Abstract
Generics are used to code algorithms in a type-ignorant fashion without losing type information. E.g., a stack has the same methods and uses the same strategy no matter what type the elements have; if the stack is coded for Object elements anything can be pushed but only Object values are popped. Generics allow a compiler to do much better. Both, Java 5 and C# 2.0, support parametrized types (aka generics); however, that is where the similarity ends. We will introduce generics, explain the implementation differences, and discuss the consequences for programmers.
• Concepts
• Objectives
• Usage
• Pitfalls: Inheritance
• Pitfalls: Arrays
• Pitfalls: static
• Pitfalls: Factories
• Example: Lisp
• Example: Tree-based Container
• Summary and References
Definition

- Generic programming has come to mean writing code where some of the types are parameterized.

- Conceptually,
  - Objects are instantiated from classes
    - by specifying constructor arguments.
  - Real code can be instantiated from generically written code
    - by specifying type parameter "values".
**Programming Practicalities**

- A stack has the same methods and uses the same strategy no matter what type the elements have.

- If the stack is coded for `Object` elements anything can be pushed but only `Object` values are popped. *(contravariance)*

- Generics are used to code algorithms
  - in a type-ignorant fashion
  - without losing type information.
public class Stack {
    public void push( Object x ) {
        head = new Node( x, head );
    }
    public Object pop() {
        Object result = head.value;
        head = head.next;
        return result;
    }
    private Node head = null;
}

Stack ss = new Stack();
ss.push( "Elias" );
String jeh = (String)ss.pop();
Java

```java
public class Stack<E> {
    public void push(E x) {
        head = new Node<E>(x, head);
    }
    public E pop() {
        E result = head.value;
        head = head.next;
        return result;
    }
    private Node<E> head = null;
}

Stack<String> ss = new Stack<String>();
ss.push( "Elias" );
String jeh = ss.pop();
```
Newspeak

```java
public class Stack<E> {
    public void push(E x) {
        head = new Node<E>(x, head);
    }

    public E pop() {
        E result = head.value;
        head = head.next;
        return result;
    }

    private Node<E> head = null;
}

Stack<String> ss = new Stack<String>();
ss.push("Elias");
String jeh = ss.pop();
```

**E** – define generic parameter

**E** – use of generic parameter

**String** – generic argument

Java
• Concepts

• **Objectives**

• Usage

• Pitfalls: Inheritance

• Pitfalls: Arrays

• Pitfalls: static

• Pitfalls: Factories

• Example: Lisp

• Example: Tree-based Container

• Summary and References
Templated source code may or may not be compiled into an intermediate form, but its genericity is maintained.

As the compiler encounters requests for specific instances of the code, the template code is fetched and compiled down to the target language.

Type safety is checked at this time.

The link phase brings together all compiled instances of the templated source.
Java 5 Approach

- Don’t change JVM.
  - Implement completely at compile time.
  - erasure
- Interoperate between legacy and new code.
- Consequences:
  - Reference (object) type parameters only.
  - No runtime overhead.
  - Invisible to reflection.

“No runtime overhead?” I’m now reading that a cast is still done in the generated byte code.
Java 5 Generics

- Source code with generics is compiled once, as if each generic parameter were `Object`.
  - (or some other type if generic parameter is constrained)
- Actual type arguments are remembered only during compilation for type checking.
- Run-time code has no memory of these different types ⇒ *erased* before byte code is generated.
C# 2.0 Approach

- Expand concrete classes through JIT compiler.
- New collection classes.
- Consequences:
  - Any parameter type.
  - Space requirements increase.
  - Competitive speed for primitive types.
  - Visible to reflection.
C# 2.0 Generics

- Source code with generics is checked and compiled into a special intermediate form that retains its genericity.
- As the program runs and classes are used, the template code is pulled in using the actual type arguments and compiled “just in time”.
  - This only happens the first time the code is needed with those argument types.
• The code executed at run-time is type-safe.

• The source code only needs to be type-checked once. (Java, C#)

There will not be a run-time type check unless the Java compiler issues unchecked warnings.
- Concepts
- Objectives

**Usage**
- Pitfalls: Inheritance
- Pitfalls: Arrays
- Pitfalls: static
- Pitfalls: Factories
- Example: Lisp
- Example: Tree-based Container

- Summary and References
• classes, interfaces, and methods can have type parameters.
• type parameters can be constrained by inheritance.
Patterns

```csharp
class|struct|interface X<T, ...> : Y, I, ...
    where T : class|struct, I, new(), ... where ... { ... }

modifier type name<T, ...>( ...) 
    where T : class|struct, I, new(), ... where ... { ... }

modifier type delegate name<T, ...>( ...) 
    where T : class|struct, I, new(), ... where ...;
```

- reference and value types, interfaces, methods, and delegates can have type parameters.
- type parameters can be constrained to value or reference type, by inheritance, and to have a trivial constructor.
public class Stack<E> {
    public void push(E x) {
        head = new Node<E>(x, head);
    }
    public E pop() {
        E result = head.value;
        head = head.next;
        return result;
    }
    private Node<E> head = null;
}

Stack<String> ss = new Stack<String>();
ss.push("Elias");
String jeh = ss.pop();
public class Stack<E> {
    public void push(E x) {
        head = new Node<E>(x, head);
    }
    public E pop() {
        E result = head.value;
        head = head.next;
        return result;
    }
    private Node<E> head = null;
}

Stack<String> ss = new Stack<String>();
ss.push( "Elias" );
String jeh = ss.pop();
Inner Classes

```java
public class Stack<E> {

    public void push( E x ) {
        head = new Node<E>( x, head );
    }

    public E pop() {
        E result = head.value;
        head = head.next;
        return result;
    }

    private Node<E> head = null;
}

protected static class Node<T> {

    public final T value;
    public final Node<T> next;

    public Node( T value, Node<T> next ) {
        this.value = value; this.next = next;
    }
}
```

Java
Inner Classes

```csharp
public class Stack<E> {
    public void Push(E x) {
        head = new Node(x, head);
    }
    public E Pop() {
        E result = head.value;
        head = head.next;
        return result;
    }
    private Node head = null;
}
```

```csharp
protected class Node {
    public readonly E value;
    public readonly Node next;
    public Node(E value, Node next) {
        this.value = value; this.next = next;
    }
}
```

- type parameter is in scope.
Instantiating Parameter Types

```java
public class List<E> {
    protected int n;
    protected E[] list =
        (E[]) new Object[]{null, null};

    public void push(E x) {
        list[n++] = x;
    }

    public E pop() {
        return list[--n];
    }
}
```

$ javac -Xlint:unchecked List.java
List.java:3: warning: [unchecked] unchecked cast

• cannot create objects or arrays of parameter type.

Java
Instantiating Parameter Types

```csharp
public class List<E,F>
    where E: new()
    where F: class {
  E[] list = new E[]{ new E(), default(E) };
  F[] fs = new F[]{ null, null };
  // ...
}
```

- can create arrays, filled with default value (null or zero).
- can initialize with `default(type)` value.
- can create elements using `new()` constraint.
Comparing Objects

```java
static <A,B> boolean equals (A a, B b) {
    return a == null ? b == null : a.equals(b);
}
static <A,B> boolean sameAs (A a, B b) {
    return a == b;
}
```

- parameters are reference types, no problem.
Comparing Objects

- `==` comparison can be (statically) overloaded.
  - casting to `object` safely checks for identity.

- `null` is meaningless for value types.
Casting Objects

```java
static <C,D extends C> C upcast (D d) {
    return (C)d;
}
static <C,D extends C> D downcast (C c) {
    return (D)c;
}
Integer i = downcast((Number)new Integer(10));
Double d = downcast((Number)new Integer(10));
```

- both(!) examples provoke unchecked warnings.
  - (upcast not really useful.)

- second result of downcast fails at run-time.
  - first use of downcast is to Integer.
Casting Objects

```csharp
static C upcast<C,D> (D d) where D: C {
    return (C)d;
}
static D downcast<C,D> (C c) where D: C {
    return (D)c;
}
class X { }
class Y: X { }
downcast<X, Y>((new Y()));
downcast<X, Y>((new X()));
```

- `upcast` is not useful.
- in the second call the body of `downcast` fails.
Checking Types

class Oops<T> {
  static <E> boolean isA (Object o) {
    return o.getClass() == E.class
    || o instanceof E;
  }
  static <E> boolean contains (Oops<?> o) {
    return o instanceof Oops<E>;
  }
}

• `type<?>` denotes a raw type, the base class and runtime representation of all variants of `type`.

• none of the type check constructions is allowed.
Checking Types

```csharp
class Oops<T> {
    protected static bool isA<E> (object o) {
        return o.GetType() == typeof(E)
            || o is E;
    }

    protected static bool contains<E> (object a) {
        return a is Oops<E>;
    }

    protected static Oops<T> downcast<T> (object a) {
        return a as Oops<T>;
    }
}
```

- all of these work.
- `typeof` produces a class object, `is` is `instanceof`, and `as` produces `null` or the result of casting.
all but the last of these produce True.

- a generic class cannot contain `Main()`.
- an `static` class cannot extend an instance of a generic class.
• Concepts
• Objectives
• Usage
• **Pitfalls: Inheritance**
• Pitfalls: Arrays
• Pitfalls: static
• Pitfalls: Factories
• Example: Lisp
• Example: Tree-based Container
• Summary and References
In generic type specs, “extends” is used for both extension of classes and implementation of interfaces. Use “&” if extension and implementation is needed.
Inheritance and Generic Parameters

```java
void drawSet1( Set<Figure> shapes ) {}
  // accepts only sets containing instances of Figure.

<Figure> void drawSet2( Set<Figure> shapes ) {}
  // accepts sets containing any type, which we'll call "Figure".

void drawSet3( Set<? extends Figure> shapes ) {}
  // accepts sets containing any type that conforms to Figure.

<T extends Figure> void drawSet4( Set<T> shapes ) {}
  // accepts sets containing any type that conforms to Figure;
  // we'll call that type "T".
```

#1. Some of the set's elements may be descendants of Figure, but the compiler won't know it. The method will not accept a Set of anything else, not even a descendant of Figure.
#2. Here Figure is an unbound generic parameter, not a named type.
#3. Here all of the set's elements are expected to be some descendant of Figure.
#4. Same as #3, but the descendant of Figure has a name within the method.
Inheritance and Generic Parameters

```csharp
void drawSet1( Set<Figure> shapes ) {}
  // accepts only sets containing instances of Figure.

void drawSet2<Figure>( Set<Figure> shapes ) {}
  // accepts sets containing any type, which we'll call "Figure".

void drawSet4<T>( Set<T> shapes ) where T: Figure {}
  // accepts sets containing any type that conforms to Figure;
  // we'll call that type "T".
```

- there is no wildcard and no raw type.
- arguments must match exactly.
• Concepts
• Objectives
• Usage
• Pitfalls: Inheritance

**Pitfalls: Arrays**

• Pitfalls: static
• Pitfalls: Factories
• Example: Lisp
• Example: Tree-based Container
• Summary and References
Instantiating Parameter Types

public class List<E> {
    protected int n;
    protected E[] list =
        (E[])new Object[]{null, null};

    public void push( E x ) {
        list[n++] = x;
    }

    public E pop() {
        return list[--n];
    }
}

$ javac -Xlint:unchecked List.java
List.java:3: warning: [unchecked] unchecked cast
  cannot create objects or arrays of parameter type.
Instantiating Parameter Types

**Java**

```java
class X<E> {
    E[] x = (E[])new Object[10];
    X<E>[] y = (X<E>[])new X[10];
}
```

- can create objects or arrays of raw and concrete types.
- cannot cast `Object[ ]` to `X[ ]`. 
Terminology

- **Covariance:**
  - A more specific type (subclass) is allowed.
  - Return types and exceptions can be more specific when overriding methods in Java.

- **Contravariance:**
  - A less specific type (superclass) is allowed.
  - In C# a method may have less specific parameters than a delegate declaration requires.

http://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)
Covariance and Arrays

- compiles just fine but fails at run-time — array element assignment is always checked to be covariant.

```
Object[] objectArray;
String[] stringArray;
stringArray = new String[2];
stringArray[0] = "hello";
objectArray = stringArray;
objectArray[1] = 42;  // (*)
for ( int i=0; i<2; ++i ) {
    System.out.println( stringArray[i].length() );
}
```

ArrayStoreException
Covariance and Arrays

- Compiles just fine but fails at run-time — array element assignment is always checked to be covariant.

```csharp
object[] objectArray;
string[] stringArray;
stringArray = new string[2];
stringArray[0] = "hello";
objectArray = stringArray;
objectArray[1] = 42; // (*)
for (int i=0; i<2; ++i) {
    Console.WriteLine(stringArray[i].Length);
}
```

<ArrayTypeMismatchException>
Covariance and Generic Classes

Vector<Object> objectArray;
Vector<String> stringArray;
stringArray = new Vector<String>(2);
stringArray.set( 0, "hello" );
objectArray = stringArray; // (*)
objectArray.set( 1, 42 );
for ( int i=0; i<2; ++i ) {
    System.out.println( stringArray.get( i ).length() );
}

- does not compile — Vector<String> is not a subclass of Vector<Object>.

Incompatible types.
Found: Vector<String>
Required: Vector<Object>
Covariance and Generic Classes

```csharp
List<object> objectArray;
List<string> stringArray;
stringArray = new List<string>(2);
stringArray[0] = "hello";
objectArray = stringArray; // (*)
objectArray[1] = 42;
for (int i = 0; i < 2; ++i) {
    Console.WriteLine(stringArray[i].Length);
}
```

- does not compile — List<string> is not a subclass of List<object>.

*Cannot implicitly convert...*
Contravariance and Generic Classes

- `super` can be used with a wildcard to express contravariance.
- Copying to a contravariant collection is allowed.

```java
static <S> void append (Collection<S> src, Collection<? super S> dest) {
  for (S element: src) {
    dest.add(element);
  }
}
```
Contravariance and Generic Classes

- contravariance must be expressed with a where clause.
- copying to a contravariant collection is allowed.

```csharp
static void append<S,T> (ICollection<S> src, ICollection<T> dest)
    where S: T {
  foreach (S element in src)
    dest.Add(element);
}
```
Checked Collections

class Legacy {
  static void crash (List list) {
    list.add("jim");
  }
}

ArrayList<Number> list = new ArrayList<Number>();
Legacy.crash(list);
for (Object o: list)
  System.out.println(o); // ClassCastException: String
• “checked” collection classes perform type checks upon insertion rather than inadvertently later.

```java
class Legacy {
  static void crash (List list) {
    list.add("jim");
  }
}

ArrayList<Number> list = new ArrayList<Number>();
Legacy.crash(list);
for (Object o: list)
  System.out.println(o); // ClassCastException: String

List<Number> list = Collections.checkedList(new ArrayList<Number>(), Number.class);
Legacy.crash(list); // ClassCastException inside crash()
```
• Concepts
• Objectives
• Usage
• Pitfalls: Inheritance
• Pitfalls: Arrays
• **Pitfalls: static**
• Pitfalls: Factories
• Example: Lisp
• Example: Tree-based Container
• Summary and References
Fixing Common Pitfalls

```java
class Stack<E> {
    static class Node {
        final E info; final Node next;
        Node (E info, Node next) {
            this.info = info; this.next = next;
        }
    }
    Node top;
    void push (E info) {
        top = new Node(info, top);
    }
}
```
Fixing Common Pitfalls

• A static nested class cannot use a type parameter.

```java
class Stack<E> {
    static class Node {
        final E info; final Node next;
        Node (E info, Node next) {
            this.info = info; this.next = next;
        }
    }
    Node top;
    void push (E info) {
        top = new Node(info, top);
    }
}
```
Fixing Common Pitfalls

- A static nested class cannot use a type parameter.

```java
class Stack<E> {
  static class Node {
    final E info; final Node next;
    Node (E info, Node next) {
      this.info = info; this.next = next;
    }
  }
  Node top;
  void push (E info) {
    top = new Node(info, top);
  }
}
```

```java
class Stack<E> {
  static class Node<T> {
    final T info; final Node<T> next;
    Node (T info, Node<T> next) {
      this.info = info; this.next = next;
    }
  }
  Node<E> top;
  void push (E info) {
    top = new Node<E>(info, top);
  }
}
```
Fixing Common Pitfalls

```java
class Map<K,V> {
    interface Entry {
        K key ();
        V value ();
    }
}
```

This fact has nothing to do with generics specifically.
Fixing Common Pitfalls

- A nested interface is implicitly static.

```java
class Map<K,V> {
    interface Entry {
        K key ();
        V value ();
    }
}
```

This fact has nothing to do with generics specifically.
Fixing Common Pitfalls

• A nested interface is implicitly static.

```java
class Map<K,V> {
    interface Entry {
        K key ();
        V value ();
    }
}
```

make 17

This fact has nothing to do with generics specifically.
This is a strange example. Why would instances of X<E> be compared to E?
Fixing Common Pitfalls

• A static method cannot use a type parameter.

```java
class X<E> implements Comparable<E> {
  public int compareTo (E b) { return 1; }
  static int compareTo (X a, E b) {
    return a != null ? a.compareTo(b) : 0;
  }
}
```

This is a strange example. Why would instances of X<E> be compared to E?
Fixing Common Pitfalls

• A static method cannot use a type parameter.

```java
class X<E> implements Comparable<E> {
    public int compareTo (E b) { return 1; }
    static int compareTo (X a, E b) {
        return a != null ? a.compareTo(b) : 0;
    }
}
```

• [unchecked]

```java
class X<E> implements Comparable<E> {
    public int compareTo (E b) { return 1; }
    static <T> int compareTo (X a, T b) {
        return a != null ? a.compareTo(b) : 0;
    }
}
```

This is a strange example. Why would instances of X<E> be compared to E?
Fixing Common Pitfalls

- A static method cannot use a type parameter.

```java
class X<E> implements Comparable<E> {
    public int compareTo (E b) { return 1; }
    static int compareTo (X a, E b) {
        return a != null ? a.compareTo(b) : 0;
    }
}
```

- A raw class is not specific enough.

```java
class X<E> implements Comparable<E> {
    public int compareTo (E b) { return 1; }
    static <T> int compareTo (X a, T b) {
        return a != null ? a.compareTo(b) : 0;
    }
}
```

[unchecked]

This is a strange example. Why would instances of X<E> be compared to E?
Fixing Common Pitfalls

- A static method cannot use a type parameter.

```java
class X<E> implements Comparable<E> {
    public int compareTo (E b) { return 1; }
    static int compareTo (X a, E b) {
        return a != null ? a.compareTo(b) : 0;
    }
}
```

- A raw class is not specific enough.

```java
class X<E> implements Comparable<E> {
    public int compareTo (E b) { return 1; }
    static <T> int compareTo (X a, T b) {
        return a != null ? a.compareTo(b) : 0;
    }
}
```

This is a strange example. Why would instances of X<E> be compared to E?
- Concepts
- Objectives
- Usage
- Pitfalls: Inheritance
- Pitfalls: Arrays
- Pitfalls: static
- **Pitfalls: Factories**
- Example: Lisp
- Example: Tree-based Container
- Summary and References
Fixing Common Pitfalls

```java
class Factory<E> {
    E make () {
        return new E();
    }
}
```
Fixing Common Pitfalls

• A type parameter cannot be instantiated.

```java
class Factory<E> {
    E make () {
        return new E();
    }
}
```
Fixing Common Pitfalls

- A type parameter cannot be instantiated.

```java
class Factory<E> {
  E make () {
    return new E();
  }
}
```

```java
class Factory<E> {
  E make (Class c) throws Exception {
    return (E)c.newInstance();
  }
}
```
Fixing Common Pitfalls

- A type parameter cannot be instantiated.

```java
class Factory<E> {
    E make () {
        return new E();
    }
}
```

- Class is generic.

```java
class Factory<E> {
    E make (Class c) throws Exception {
        return (E)c.newInstance();
    }
}
```
Fixing Common Pitfalls

- A type parameter cannot be instantiated.

```java
class Factory<E> {
    E make () {
        return new E();
    }
}
```

- Class is generic.

```java
class Factory<E> {
    E make (Class c) throws Exception {
        return (E)c.newInstance();
    }
}
```

```java
class Factory<E> {
    E make (Class<E> c) throws Exception {
        return c.newInstance();
    }
}
```
• Concepts
• Objectives
• Usage
• Pitfalls: Inheritance
• Pitfalls: Arrays
• Pitfalls: static
• Pitfalls: Factories

**Example: Lisp**
• Example: Tree-based Container
• Summary and References
public class Cons<T,U> {
    public T car;
    public U cdr;

    public Cons( T x, U y ) {
      car = x; cdr = y;
    }

    public String toString() {
      return "(" + car + "." + cdr + ")";
    }
}
public class List<A> extends Cons<A,List<A>> {
    public List( A value ) { super( value, null ); }
    public List( A value, List<A> next ) { super( value, next ); }
    public String toString() {
        return "(" + toStringNoParens() + ")";
    }
    private String toStringNoParens() {
        if ( cdr == null ) return car.toString();
        else return car + " " + cdr.toStringNoParens();
    }
    // Another member function will go here.
}
public <B> List<Cons<A,B>> zip( List<B> other ) {
    if ( other == null )
      return null;
    else
      return ( cdr == null ) ?
        ? new List<Cons<A,B>>(
            new Cons<A,B>( car, other.car ),
            null )
        : new List<Cons<A,B>>(
            new Cons<A,B>( car, other.car ),
            cdr.zip( other.cdr ) );
  }
Output from a Test Program

is: (1 2 3 4)
cs: (Z Y X)
is|cs: ((1.Z) (2.Y) (3.X))
cs|is: ((Z.1) (Y.2) (X.3))
• Concepts
• Objectives
• Usage
• Pitfalls: Inheritance
• Pitfalls: Arrays
• Pitfalls: static
• Pitfalls: Factories
• Example: Lisp
• Example: Tree-based Container
• Summary and References
Container Example

```
public interface Container< G > extends Iterable< G > {
  public void add( G newValue );
  public boolean contains( G value );
}
```
/**
 * A set implemented with a binary search tree.
 * This code was designed to demonstrate generics.
 * Within you'll find:<UL>
 * <LI>A static generic member class</LI>
 * <LI>A static abstract generic member class</LI>
 * <LI>4 distinct anonymous subclasses of the above</LI>
 * <LI>2 distinct anonymous subclasses of Iterator&lt;G&gt;</LI>
 * </UL>
 * Removing values from a B.S.T. is not implemented here.
 *
 * @author James Heliotis
 * @author Axel Schreiner
 *
 * @param G the set's element type
 */

public class Set< G extends Comparable< G > > implements Container< G > {
public static void main( String[] args ) {
    if ( args.length == 0 ) return;
    Container<String> set = new Set<String>();
    String testString = args[0];
    System.out.println("\tSet does " +
            ( set.contains( testString ) ? "" : "not " ) +
            "contain " + testString + "." );
    for ( int i = 1; i < args.length; ++i ) {
        set.add( args[i] );
        System.out.println("Adding " + args[i] + "." );
        System.out.println("\tSet does " +
                ( set.contains( testString ) ? "" : "not " ) +
                "contain " + testString + "." );
    }
    for ( String s : set ) System.out.print( s + ' ' );
    System.out.println();
}
The Nested Static BSTNode Class

@ SuppressWarnings( "hiding" )
private static class BSTNode< G > {
    /** value storage */
    G value;
    /** the children */
    BSTNode< G > lesser, greater;

    BSTNode( G newValue ) { value = newValue; }

    // iterator() goes here
}

// iterator() goes here
/**
 * Data type that reports the result of a search.
 * If something was found, it can be fetched from a Location.
 * If not found, the Location knows where to put it.
 */

private static abstract class Location<T> {
    boolean found() { return false; }
    T get() { throw new UnsupportedOperationException(); }
    void set(T newValue) { throw new UnsupportedOperationException(); }
}
/**
 * Locate a value in the tree.
 * An object of an appropriate subclass of Location
 * is returned. They are all created anonymously.
 */

private Location< G > locate( G value ) {
    BSTNode< G > node = root;
    while ( node != null ) {
        final BSTNode< G > here = node;
        int comparison = value.compareTo( here.value );
        if ( comparison == 0 )
            return new Location<G>() {
                // found node
                boolean found() { return true; }
                G get() { return here.value; }
            };
    }
    // ... continued ...
}

Java
if ( comparison < 0 ) {
    if ( here.lesser == null )
        return new Location<G>() { // value should be put to the left
            void set( G newValue ) {
                here.lesser = new BSTNode< G >( newValue );
            }
        };
    node = node.lesser;
} else {
    if ( here.greater == null )
        return new Location<G>() { // value should be put to the right
            void set( G newValue ) {
                here.greater = new BSTNode< G >( newValue );
            }
        };
    node = node.greater;
}
return new Location<G>() { // no root
    void set( G newValue ) { root = new BSTNode< G >( newValue ); } 
};
Using Location Objects

```java
public void add( G newValue ) {
    Location< G > loc = locate( newValue );
    if ( !loc.found() )
      loc.set( newValue );
}
public boolean contains( G value ) {
    Location< G > loc = locate( value );
    return loc.found();
}
```
```java
/**
 * @return an iterator over the elements of the set.
 * If the set is not empty, an iterator on the root node is returned.
 * If the set is empty, a dead-end iterator is returned.
 */

public Iterator< G > iterator() {
    return root != null
        ? root.iterator()
        : new Iterator< G >() {
            public boolean hasNext () { return false; }
            public G next () { throw new NoSuchElementException(); }
            public void remove () {
                throw new UnsupportedOperationException();
            }
        };
}
```
enum Stage { left, info, right, done }

Iterator<
  iterator() {
    return new Iterator<
      Stage n = Stage.left;
      Iterator i = null;

      public boolean hasNext() // see below
        public G next() {
          if (hasNext()) {
            if (i != null) return i.next();
                      n = Stage.right; return value;
          }
          throw new NoSuchElementException();
        }
        public void remove() {
          throw new UnsupportedOperationException();
        }
    }
  }
}
public boolean hasNext () {
    for (;;) {
        if (i != null && i.hasNext())
            return true;
    else
        switch (n) {
            case left:
                n = Stage.info;
                i = lesser == null ? null : lesser.iterator();
                continue;
            case info:
                i = null;
                return true;
            case right:
                n = Stage.done;
                i = greater == null ? null : greater.iterator();
                continue;
            default:
                return false;
        }
    }
}
• Concepts
• Objectives
• Usage
• Pitfalls: Inheritance
• Pitfalls: Arrays
• Pitfalls: static
• Pitfalls: Factories
• Example: Lisp
• Example: Tree-based Container

• **Summary and References**
Summary

- Generics catch more type errors at compile time.
- C# generics are easier to use.
- Java generics are more space-efficient for reference types.
- C# generics are speed-efficient, even for primitive types.
- Java generics must rely on boxing for primitive types.
- Java generics interoperate with legacy code.
References

- **Bracha: Generics in the Java Programming Language (2004)**
  http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html

  http://download.microsoft.com/download/1/1/b/11b54b37-7b64-4f06-ad6a-d7ba081bf1d0/TLS320.ppt

- **Container Examples**

- **Sestoft: Java Precisely (2nd ed 2005)**
  http://www.dina.dk/~sestoft/javaprecisely/

- **Sestoft, Hansen: C# Precisely (2004)**
  http://www.dina.dk/~sestoft/csharpprecisely/