GNU Classpath and Java 1.5
Andrew Hughes

http://planet.classpath.org
Outline

1. Java 1.5
   - Generics
   - Enumerations
   - Annotations
   - Other Additions
Outline

1 Java 1.5
   - Generics
   - Enumerations
   - Annotations
   - Other Additions

2 GNU Classpath
   - The Generics Branch
   - Virtual Machines and Compilers
   - The Future
Outline

1. Java 1.5
   - Generics
   - Enumerations
   - Annotations
   - Other Additions

2. GNU Classpath
   - The Generics Branch
   - Virtual Machines and Compilers
   - The Future
‘the largest set of changes in the language’s history’

---

1 Preface to the Third Edition of the Java Language Specification
“the largest set of changes in the language’s history”\(^1\)

Three major changes:

\(^1\)Preface to the Third Edition of the Java Language Specification
‘the largest set of changes in the language’s history’ \(^1\)

Three major changes:

- Generics

\(^1\)Preface to the Third Edition of the Java Language Specification
Java 1.5

‘the largest set of changes in the language’s history’ \(^1\)

Three major changes:

- Generics
- Annotations

\(^1\)Preface to the Third Edition of the Java Language Specification
Java 1.5

‘the largest set of changes in the language’s history’

Three major changes:

- Generics
- Annotations
- Enumerations

---

¹Preface to the Third Edition of the Java Language Specification
‘the largest set of changes in the language’s history’ ¹

Three major changes:

- Generics
- Annotations
- Enumerations

Plus lots of syntactic sugar

¹Preface to the Third Edition of the Java Language Specification
Type Variables

- The term ‘generics’ basically refers to the introduction of type variables.
The term ‘generics’ basically refers to the introduction of type variables.

An instance of a generic class contains both:
Type Variables

- The term 'generics' basically refers to the introduction of type variables.
- An instance of a generic class contains both:
  - Normal instance variables with values
The term 'generics' basically refers to the introduction of type variables.

An instance of a generic class contains both:
- Normal instance variables with values
- Type variables instantiated with a particular type.
Type Variables

- The term ‘generics’ basically refers to the introduction of type variables.
- An instance of a generic class contains both:
  - Normal instance variables with values
  - Type variables instantiated with a particular type.
- Thus, a class can be defined that generalizes over certain types.
The term ‘generics’ basically refers to the introduction of type variables.

An instance of a generic class contains both:
- Normal instance variables with values
- Type variables instantiated with a particular type.

Thus, a class can be defined that generalizes over certain types.

Derives from work on GJ by (Bracha, Odersky, Stoutamire and Wadler, 1998) – see the proceedings of OOPSLA in ACM SIGPLAN Notices 33 (10) pages 183–200.
Parameterized Types

- Type variables can be associated with:
Parameterized Types

- Type variables can be associated with:
  - Classes and Interfaces
Parameterized Types

- Type variables can be associated with:
  - Classes and Interfaces
  - Constructors and Methods
Parameterized Types

- Type variables can be associated with:
  - Classes and Interfaces
  - Constructors and Methods
- Variables which use a generic class as their type are referred to as *parameterized types*. 
Parameterized Types

- Type variables can be associated with:
  - Classes and Interfaces
  - Constructors and Methods

- Variables which use a generic class as their type are referred to as *parameterized types*.

- That is, the exact type used is determined by the parameters used to instantiate the type variables.
Parameterized Types

- Type variables can be associated with:
  - Classes and Interfaces
  - Constructors and Methods

- Variables which use a generic class as their type are referred to as parameterized types.

- That is, the exact type used is determined by the parameters used to instantiate the type variables.

Example

```
List<String> names;
```
How Is This Useful?

- Collections benefit most obviously from type variables.
How Is This Useful?

- Collections benefit most obviously from type variables.
- When defining a collection of things, we want to say what type of thing we want to put in the collection.
How Is This Useful?

- Collections benefit most obviously from type variables.
- When defining a collection of things, we want to say what type of thing we want to put in the collection.
- Prior to Java 1.5, we can’t
How Is This Useful?

- Collections benefit most obviously from type variables.
- When defining a collection of things, we want to say what type of thing we want to put in the collection.
- Prior to Java 1.5, we can’t
- Instead, we just call everything in the collection an instance of the common supertype, `Object`
How Is This Useful?

This leads to some nasty code:

```java
String name = (String) names.get(3);
```

With Java 1.5, collections use type variables. So lists contain elements of a type variable, `E`, rather than of a specific type, `Object`. The type of `names` will then be `List<String>`, where `String` is the parameter used to give `E` its value.
How Is This Useful?

- This leads to some nasty code:

```
Example

String name = (String) names.get(3);
```
This leads to some nasty code:

```
String name = (String) names.get(3);
```

With Java 1.5, collections use type variables.
How Is This Useful?

- This leads to some nasty code:

  **Example**
  ```java
  String name = (String) names.get(3);
  ```

- With Java 1.5, collections use type variables.
- So lists contain elements of a type variable, \(E\), rather than of a specific type, `Object`
How Is This Useful?

- This leads to some nasty code:

```java
String name = (String) names.get(3);
```

- With Java 1.5, collections use type variables.
- So lists contain elements of a type variable, `E`, rather than of a specific type, `Object`.
- The type of `names` will then be `List<String>`, where `String` is the parameter used to give `E` its value.
Another Example

Generics also give the flexibility to design new types that would be difficult to realise fully before.
Another Example

- Generics also give the flexibility to design new types that would be difficult to realise fully before.
- For example, we can have a heterogenous pair type, `Pair<A,B>`.
Another Example

- Generics also give the flexibility to design new types that would be difficult to realise fully before.
- For example, we can have a heterogenous pair type, Pair\langle A, B \rangle.

```java
public class Pair\lt A, B \gt  {
    A left;
    B right;
    A getLeft();
    B getRight();
}
```
Wildcards allow us to say that we don’t care how a particular type variable is being used.
Wildcards allow us to say that we don’t care how a particular type variable is being used.

We can’t use `Object` for this, as `Object` is a type itself.
Wildcards

- Wildcards allow us to say that we don’t care how a particular type variable is being used.
- We can’t use `Object` for this, as `Object` is a type itself.
- This allows us to still define general methods, such as those which print the contents of a collection.
Wildcards allow us to say that we don’t care how a particular type variable is being used.

We can’t use `Object` for this, as `Object` is a type itself.

This allows us to still define general methods, such as those which print the contents of a collection.

Example

```java
public void printCollection(Collection<??> c)
```
Bounded Type Variables

Type variables can be *bounded*
Bounded Type Variables

- Type variables can be *bounded*
- By default, the type used to instantiate a type variable must be a subclass of `Object`.
Bounded Type Variables

- Type variables can be *bounded*
- By default, the type used to instantiate a type variable must be a subclass of `Object`.
- So, anything basically...
Bounded Type Variables

- Type variables can be *bounded*
- By default, the type used to instantiate a type variable must be a subclass of `Object`.
- So, anything basically...
- But we can instead say that the type must be a subclass of something else.
Bounded Type Variables

- Type variables can be *bounded*
- By default, the type used to instantiate a type variable must be a subclass of `Object`.
- So, anything basically...
- But we can instead say that the type must be a subclass of something else.
- For example, we want to ensure that adding the contents of a collection doesn’t break our nice well-typed collections...
Bounded Type Variables

- Type variables can be *bounded*
- By default, the type used to instantiate a type variable must be a subclass of `Object`.
- So, anything basically...
- But we can instead say that the type must be a subclass of something else.
- For example, we want to ensure that adding the contents of a collection doesn’t break our nice well-typed collections...

**Example**

```java
public boolean addAll(Collection<? extends E> c)
```
Bounded Type Variables

- Type variables can be *bounded*
- By default, the type used to instantiate a type variable must be a subclass of `Object`.
- So, anything basically...
- But we can instead say that the type must be a subclass of something else.
- For example, we want to ensure that adding the contents of a collection doesn’t break our nice well-typed collections...

Example

```java
public boolean addAll(Collection<? extends E> c)
```

- The lack of inferencing with wildcards means that they can also have a lower bound, specified by `super`.
The Chains of Backward Compatibility

- Generics are very nice in theory...
The Chains of Backward Compatibility

- Generics are very nice in theory...
- ...unfortunately, the implementation suffers from having to maintain compatibility with earlier versions.
Generics are very nice in theory . . .

. . . unfortunately, the implementation suffers from having to maintain compatibility with earlier versions.

Essentially, none of this exists at run-time in the bytecode.
Type Erasure

- A type variable is erased to become its lowest bound.
Type Erasure

- A type variable is erased to become its lowest bound.
- In most cases, this will be the all to familiar `Object`.
Type Erasure

- A type variable is erased to become its lowest bound.
- In most cases, this will be the all too familiar `Object`.
- Thus, the bytecode still only includes collections of objects.
Type Erasure

- A type variable is erased to become its lowest bound.
- In most cases, this will be the all to familiar `Object`.
- Thus, the bytecode still only includes collections of objects.
- The difference is the compiler is nice enough to put the casts in for us, and check that we don’t do something stupid.
Type Erasure

- A type variable is erased to become its lowest bound.
- In most cases, this will be the all to familiar `Object`.
- Thus, the bytecode still only includes collections of objects.
- The difference is the compiler is nice enough to put the casts in for us, and check that we don’t do something stupid.
- The lack of type variables at run-time effectively puts an end to some constructs such as `new T[]`.

Andrew Hughes  http://planet.classpath.org  GNU Classpath and Java 1.5
Type Erasure

- A type variable is erased to become its lowest bound.
- In most cases, this will be the all to familiar `Object`.
- Thus, the bytecode still only includes collections of objects.
- The difference is the compiler is nice enough to put the casts in for us, and check that we don’t do something stupid.
- The lack of type variables at run-time effectively puts an end to some constructs such as `new T[]`.
- Instead...
Type Erasure

- A type variable is erased to become its lowest bound.
- In most cases, this will be the all to familiar `Object`.
- Thus, the bytecode still only includes collections of objects.
- The difference is the compiler is nice enough to put the casts in for us, and check that we don’t do something stupid.
- The lack of type variables at run-time effectively puts an end to some constructs such as `new T[]`.
- Instead...

Example

```java
T[] stuff = (T[]) new Object[3];
```
Raw Types

- Backwards compatibility means we must also have raw types
Raw Types

- Backwards compatibility means we must also have *raw types*
- Raw types are the variant of a generic class where the type variables are not specified.
Raw Types

- Backwards compatibility means we must also have *raw types*
- Raw types are the variant of a generic class where the type variables are not specified.
- More simply, *List* is the raw type of *List<E>*.
Raw Types

- Backwards compatibility means we must also have *raw types*.
- Raw types are the variant of a generic class where the type variables are not specified.
- More simply, `List` is the raw type of `List<E>`.
- This allows all the old code to still work...
Raw Types

- Backwards compatibility means we must also have *raw types*
- Raw types are the variant of a generic class where the type variables are not specified.
- More simply, *List* is the raw type of *List*<E>.
- This allows all the old code to still work...
- ...but means that all the type safety gets thrown away as soon as a list is passed without its type variables.
Raw Types

- Backwards compatibility means we must also have *raw types*
- Raw types are the variant of a generic class where the type variables are not specified.
- More simply, `List` is the raw type of `List<E>`.
- This allows all the old code to still work...
- ...but means that all the type safety gets thrown away as soon as a list is passed without its type variables.
- In the class library, there are lots of places where collisions occur between raw types, types with type variables and types that haven’t been converted.
Raw Types

- Backwards compatibility means we must also have raw types.
- Raw types are the variant of a generic class where the type variables are not specified.
- More simply, `List` is the raw type of `List<E>`.
- This allows all the old code to still work...
- ...but means that all the type safety gets thrown away as soon as a list is passed without its type variables.
- In the class library, there are lots of places where collisions occur between raw types, types with type variables and types that haven’t been converted.
- The result is fairly nasty, and leads to some interesting compiler messages. Unfortunately, we’ve lost some casts for other more strange ones...
The addition of enumerations to the language provides a safe way of defining types with a set range of values.

Example
```
enum Color {red, green, blue;}
```

Pre-defined instances of a compiler-generated class are used instead of integers alone to ensure greater type safety.

Enumerations can be used in `switch` statements.
The addition of enumerations to the language provides a safe way of defining types with a set range of values. This avoids the need for doing the same thing via constants, and having to provide validity checks.
The addition of enumerations to the language provides a safe way of defining types with a set range of values. This avoids the need for doing the same thing via constants, and having to provide validity checks. Instead, only valid values will type correctly.
Enumerations

- The addition of enumerations to the language provides a safe way of defining types with a set range of values.
- This avoids the need for doing the same thing via constants, and having to provide validity checks.
- Instead, only valid values will type correctly.
- The C syntax is used:
The addition of enumerations to the language provides a safe way of defining types with a set range of values. This avoids the need for doing the same thing via constants, and having to provide validity checks. Instead, only valid values will type correctly. The C syntax is used:

```java
enum Color {red, green, blue;}
```
The addition of enumerations to the language provides a safe way of defining types with a set range of values. This avoids the need for doing the same thing via constants, and having to provide validity checks. Instead, only valid values will type correctly.

The C syntax is used:

```java
enum Color {red, green, blue;}
```

Pre-defined instances of a compiler-generated class are used instead of integers alone to ensure greater type safety.
The addition of enumerations to the language provides a safe way of defining types with a set range of values. This avoids the need for doing the same thing via constants, and having to provide validity checks. Instead, only valid values will type correctly. The C syntax is used:

```
enum Color {red, green, blue;}
```

Pre-defined instances of a compiler-generated class are used instead of integers alone to ensure greater type safety. Enumerations can be used in `switch` statements.
Annotations

- Annotations essentially allow arbitrary metadata to be associated with a class.
Annotations

- Annotations essentially allow arbitrary metadata to be associated with a class.
- For example, deprecation can now be determined programmatically via the use of the `@Deprecation` annotation.
Annotations

- Annotations essentially allow arbitrary metadata to be associated with a class.
- For example, deprecation can now be determined programmatically via the use of the `@Deprecation` annotation.
- These will be more fully realised when they are used in code that makes use of such runtime data.
Annotations

- Annotations essentially allow arbitrary metadata to be associated with a class.
- For example, deprecation can now be determined programmatically via the use of the `@Deprecation` annotation.
- These will be more fully realised when they are used in code that makes use of such runtime data.
- For example, the new version of the web services stack will make use of annotations.
Annotations

- Annotations essentially allow arbitrary metadata to be associated with a class.
- For example, deprecation can now be determined programmatically via the use of the @Deprecation annotation.
- These will be more fully realised when they are used in code that makes use of such runtime data.
- For example, the new version of the web services stack will make use of annotations.
- Classes like Serializable and Remote would probably have been annotations, if they existed earlier.
Annotations

- Annotations essentially allow arbitrary metadata to be associated with a class.
- For example, deprecation can now be determined programmatically via the use of the `@Deprecation` annotation.
- These will be more fully realised when they are used in code that makes use of such runtime data.
- For example, the new version of the web services stack will make use of annotations.
- Classes like `Serializable` and `Remote` would probably have been annotations, if they existed earlier.
- They are realised as a restricted form of interface (which causes problems when they need to be instantiated...)

Andrew Hughes  http://planet.classpath.org  GNU Classpath and Java 1.5
Syntactic Sugar

*Autoboxing*, and its converse, *auto-unboxing*, provide automatic conversion between primitive types (such as `int`) and their wrapper type (`Integer` in this case).
Syntactic Sugar

- **Autoboxing**, and its converse, **auto-unboxing**, provide automatic conversion between primitive types (such as `int`) and their wrapper type (`Integer` in this case).
- A new **for** loop syntax simplifies iterating over collections.
Syntactic Sugar

- **Autoboxing**, and its converse, **auto-unboxing**, provide automatic conversion between primitive types (such as `int`) and their wrapper type (`Integer` in this case).
- A new `for` loop syntax simplifies iterating over collections.

Example

```java
for (String n : names)
```

Andrew Hughes http://planet.classpath.org GNU Classpath and Java 1.5
Syntactic Sugar

- Autoboxing, and its converse, auto-unboxing, provide automatic conversion between primitive types (such as `int`) and their wrapper type (`Integer` in this case).
- A new `for` loop syntax simplifies iterating over collections.

**Example**

```java
for (String n : names)
```

- This works with anything that implements `Iterable` (including arrays)
Methods

- A method can now take a variable number of arguments.
A method can now take a variable number of arguments. This allows Java to have its equivalent of `printf`.
Methods

- A method can now take a variable number of arguments.
- This allows Java to have its equivalent of `printf`

Example

```
System.out.printf("Hello %s", name)
```
A method can now take a variable number of arguments. This allows Java to have its equivalent of `printf`.

**Example**

```java
System.out.printf("Hello %s", name)
```

Methods are also now allowed to have *covariant* return types.
Methods

- A method can now take a variable number of arguments.
- This allows Java to have its equivalent of `printf`.

**Example**

```
System.out.printf("Hello %s", name)
```

- Methods are also now allowed to have *covariant* return types.
- So a return type in a subclass method can be more specific than that in the superclass.
Methods

- A method can now take a variable number of arguments.
- This allows Java to have its equivalent of `printf`.

Example

```
System.out.printf("Hello %s", name)
```

- Methods are also now allowed to have `covariant` return types.
- So a return type in a subclass method can be more specific than that in the superclass.
- This is used by `Appendable`.
Outline

1. Java 1.5
   - Generics
   - Enumerations
   - Annotations
   - Other Additions

2. GNU Classpath
   - The Generics Branch
   - Virtual Machines and Compilers
   - The Future
GNU Classpath aims to provide a Free Software implementation of the class libraries used by virtual machines and compilers for the Java programming language.

\[^2\text{http://builder.classpath.org/japi/jdk14-classpath.html as of 2006/08/15}\]
The GNU Classpath Project

- GNU Classpath aims to provide a Free Software implementation of the class libraries used by virtual machines and compilers for the Java programming language.
- Started in 1998 with five developers. There are now roughly twenty active developers and over seventy registered.

\[2^{nd}\] http://builder.classpath.org/japi/jdk14-classpath.html as of 2006/08/15
The GNU Classpath Project

- GNU Classpath aims to provide a Free Software implementation of the class libraries used by virtual machines and compilers for the Java programming language.
- Started in 1998 with five developers. There are now roughly twenty active developers and over seventy registered.
- About eight years later, current CVS has 99.09%\(^2\) binary compatibility with version 1.4.

\(^2\)http://builder.classpath.org/japi/jdk14-classpath.html as of 2006/08/15
The GNU Classpath Project

- GNU Classpath aims to provide a Free Software implementation of the class libraries used by virtual machines and compilers for the Java programming language.
- Started in 1998 with five developers. There are now roughly twenty active developers and over seventy registered.
- About eight years later, current CVS has 99.09%\(^2\) binary compatibility with version 1.4.
- Applications such as Eclipse and JEdit can be run in a Free environment using GNU Classpath.

\(^2\)http://builder.classpath.org/japi/jdk14-classpath.html as of 2006/08/15
GNU Classpath aims to provide a Free Software implementation of the class libraries used by virtual machines and compilers for the Java programming language.

Started in 1998 with five developers. There are now roughly twenty active developers and over seventy registered.

About eight years later, current CVS has 99.09%\(^2\) binary compatibility with version 1.4.

Applications such as **Eclipse** and **JEdit** can be run in a Free environment using GNU Classpath.

The most recent version, 0.92, was released on the 10th of August (just last week)

\(^2\)http://builder.classpath.org/japi/jdk14-classpath.html as of 2006/08/15
GNU Classpath is distributed with many GNU/Linux distributions, something which isn’t possible with Sun’s implementation.
Why?

- GNU Classpath is distributed with many GNU/Linux distributions, something which isn’t possible with Sun’s implementation.
- The licensing of GNU Classpath makes it more suitable for research.
Why?

- GNU Classpath is distributed with many GNU/Linux distributions, something which isn’t possible with Sun’s implementation.
- The licensing of GNU Classpath makes it more suitable for research.
- **Kaffe** and **JikesRVM** are used for academic research and both utilise GNU Classpath as their class library.
Why?

- GNU Classpath is distributed with many GNU/Linux distributions, something which isn’t possible with Sun’s implementation.
- The licensing of GNU Classpath makes it more suitable for research.
- **Kaffe** and **JikesRVM** are used for academic research and both utilise GNU Classpath as their class library.
- **GCJ** offers a different approach in providing ahead-of-time (AOT) compilation of Java code.
Why?

- GNU Classpath is distributed with many GNU/Linux distributions, something which isn’t possible with Sun’s implementation.
- The licensing of GNU Classpath makes it more suitable for research.
- Kaffe and JikesRVM are used for academic research and both utilise GNU Classpath as their class library.
- GCJ offers a different approach in providing ahead-of-time (AOT) compilation of Java code.
- Going further, the JNode project are basing an entire operating system around the language and GNU Classpath. Also, the IKVM.NET virtual machine is implemented on top of the .NET framework.
The generics branch was started by Tom Tromey on the 5th of August, 2004:
The generics branch was started by Tom Tromey on the 5th of August, 2004:

‘I’m going to make a branch for generics work, including all the other parts of 1.5; for the most part these rely on generics and can’t be done independently anyway. I plan to name it “generics-branch”’.

Andrew Hughes  http://planet.classpath.org  GNU Classpath and Java 1.5
The generics branch was started by Tom Tromey on the 5th of August, 2004:

‘I’m going to make a branch for generics work, including all the other parts of 1.5; for the most part these rely on generics and can’t be done independently anyway. I plan to name it “generics-branch”.

I started contributing to the branch shortly afterwards.
The generics branch was started by Tom Tromey on the 5th of August, 2004:

‘I’m going to make a branch for generics work, including all the other parts of 1.5; for the most part these rely on generics and can’t be done independently anyway. I plan to name it “generics-branch”’.

I started contributing to the branch shortly afterwards.

It compiled for the first time on the 13th of January, 2005 (including most of the changes to the collections hierarchy)
The Beginning

- The generics branch was started by **Tom Tromey** on the 5th of August, 2004:
  - ‘I’m going to make a branch for generics work, including all the other parts of 1.5; for the most part these rely on generics and can’t be done independently anyway. I plan to name it “generics-branch”’.
- I started contributing to the branch shortly afterwards.
- It compiled for the first time on the 13th of January, 2005 (including most of the changes to the collections hierarchy)
- Shortly after, I started getting the branch back in sync with the main development branch, and having continued this to the present day. Recent exponential rises in GNU Classpath development haven’t helped in this case!
The Rise and Rise of GNU Classpath Development
Two years later, the current generics branch has 94.67%\(^3\) binary compatibility with version 1.5.

\(^3\)http://builder.classpath.org/japi/jdk15-generics.html as of 2006/08/15
Two years later, the current generics branch has 94.67\%[^3] binary compatibility with version 1.5.

The branch has evolved to only contain changes which won’t compile with 1.4 and earlier compilers, rather than all of the 1.5 changes.

[^3]: http://builder.classpath.org/japi/jdk15-generics.html as of 2006/08/15
Two years later, the current generics branch has 94.67%\(^3\) binary compatibility with version 1.5.

The branch has evolved to only contain changes which won’t compile with 1.4 and earlier compilers, rather than all of the 1.5 changes.

As a result, most of the missing material will first appear on the main branch and be merged across. A large part of this is support for the management extensions, which I’ve recently been working on as part of Google’s Summer of Code.

\(^3\)http://builder.classpath.org/japi/jdk15-generics.html as of 2006/08/15
Why did we need a branch in the first place?
But Why?

- Why did we need a branch in the first place?
- The new language constructs can not be handled by earlier compilers.
But Why?

- Why did we need a branch in the first place?
- The new language constructs can not be handled by earlier compilers.
- The way the changes have been implemented means that the biggest burden falls on the compiler.
But Why?

- Why did we need a branch in the first place?
- The new language constructs can not be handled by earlier compilers.
- The way the changes have been implemented means that the biggest burden falls on the compiler.
- In contrast, virtual machines can support the changes relatively easily.
But Why?

- Why did we need a branch in the first place?
- The new language constructs can not be handled by earlier compilers.
- The way the changes have been implemented means that the biggest burden falls on the compiler.
- In contrast, virtual machines can support the changes relatively easily.
- To an extent, a virtual machine can run 1.5 code without any changes if it ignores the version number.
An Epiphany for Free Compilers

The changes in 1.5 have resulted in significant changes in the landscape of Free Java compilers.
The changes in 1.5 have resulted in significant changes in the landscape of Free Java compilers.

Currently, the only Free compiler that can handle the generics branch is ecj, the compiler which forms part of the Eclipse project.
The changes in 1.5 have resulted in significant changes in the landscape of Free Java compilers.

Currently, the only Free compiler that can handle the generics branch is ecj, the compiler which forms part of the Eclipse project.

Fortunately, the most recent release provides the compiler in a separate JAR file so it can be used independently.
An Epiphany for Free Compilers

- The changes in 1.5 have resulted in significant changes in the landscape of Free Java compilers.
- Currently, the only Free compiler that can handle the generics branch is ecj, the compiler which forms part of the Eclipse project.
- Fortunately, the most recent release provides the compiler in a separate JAR file so it can be used independently.
- The easiest way to obtain a copy is to use the version provided in some GNU/Linux distributions (e.g. Fedora Core, Debian)
Compilers Fallen Along The Way

- IBM’s **Jikes** compiler can parse the syntax used by 1.5, but nothing else. Development seems to have ceased altogether.
Compilers Fallen Along The Way

- IBM’s **Jikes** compiler can parse the syntax used by 1.5, but nothing else. Development seems to have ceased altogether.
- **GCJ** is unable to handle 1.5 at present.
Compilers Fallen Along The Way

- IBM’s Jikes compiler can parse the syntax used by 1.5, but nothing else. Development seems to have ceased altogether.
- GCJ is unable to handle 1.5 at present.
- The first proposed solution for this was the development of a new compiler in C++ called gcjx. This was worked on by Tom Tromey.
IBM’s **Jikes** compiler can parse the syntax used by 1.5, but nothing else. Development seems to have ceased altogether.

**GCJ** is unable to handle 1.5 at present.

The first proposed solution for this was the development of a new compiler in C++ called **gcjx**. This was worked on by **Tom Tromey**.

This was declared dead on June the 6th, 2006 when permission was given by the steering committee for GCJ to utilise ecj to provide preliminary source code to bytecode compilation.
‘RMS approved the plan to use the Eclipse compiler as the new gcj front end. Work is being done on the gcj-eclipse branch; it can already build libgcj. This project will allow us to ship a 1.5 compiler in the relatively near future. The old gcjx branch and project is now dead.’
‘RMS approved the plan to use the Eclipse compiler as the new gcj front end. Work is being done on the gcj-eclipse branch; it can already build libgcj. This project will allow us to ship a 1.5 compiler in the relatively near future. The old gcjx branch and project is now dead.’

A copy of GNU Classpath 0.92’s generics branch has just been merged in to the gcj-eclipse branch earlier this week.
‘RMS approved the plan to use the Eclipse compiler as the new gcj front end. Work is being done on the gcj-eclipse branch; it can already build libgcj. This project will allow us to ship a 1.5 compiler in the relatively near future. The old gcjx branch and project is now dead.’

A copy of GNU Classpath 0.92’s generics branch has just been merged in to the gcj-eclipse branch earlier this week.

Version 4.3.0 is likely to be the first version of GCC to ship with a version of GCJ that supports Java 1.5.
As mentioned earlier, the VM developers have the easier job.
As mentioned earlier, the VM developers have the easier job.

The only bytecode change in 1.5 is the addition of class objects to the constant pool.
Virtual Machines

- As mentioned earlier, the VM developers have the easier job.
- The only bytecode change in 1.5 is the addition of class objects to the constant pool.
- Full support, however, requires supporting a mass of new VM interfaces in order to provide support for reflection.
As mentioned earlier, the VM developers have the easier job.

The only bytecode change in 1.5 is the addition of class objects to the constant pool.

Full support, however, requires supporting a mass of new VM interfaces in order to provide support for reflection.

Type variables, annotations and other such material are retained in the bytecode, but only as metadata.
As mentioned earlier, the VM developers have the easier job.

The only bytecode change in 1.5 is the addition of class objects to the constant pool.

Full support, however, requires supporting a mass of new VM interfaces in order to provide support for reflection.

Type variables, annotations and other such material are retained in the bytecode, but only as metadata.

Changes have been made to the reflection API in 1.5 to allow this information to be retrieved, so you can at least find the type parameters via reflection...
Making Generics Mainstream

Once a version of GCJ with 1.5 support is available (possibly as soon as it appears on the main development branch), GNU Classpath will move over to the generics branch as the mainstream development branch.
Once a version of GCJ with 1.5 support is available (possibly as soon as it appears on the main development branch), GNU Classpath will move over to the generics branch as the mainstream development branch.

Due to the separation of the library, the compiler and the virtual machine, such changes are contentious. However, we hope that, by now, most projects which rely on GNU Classpath will be able to support these changes.
Making Generics Mainstream

- Once a version of GCJ with 1.5 support is available (possibly as soon as it appears on the main development branch), GNU Classpath will move over to the generics branch as the mainstream development branch.

- Due to the separation of the library, the compiler and the virtual machine, such changes are contentious. However, we hope that, by now, most projects which rely on GNU Classpath will be able to support these changes.

- Many virtual machines can already run 1.5 code, and most have at least some of the required reflection support.
Making Generics Mainstream

- Once a version of GCJ with 1.5 support is available (possibly as soon as it appears on the main development branch), GNU Classpath will move over to the generics branch as the mainstream development branch.
- Due to the separation of the library, the compiler and the virtual machine, such changes are contentious. However, we hope that, by now, most projects which rely on GNU Classpath will be able to support these changes.
- Many virtual machines can already run 1.5 code, and most have at least some of the required reflection support.
- It’s been a long road, but it looks like we may finally be at the end...
Thanks for listening. Any questions?