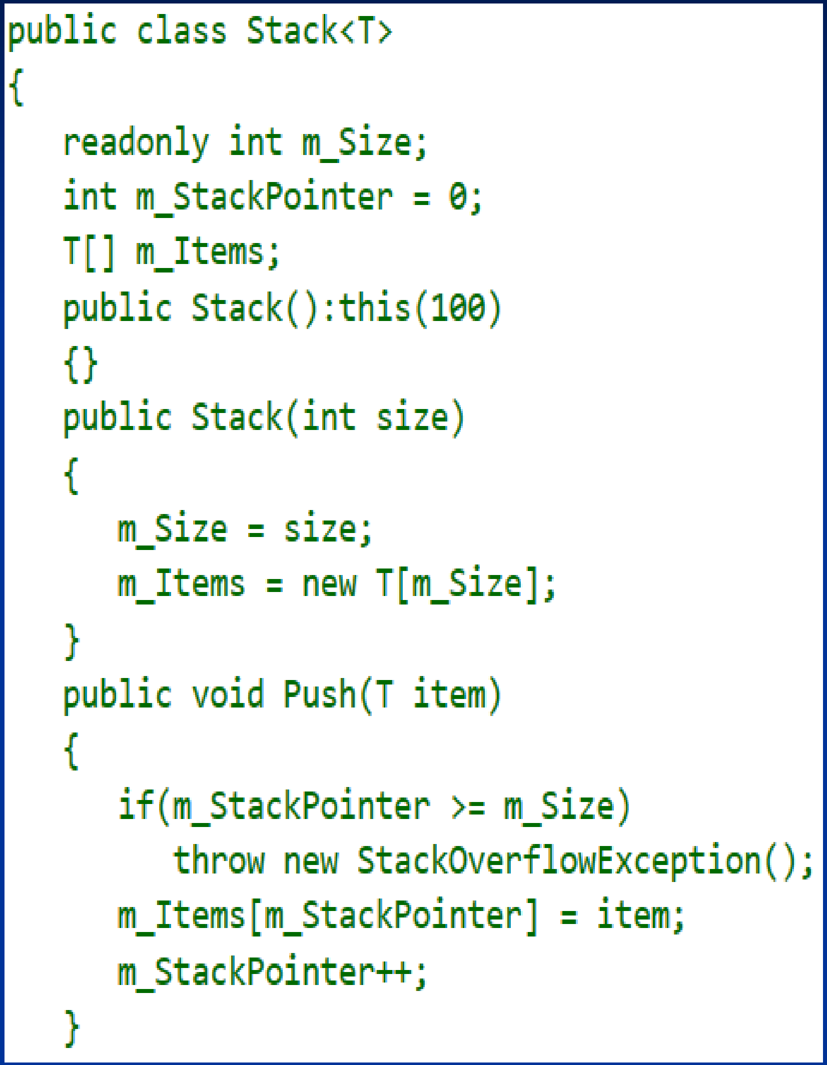
Generics w/o Help

- Drawbacks
  - Type safety ! Major issue !
    - An Object reference can reference any type of object
    - You can cast it to any other type
    - We lose compile-time safety (performance – type verifications at runtime)
    - Runtime bugs !

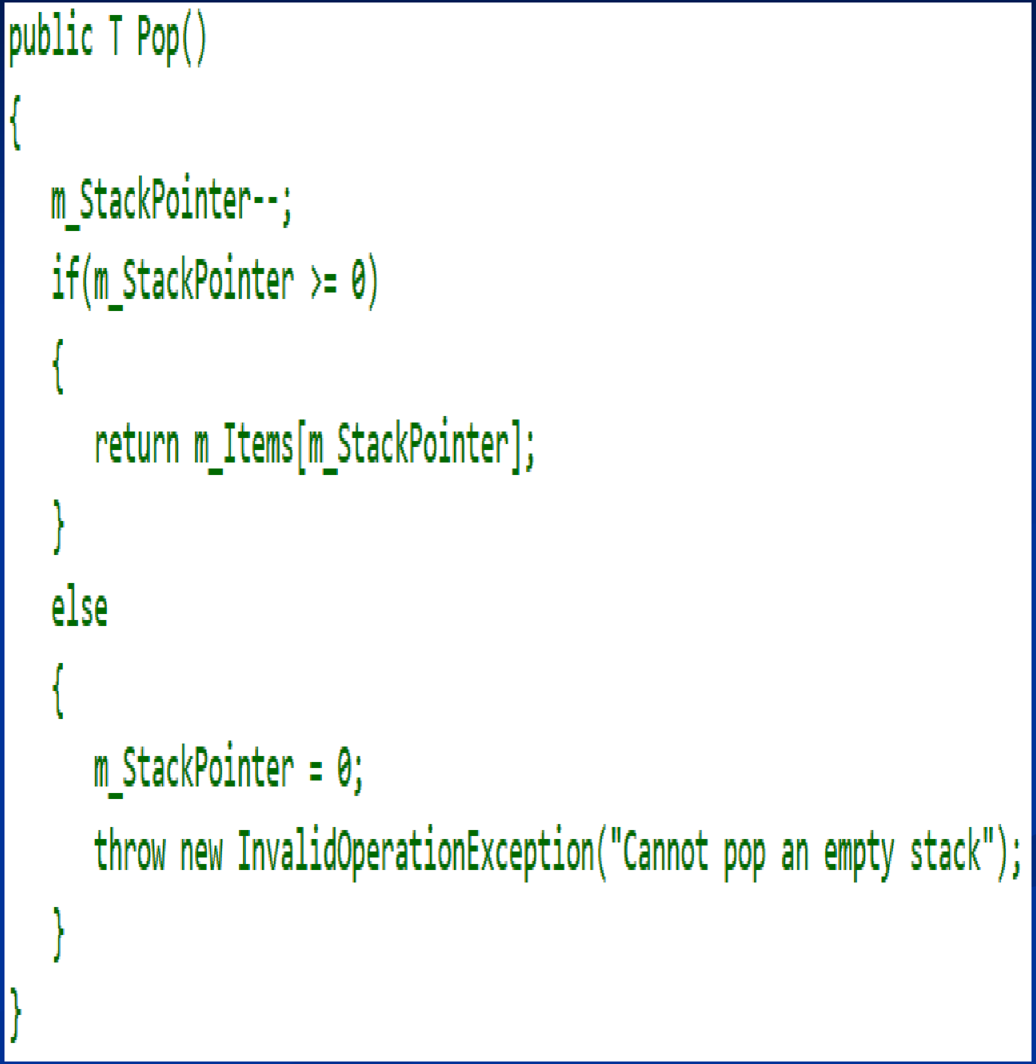


```
Stack stack = new Stack();
stack.Push(1);
//This compiles, but is not type safe, and will throw an exception:
string number = (string)stack.Pop();
```

Generics in C#

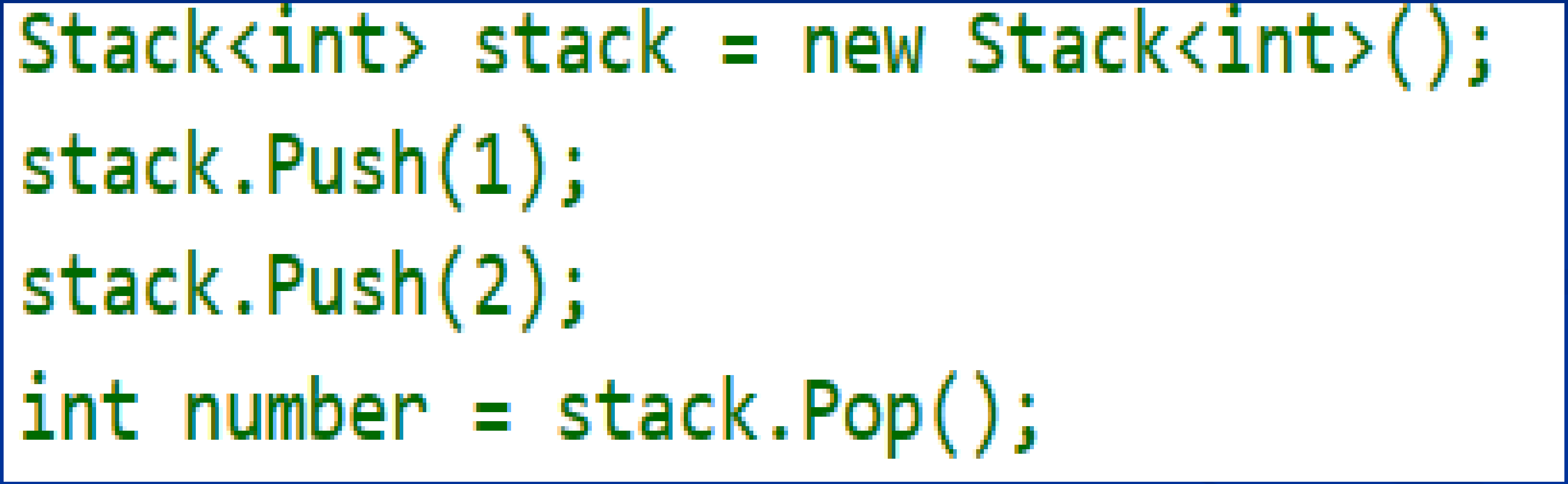


```
public class Stack<T>
{
   readonly int m_Size;
   int m_StackPointer = 0;
   T[] m_Items;
   public Stack():this(100)
   {}
   public Stack(int size)
   {
      m_Size = size;
      m_Items = new T[m_Size];
   }
   public void Push(T item)
   {
      if(m_StackPointer >= m_Size)
         throw new StackOverflowException();
      m_Items[m_StackPointer] = item;
      m_StackPointer++;
   }
```

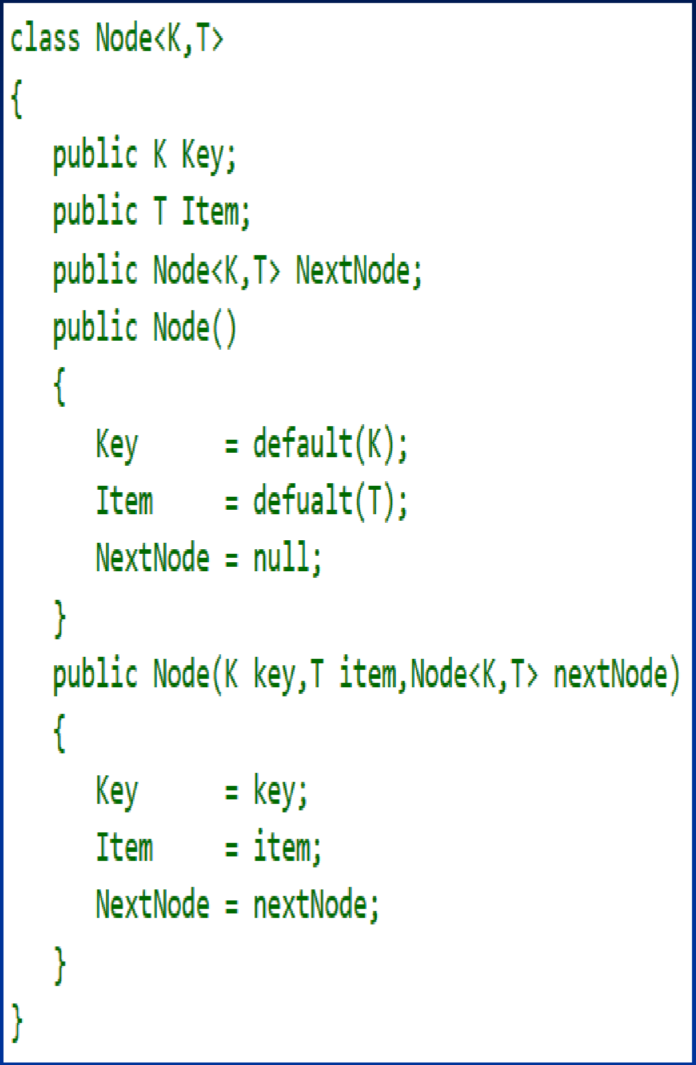


```
public T Pop()
{
   m_StackPointer--;
   if(m_StackPointer >= 0)
   {
      return m_Items[m_StackPointer];
   }
   else
   {
      m_StackPointer = 0;
      throw new InvalidOperationException("Cannot pop an empty stack");
   }
}
```

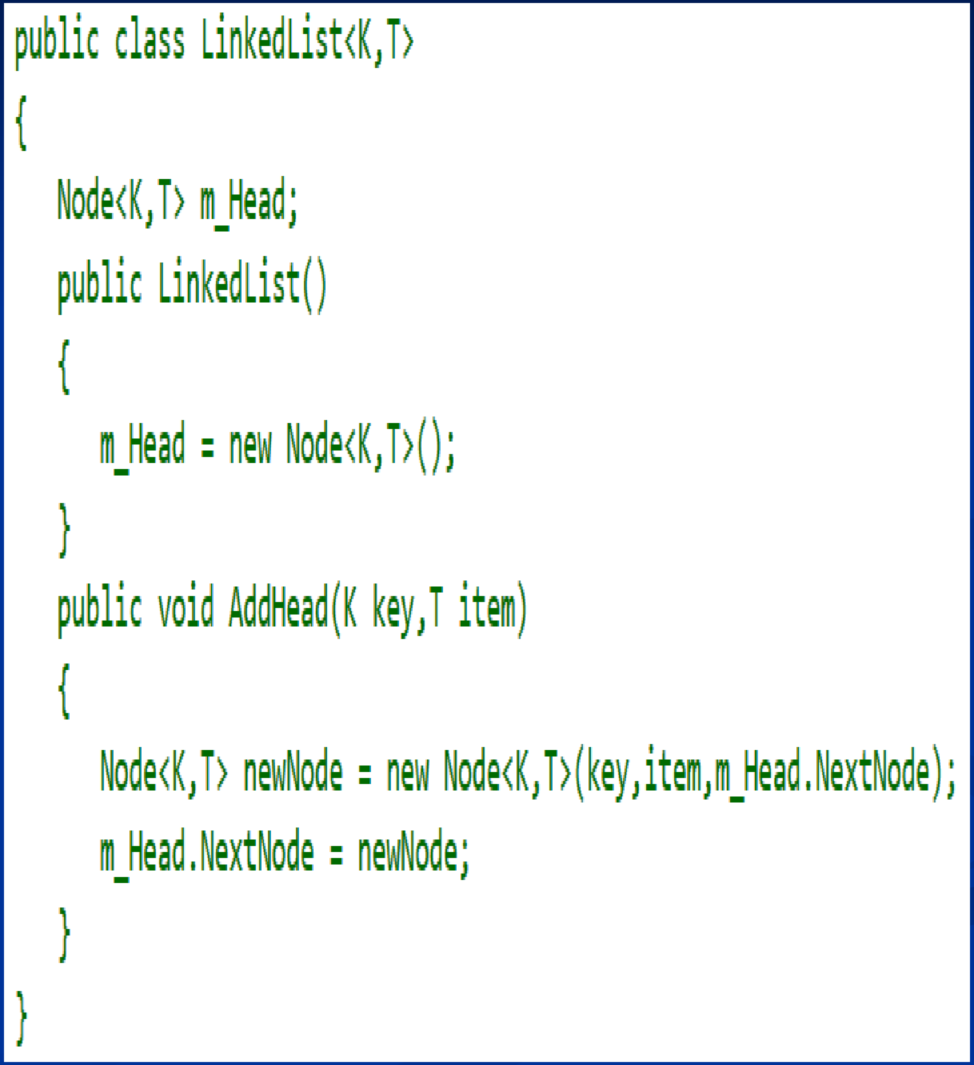
Generics in C#


```
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```

Generics w Multiple Parameters



```
class Node<K,T>
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;
   public Node()
   {
      Key      = default(K);
      Item     = defualt(T);
      NextNode = null;
   }
   public Node(K key,T item,Node<K,T> nextNode)
   {
      Key      = key;
      Item     = item;
      NextNode = nextNode;
   }
}
```



```
public class LinkedList<K,T>
{
   Node<K,T> m_Head;
   public LinkedList()
   {
      m_Head = new Node<K,T>();
   }
   public void AddHead(K key,T item)
   {
      Node<K,T> newNode = new Node<K,T>(key,item,m_Head.NextNode);
      m_Head.NextNode = newNode;
   }
}
```

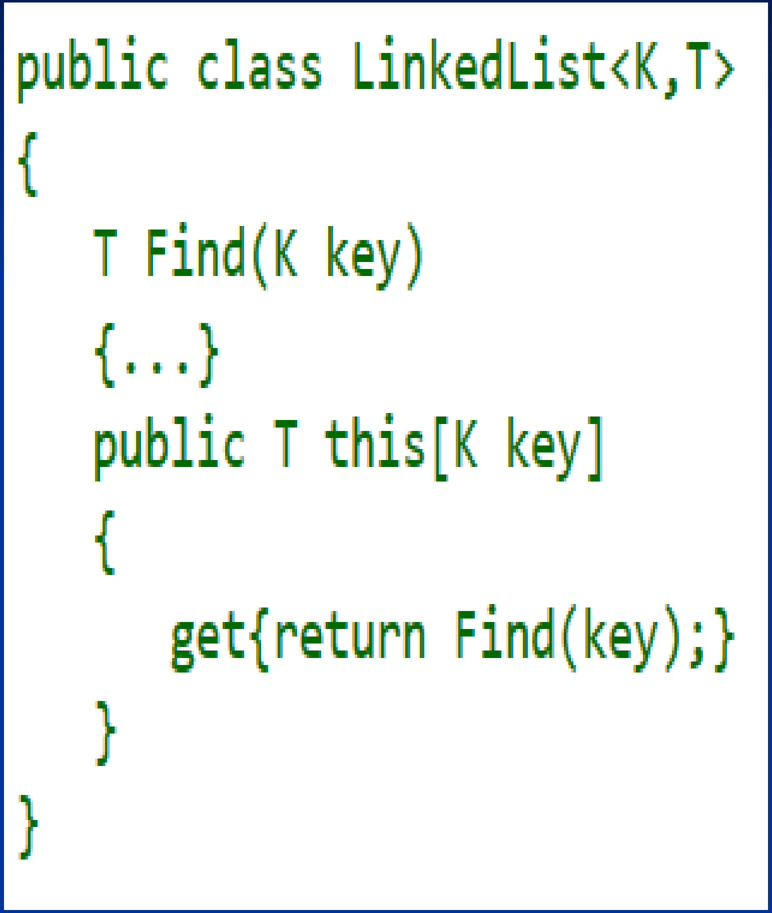
Generics w Multiple Parameters

```
LinkedList<DateTime,string> list = new LinkedList<DateTime,string>();
list.AddHead(DateTime.Now,"AAA");
```

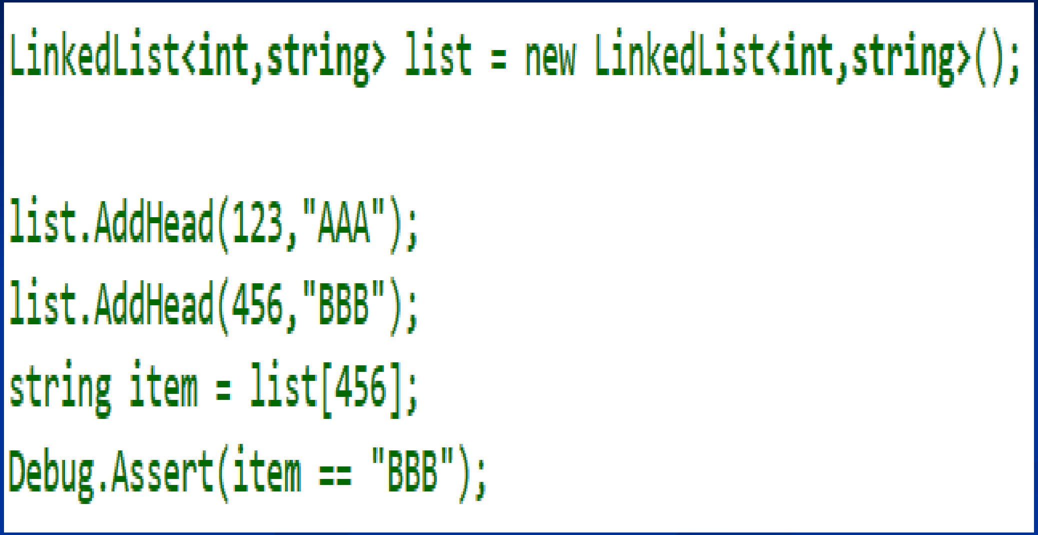
```
LinkedList<DateTime,string> list = new LinkedList<DateTime,string>();
list.AddHead(DateTime.Now,"AAA");
```

```
using List = LinkedList<int,string>;

class ListClient
{
   static void Main(string[] args)
   {
      List list = new List();
      list.AddHead(123,"AAA");
   }
}
```

Generic Constraints (1/6)

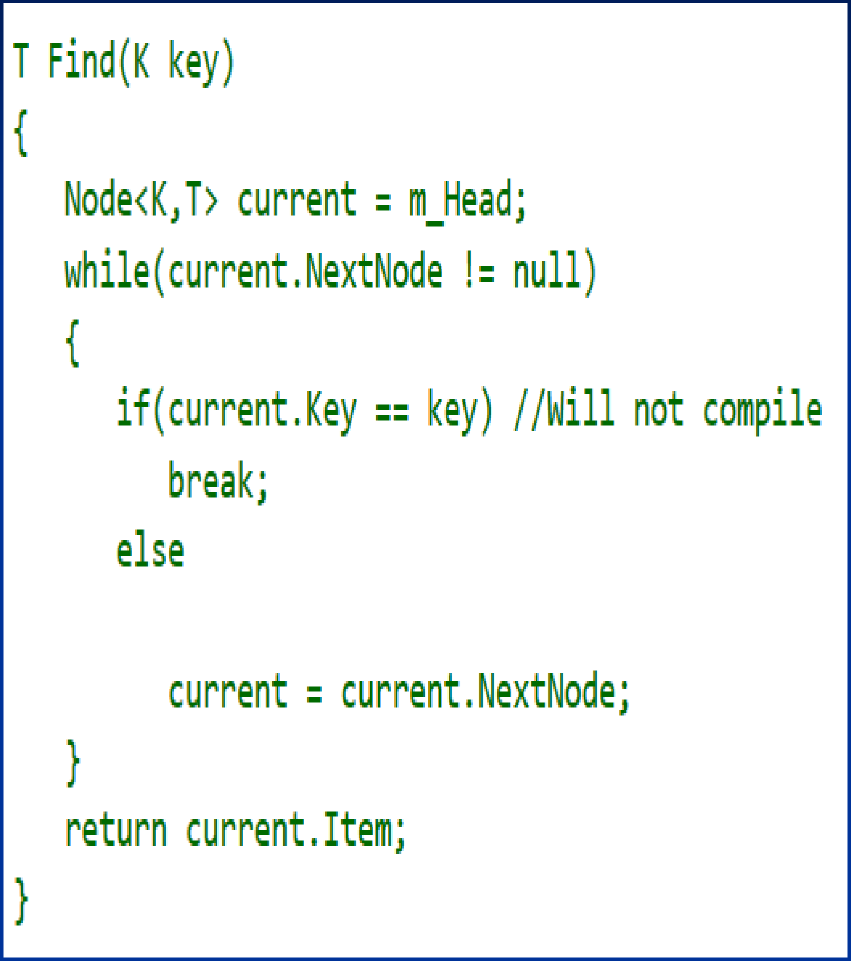


```
public class LinkedList<K,T>
{
   T Find(K key)
   {...}
   public T this[K key]
   {
      get{return Find(key);}
   }
}
```



```
LinkedList<int,string> list = new LinkedList<int,string>();


list.AddHead(123,"AAA");
list.AddHead(456,"BBB");
string item = list[456];
Debug.Assert(item == "BBB");
```

Generic Constraints (2/6)



```
T Find(K key)
{
   Node<K,T> current = m_Head;
   while(current.NextNode != null)
   {
      if(current.Key == key) //Will not compile
         break;
      else

         current = current.NextNode;
   }
   return current.Item;
}
```

- Will not compile
  - Are you sure K supports the == operator ???? !

Generic Constraints (3/6)

- Overcome using the Icomparable interface

```
if(current.Key.CompareTo(key) == 0)
```

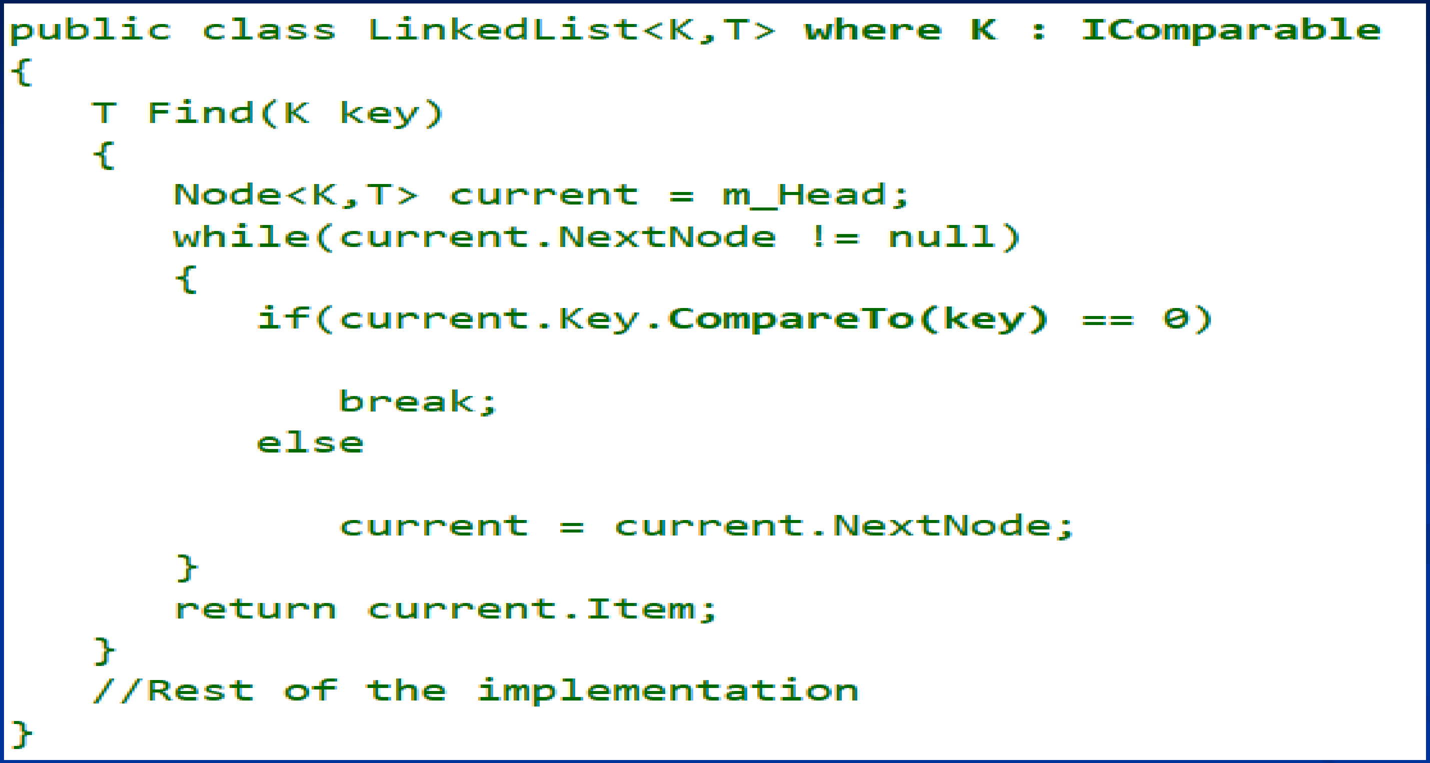
  - Will not compile ! K does not necessarily implement the Icomparable interface !

- Overcome using casting !

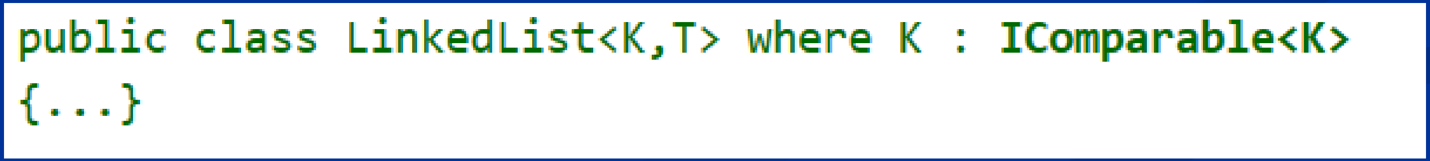
```
If (((IComparable)(current.Key)).CompareTo(key) == 0)
```

  - Runtime Error !!!!

Generic Constraints (4/6)



```
public class LinkedList<K,T> where K : IComparable
{
   T Find(K key)
   {
      Node<K,T> current = m_Head;
      while(current.NextNode != null)
      {
         if(current.Key.CompareTo(key) == 0)

            break;
         else

            current = current.NextNode;
      }
      return current.Item;
   }
   //Rest of the implementation
}
```



```
public class LinkedList<K,T> where K : IComparable<K>
{...}
```

Generic Constraints (5/6)

- Multiple interfaces

```
public class LinkedList<K,T> where K :
    Icomparable<K>,Iconvertible
```

- Constraints per parameter

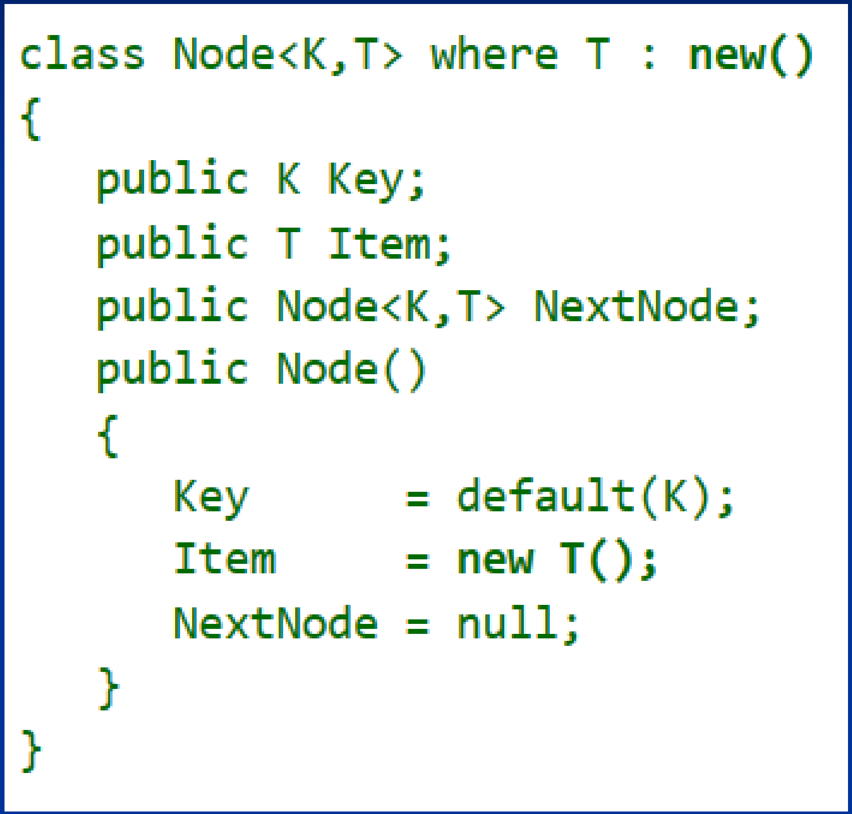
```
public class LinkedList<K,T>
            where K :   Icomparable<K>
            where T : ICloneable
```

Generic Constraints (6/6)

- Can use a base class as a constraint
  - Just ONE as C# does not support multiple inheritance of concrete classes
  - Should be first ont the list !

```
public class LinkedList<K,T> where K :
    MyBaseClass, IComparable<K>
```

Constructor Constraint

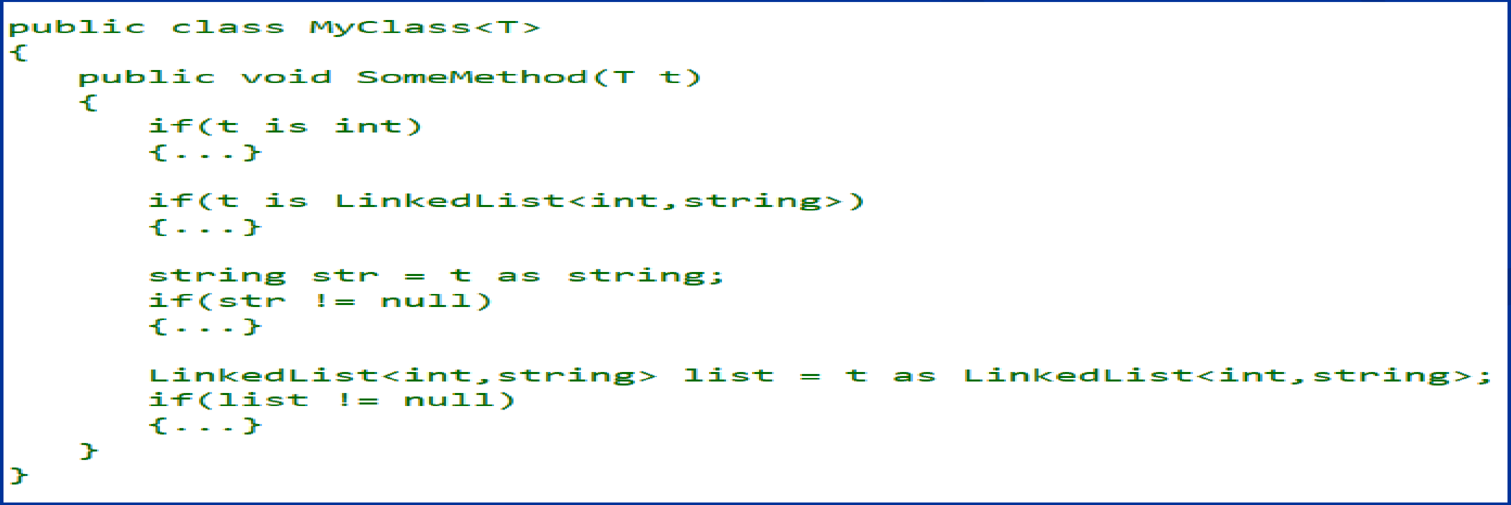


```
class Node<K,T> where T : new()
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;
   public Node()
   {
      Key      = default(K);
      Item     = new T();
      NextNode = null;
   }
}
```

- Make sure the T type has a default constructor !

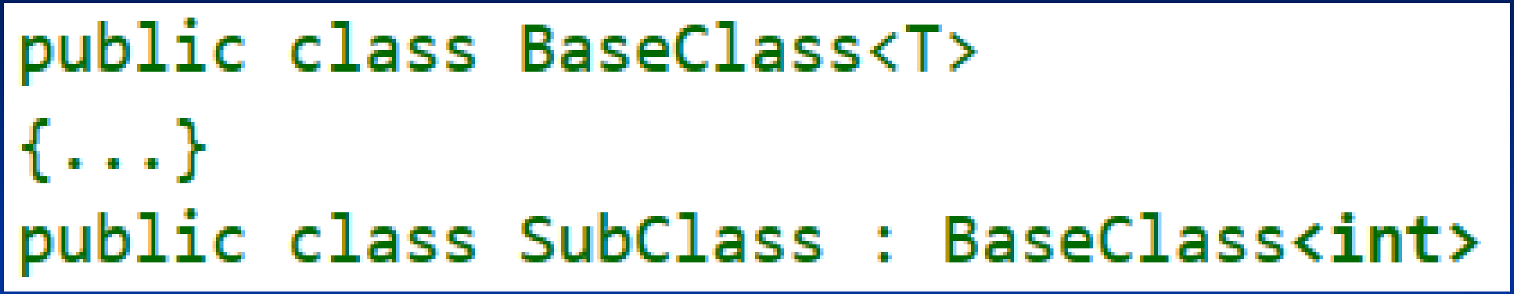
Is/As Operators Vs Casting

- Avoid runtime bugs !
- is returns true if the generic type parameter is of the queried type
- as will perform a cast if the types are compatible, and will return null otherwise

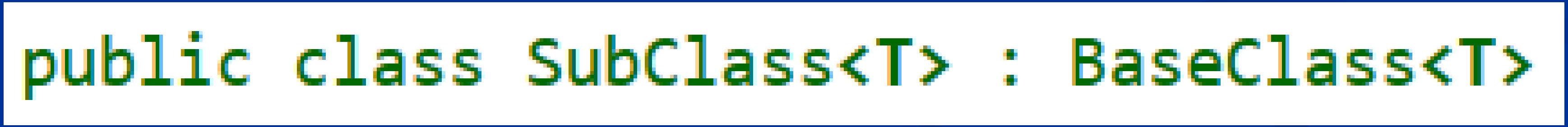


```
public class MyClass<T>
{
    public void SomeMethod(T t)
    {
        if(t is int)
        {...}

        if(t is LinkedList<int,string>)
        {...}

        string str = t as string;
        if(str != null)
        {...}

        LinkedList<int,string> list = t as LinkedList<int,string>;
        if(list != null)
        {...}
    }
}
```

Generics And Inheritance (1/3)

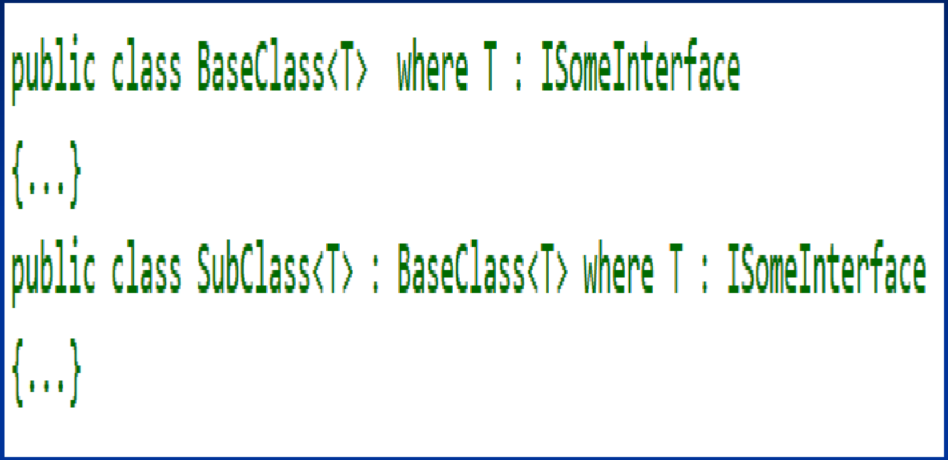


```
public class BaseClass<T>
{...}
public class SubClass : BaseClass<int>
```



```
public class SubClass<T> : BaseClass<T>
```

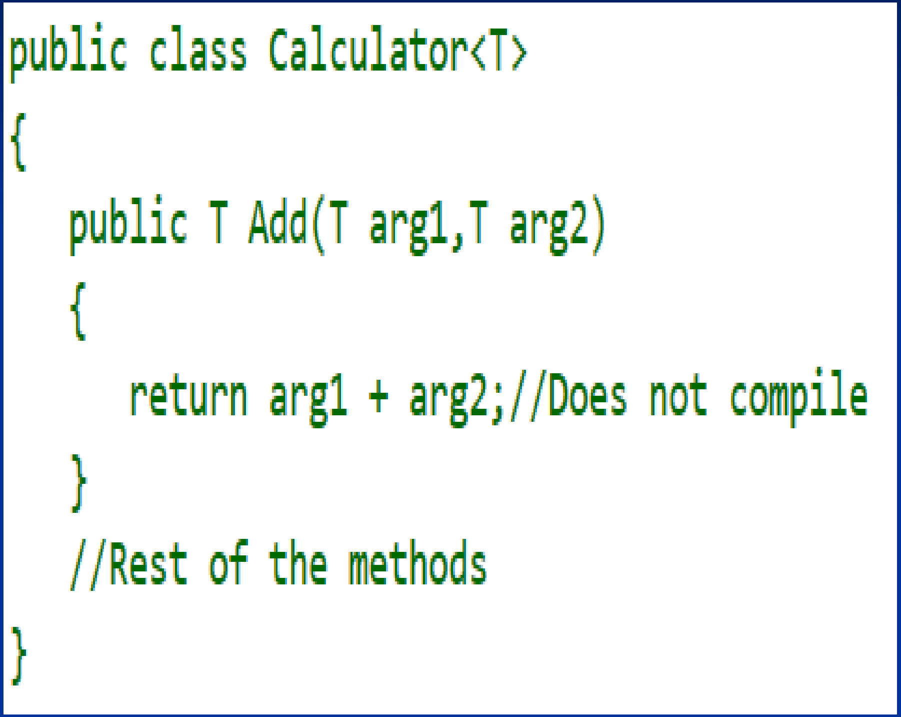
Generics And Inheritance (2/3)



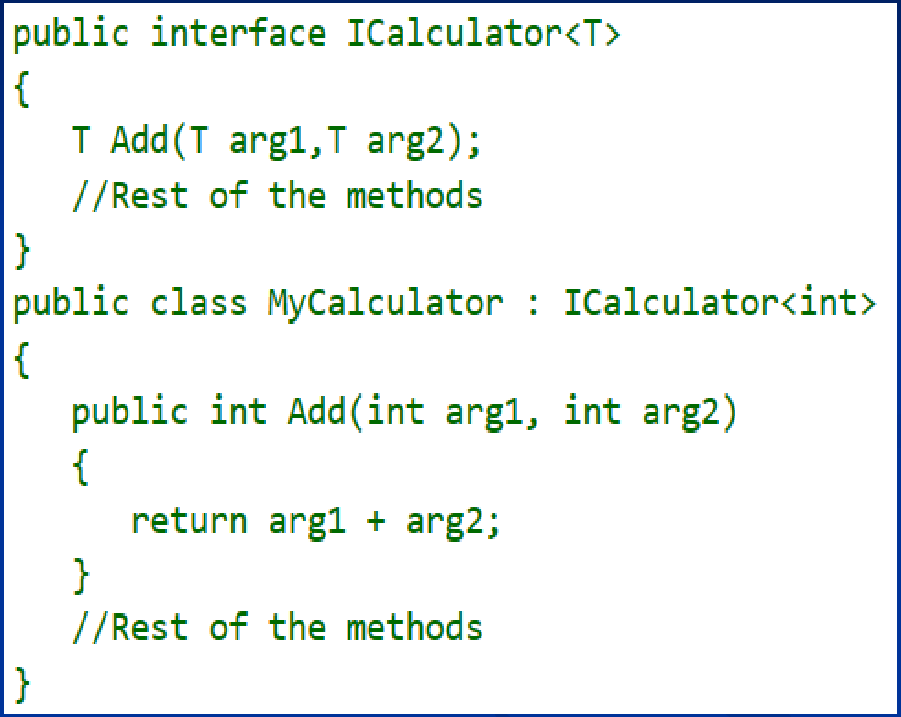
```
public class BaseClass<T>  where T : ISomeInterface
{...}
public class SubClass<T> : BaseClass<T> where T : ISomeInterface
{...}
```

- Repeat constraints in subclasses
  - Types, constructor constraints

Generics And Inheritance (3/3)



```
public class Calculator<T>
{
   public T Add(T arg1,T arg2)
   {
      return arg1 + arg2;//Does not compile
   }
   //Rest of the methods
}
```



```
public interface ICalculator<T>
{
   T Add(T arg1,T arg2);
   //Rest of the methods
}
public class MyCalculator : ICalculator<int>
{
   public int Add(int arg1, int arg2)
   {
      return arg1 + arg2;
   }
   //Rest of the methods
}
```

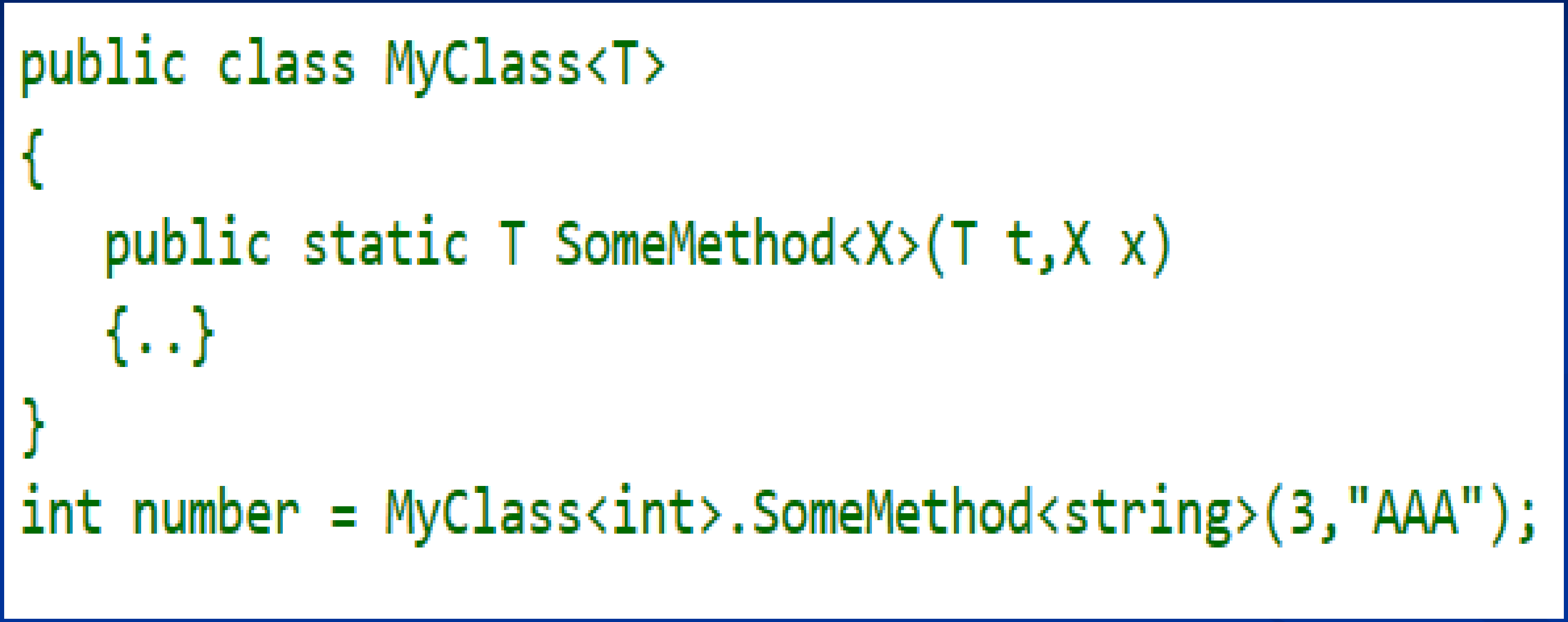
Generic Methods

- allows you to call the method with a different type every time
- Handy for utility classes

```
public class MyClass
{
   public void MyMethod<T>(T t)
   {...}
}
```

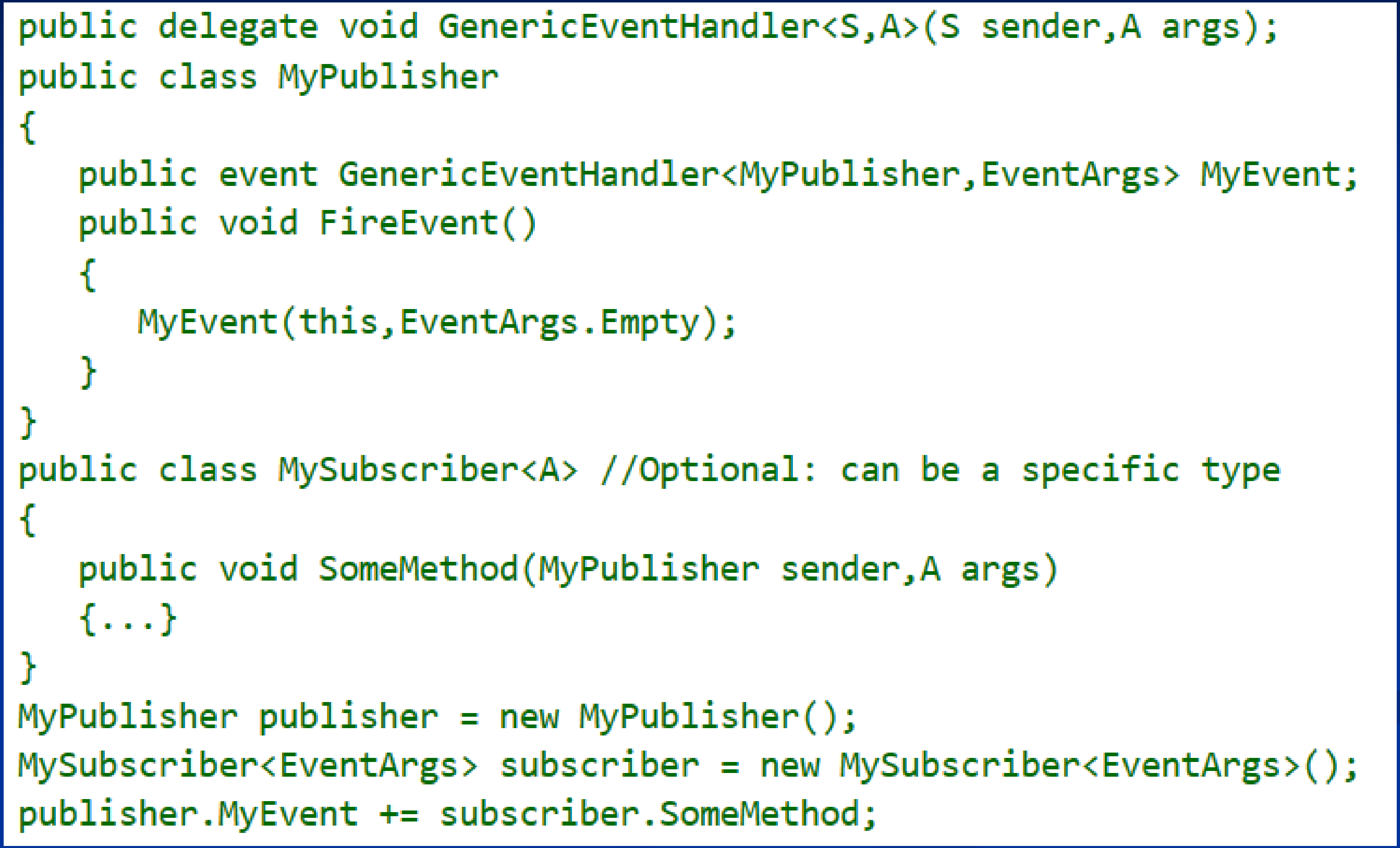
```
MyClass obj = new MyClass();
obj.MyMethod<int>(3);
```

Generic Static Methods



```
public class MyClass<T>
{
   public static T SomeMethod<X>(T t,X x)
   {..}
}
int number = MyClass<int>.SomeMethod<string>(3,"AAA");
```

Generic Event Handling



```
public delegate void GenericEventHandler<S,A>(S sender,A args);
public class MyPublisher
{
   public event GenericEventHandler<MyPublisher,EventArgs> MyEvent;
   public void FireEvent()
   {
      MyEvent(this,EventArgs.Empty);
   }
}
public class MySubscriber<A> //Optional: can be a specific type
{
   public void SomeMethod(MyPublisher sender,A args)
   {...}
}
MyPublisher publisher = new MyPublisher();
MySubscriber<EventArgs> subscriber = new MySubscriber<EventArgs>();
publisher.MyEvent += subscriber.SomeMethod;
```