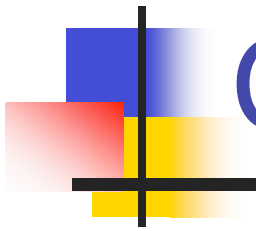


Generic Java

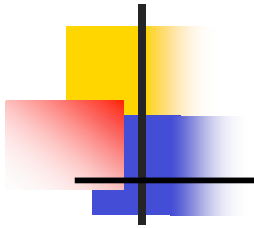
Generics in J2SE 5.0

Dave Landers

BEA Systems, Inc.

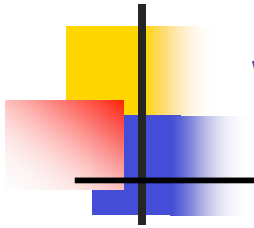
Colorado Software Summit

bea™



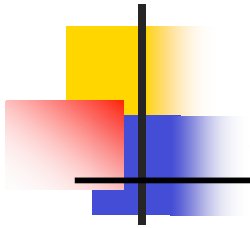
Generic Agenda

- The Basics
  - Using standard Generic Collections, *etc.*
- Writing your own Generic classes
  - Syntax
  - Maintaining legacy compatibility
  - Examples
- Implementation details
  - And resulting limitations
- Advanced syntax topics



What Is Generics?

- Allows abstraction of classes over Types
  - Common Example - Containers (Collections):
    - Container of <What> type
- Enables compiler to enforce Type Safety
  - Moves error checking from run-time (ClassCastException) to compile-time
- Improved Robustness
  - And Readability (?)
    - Yes, once you are familiar with syntax



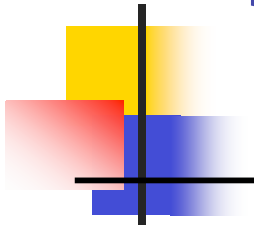
A Basic Look at Generics

- Before

```
List thingList = new ArrayList();
thingList.add( aThing );
Thing t = (Thing) thingList.get( 0 );
thingList.add( notAThing );
Thing t2 = (Thing) thingList.get( 1 ); // exception
```

- With Generics

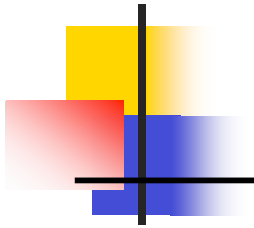
```
List<Thing> thingList = new ArrayList<Thing>();
thingList.add( aThing );
Thing t = thingList.get( 0 ); // no cast needed
thingList.add( notAThing );   // compile error
```



Just Less Typing, Fewer Casts?

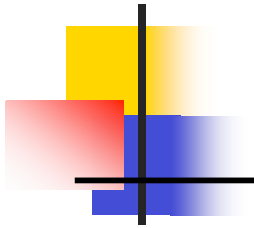
Or Rearranged Clutter? It's more than that…

- Compile-time type checking
  - Errors show up where errant object is inserted into collection
    - Rather than ClassCastException when it is retrieved
    - Error occurs *when* (compile) and *where* the problem is
- Enforcement of relationships
  - Between various objects used by classes or methods
- Declaration of programmer's intent
  - Code says how this instance of the class is to be used



Basic Generics Syntax

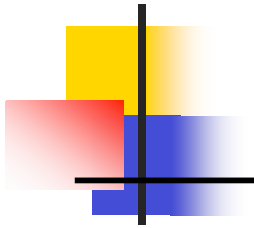
- Example - java.util.Map<K,V>
  - Map of Keys (K) and Values (V)
    - K and V are Parameterized Types
      - ✓ Convention is to use single-character capitalized symbols
    - Map.java uses K and V rather than "normal" types (Object)
  - public V put(K key, V val);
    - Put a K / V (key / value) pair in the map
  - public V get(Object key);
    - Get V (value) for the key
  - public Collection<V> values();
    - Returns a Collection of V (value) types
  - public Set<Map.Entry<K,V>> entrySet();
    - Returns a Set of Map.Entry containing K's and V's



Generic Classes

- class Foo<T> { void method(T arg); }
- Instances of Foo are parameterized by T
  - Foo is *generic*
    - Same behavior for any possible parameter T
    - Only one Foo.class file, only one Foo.class object
  - T is a Parameterized Type
    - Parameterization of Foo is handled by the compiler

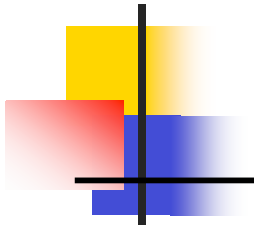
```
Foo<Bar> fb = new Foo<Bar>();
fb.method( aBar );  // OK
fb.method( aBaz );  // not allowed
```



Generic Methods

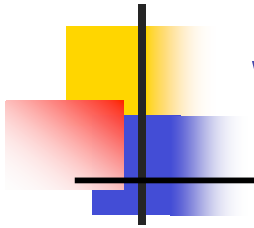
- <T> T method(T arg) { }
- Invocations of method() are parameterized by T
  - Compiler ensures arg and return are same type
    - When method itself is compiled
      - ✓ Ensures type safety within the method body
    - When someone makes a call to method:

```
Bar b = method( aBar );  // OK
Bar b = method( aBaz );  // not allowed
```



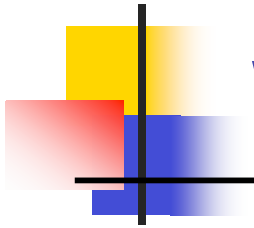
Code Break

- Examples of basic usage, compiler warnings, *etc.*
  - UseCollections
  - build.xml
  - BeGeneric



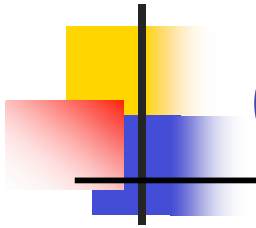
Where Will I Use This?

- Use standard Generic classes
  - Like Collections
  - Take advantage of extra type safety and compiler checks
- Write new classes using Generics
- Port existing classes to Generics



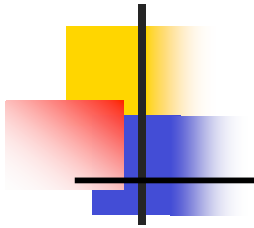
Writing Generic Classes

- Writing new classes using generics
  - Straight-forward
    - Except for that design thing…
- Applying Generics to existing classes
  - Maintaining backwards compatibility can get tricky
  - Highly likely to break a method signature
  - Have good unit tests before you touch anything
  - Might not be able to do exactly what you want



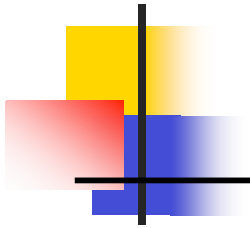
Generics Syntax

- The basics we have seen
  - Decorate class name with <T>
  - Use T where you would otherwise have Object
- Bounded type parameter
  - Decorate class name with <T extends Foo>
  - Use T where would otherwise have Foo
  - T is any subtype of Foo (or a Foo)
- Nested generic types
  - MyClass<T extends Map<Foo,Bar>>
- More
  - Wildcards, Multiple Bounds, Lower Bounds
  - Later…



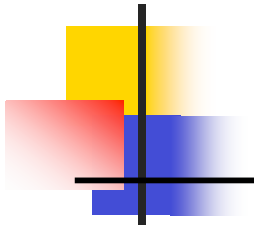
Code Break

- Examples of coding with generics
  - BeGeneric
  - PropertyView
  - Table



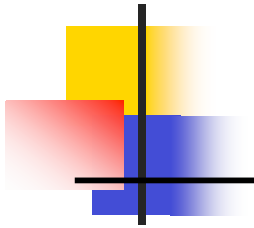
More Syntax

- Wildcards
  - <?>
- Bounded Wildcards
  - <? extends Foo>
- Lower Bound
  - <? super Bar>



Simple Wildcards

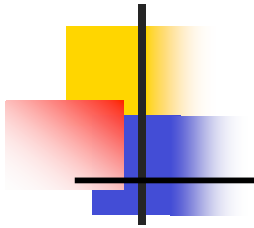
- void doIt(Collection c)
  - Not generic
- void doIt(Collection<Object> c)
  - This only works with Collection<Object>
    - Or plain Collection with unsafe warning
  - Collection<Foo> is not a Collection<Object>
    - c.add(someObject) not allowed for Collection<Foo>



Simple Wildcards <?>

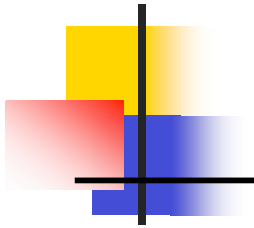
- void doIt(Collection<?> c)
  - "Collection of Unknown"
  - c.get() returns an Object
  - c.add(o) not allowed since we don't know the type of the <?> parameter

- Difference between using Collection and Collection<?>
  - Read only usages in method body



Bounded Wildcards

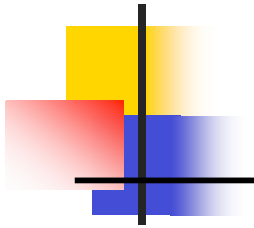
- Upper Bounds <? extends Foo>
  - Wildcard can be any subclass of Foo
  - Foo is the upper bound of the wildcard
- void doIt(Collection<Base> c);
  - Can not call this with anything except a Collection<Base>
  - Collection<Sub> won't allow c.add(aBase)
- void doIt(Collection<? extends Base> c);
  - Can be any subclass of Base
  - As with unbounded wildcards, c is "read-only"



Generic Methods

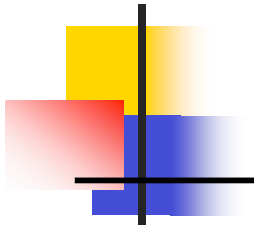
- To specify relationships between method arguments and parameters
- Type Parameter on single method not class

- static <T> T grabIt(T[] all, int which)
  - <T> declares a generic method using type T
  - Takes an array of T, returns a T
  - Each invocation of grabIt() can have a different type for T



Generic Methods with Bounds

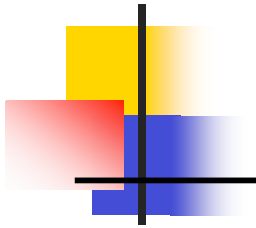
- Generic method with type T, argument with bounds
  - <T> T grabIt(List<? extends T> stuff, T matchMe)
- Same thing with multiple types
  - <T, Q extends T> T grabIt(List<Q> stuff, T matchMe)
- Generic method with bounded type T
  - <T extends Number> Set<T> lessThan(Set<T> stuff, T max)
  - Equivalent to:
    - Set lessThan(Set stuff, Number max)
      - ✓ Except all Sets must be Set<Number>
      - ✓ And the Numbers must be same type (all Integer, or Float, *etc.*)



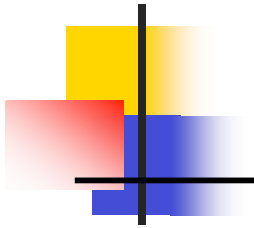
Code Break

- Examples of wildcards and generic methods
  - Bag
  - BeGeneric methods

Generics Implementation and Resulting Limitations

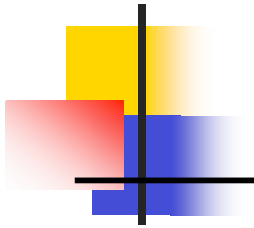


- Generics is a Compile-time feature
- Compiler implements the type checking
  - Good thing - catch errors at compile time
  - Results in some limitations and occasional surprises



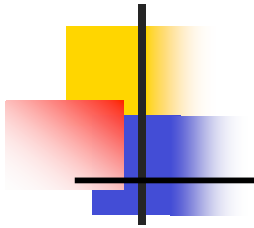
Compile-Time Generics

- Type information for the *class* is used by compiler
  - And available by reflection on the Class
  - Not used at runtime
  - Erased from the method signatures
- Type information is *not* held by any object *instances*
  - Parameterization of instances is only known by compiler
  - At run-time, List foo and List<String> bar are both just Lists
    - Only the compiler can tell



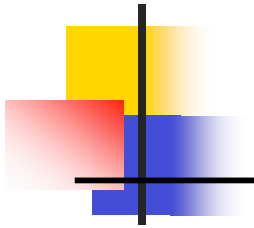
Erasure

- Erasure is the process of "erasing" the generics information during compile
- What is the "real" method signature
  - class Foo<T extends Number> {
    T m(Set<? extends T> s) {} }
  - class Foo { Number m(Set s) {} }
- Understanding this is critical for maintaining backwards & migration compatibility
  - javap is helpful



Generics *vs.* C++ Templates

- Similarities:
  - Syntax <>
- Differences:
  - Generics is a compile-time feature
    - Object instances do not have Type information outside the compiler
  - List<Integer> and List<Thing> are the same Class type
    - `List<Integer>.class == List<Thing>.class`
    - Dynamic Class instances are *not* created for each generic usage



One Class Object - Loophole?

- Since List and List<Integer> are the same Class, you can do this:

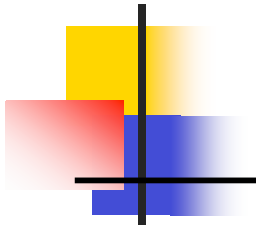
```
List<Integer> intList = new ArrayList<Integer>();
List wideOpen = intList;
wideOpen.add( someObject ); // A Loophole!
Integer ii = intList.get( 0 ); // ClassCastException - Oh No!
```

- Yes… but:
  - Compiler generates warnings

```
wideOpen.add( someObject );  // Compiler warning here
```

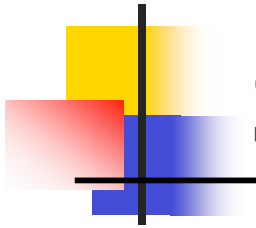
  - No less safe than code without Generics
    - Haven't lost anything (you could do this before, too)
    - Still get the same runtime check (ClassCastException) when retrieving "bad" data from the list

Generic Types, Subtypes, and instanceof

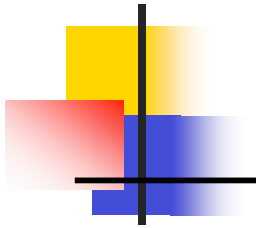


- Q:
  - List<String> strList = …;
  - Is strList instanceof List<Object> ?
- A: ? Yes, No, Maybe, … whatever
  - Whatever: You can't use instanceof with generic types
- However…
  - Yes: Both are the same Class at runtime
  - No: The compiler considers them to be different
    - List<String> is not compatible with List<Object>

More on Generics and Subtypes

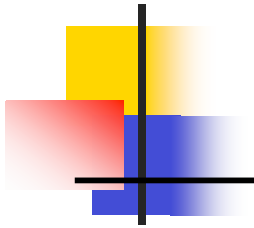


- You can't do this either:
  - List<String> strList = …;
  - List<Object> objList = strList; // compile failure
- Seems counter-intuitive
  - String is an Object, so why isn't List<String> a List<Object>
- Because it breaks compile-time type safety
  - objList.add(anObject);
  - String strList = strList.get(0); // Might be Object



More Generic Limitations

- Can not construct instances of generic types
  - new T(); // not allowed
  - Don't know what type to create at runtime
  - Don't know constructor signatures
  - Can use Factory methods and Class tokens
    - Later…
- Parameterized Types are Objects
  - Not Primitives
  - Autoboxing can help here
- Static methods and fields can not refer to the class's Parameterized Types
  - Can have static generic methods

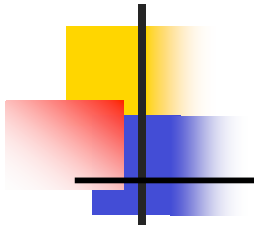


Generic Arrays

- Component type of Array can not be parameterized
  - List<Foo>[] not allowed
    - Arrays are just Objects, can get around type safety

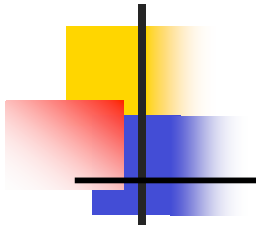
```
List<String>[] lsa = new List<String>[10]; // Not allowed
Object o = lsa;  // arrays are just objects
Object[] oa = (Object[]) o;  // this object is an array
List<Integer> li = new ArrayList<Integer>();
li.add(3);
oa[0] = li;
String s = lsa[0].get(0); // run-time error
```

  - List<?>[] - Unbounded wildcards are allowed
  - Arrays whose element type is a variable are allowed
    - void method(T[] args);



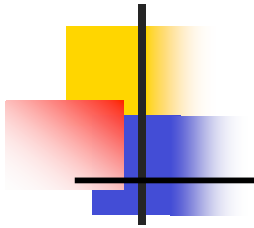
More Topics

- Multiple Bounds
  - And Erasure
- Lower Bounds
- .class and Class<T>



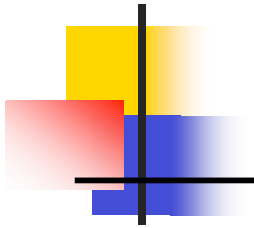
Multiple Bounds

- Can have multiple bounds on generic type
- <T extends Foo & Comparable<T>>
  - T must extend Foo and implement Comparable<T>
- <? extends Foo & Comparable<T>>
  - Similar, with wildcard
  - Signature erases to first type
  - Compiler enforces other types



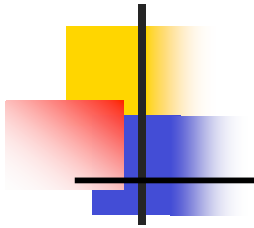
Erasure and Multiple Bounds

- Sometimes you need "trickery" to
  - Do what you want with generics, AND
  - Maintain backwards compatibility in method signatures
- Multiple bounds can help
  - Erasure uses the first type in the signature
- <T extends Comparable<T>>
  - T is a subclass of Comparable<T>
  - Erased type is Comparable
- <T extends Object & Comparable<T>>
  - Effectively the same thing
  - Erased type is now Object



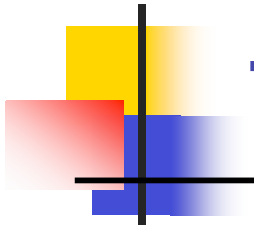
Lower Bounded Wildcards

- <? super T>
- Represents a type which is a superclass of T
  - Or a T
- "Opposite" of extends
- Helpful to rearrange argument types



Need for Super

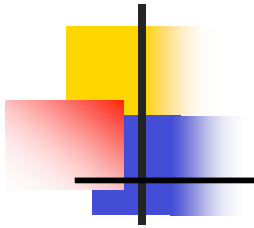
- Object copySet(Set src, Set dest);
  - Copy entries from src to dest, returning the last one
- T copySet(Set<T> src, Set<T> dest);
  - The src and dest sets must be same type
  - Maybe dest is a supertype of src
- T copySet(Set<? extends T> src, Set <T> dest);
  - Works, but return type is the supertype (less specific)
  - Will require a cast
- T copySet(Set<T> src, Set<? super T> dest);
  - Src entries must be superclass of T (assignable from)
  - Now return type matches the src Set
  - String last = copySet(stringSet, objectSet);



The .class Literal

- java.lang.Class is generic: Class<T>
  - Class literal operator understands this
  - Foo.class results in Class<Foo>
- public T newInstance();
- public T cast(Object o);
  - Useful in factory method
  - Use Class objects as “tokens” for types
  - Since you can’t use new with generics

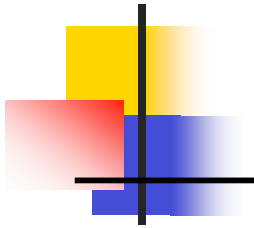
```
T foo = new T();  // can’t do this
```



Class<T> and newInstance

```
public static <T> Collection<T>
    fill( Class<T> clazzT, int count ) {
  Collection<T> data = new ArrayList<T>();
  for ( int I = 0; I < count; ++I ) {
    // create a new T - can't do new T(), but can do
    // Class.newInstance, assuming a no-args c'tor
    T item = clazzT.newInstance();
    data.add( item );
  }
  return data;
}

Collection<Dog> tenDogs = fill( Dog.class, 10 );
```

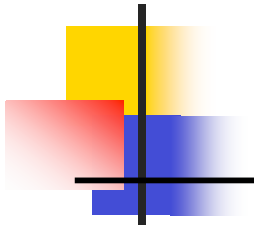


Code Break

- Examples
  - Erasure
  - Multiple Bounds
  - Lower Bounds
  - .class literal and factory
  - ThingFactory
  - Bag.max()

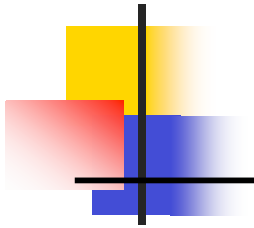
Other Related J2SE 5.0 Features

- Autoboxing
  - List intList = new ArrayList<Integer>();
  - intList.add(3); // Integer.valueOf(3), but with caching
  - int x = intList.get(0);
- For loop
  - for (Integer eachOne : intList) { … }
- Covariant Return Types
  - public Foo clone() {…} // Foo rather than Object
  - Foo foo2 = foo1.clone(); // no cast needed
- Annotations - especially @Override
  - Make sure you're overriding the method you intend
    - Especially considering Erasure, Translation, Covariant methods…



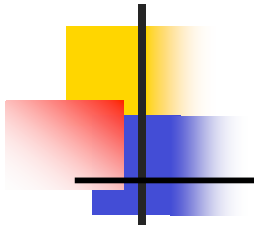
Generic Summary

- Move error checking from run-time to compile-time
  - More robust code
  - Didn't loose any runtime safety
- Compile-time feature
  - Generic type information is Erased from signatures
- Foo<Bar> and Foo<Baz> are incompatible types
  - Although they are both erased to just Foo
- Use Generics where it can make your code better
- Experiment and have fun



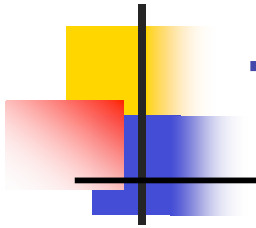
References

- Generics Tutorial
  - http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
- Generics FAQ
  - http://www.langer.camelot.de/GenericsFAQ/JavaGenericsFAQ.html



Other Related Sessions

- Bruce Eckel
  - Issues…
    - Generics, Templates, Latent Typing…
- Mark Reinhold
  - The Rest of Tiger
    - Other J2SE 5.0 features
- Donald Smith
  - Caging the Tiger
    - Persistence



The End

- Please fill out the evaluations
- Example code available
  - On the conference CDROM
  - http://boulderites.bea.com/~landers
    - References there, too

Colorado Software Summit

dave.landers@bea.com