eclipse

Tutorial: Eclipse APIs and Java 5

Boris Bokowski, John Arthorne, Jim des Rivières

IBM Rational Software, Ottawa Lab

Tutorial schedule (provisional)

08:00-08:15	Welcome + Introduction
08:15-09:00	Recap: API Design
09:00-09:15	Autoboxing, Variable Arity
09:15-09:30	Enumerations
09:30-09:45	Annotations
09:45-10:00	Covariant Return Types
10:00-10:30	Break
10:30-11:00	Generifying Classes and Interfaces
11:00-11:30	Generifying Fields and Methods
11:30-12:00	Evolving Generic Types and Methods
12:00-12:30	New in 3.3: API tools

APIs and Java 5

- This is not a tutorial on Java 5 language features
- This is tutorial on impact of Java 5 language features on API design
- Ref: Evolving Java-based APIs, rev 1.1
- http://wiki.eclipse.org/index.php/Evolving_Java-based_APIs

Language features added in Java 5

- Recap – new in JLS3
  - Autoboxing
  - Variable arity methods
  - Enumerations
  - Annotations
  - Covariant return types
  - Generic types
- Java Language Specification, Third edition (JLS3)
- Full text is available online
- http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

- JLS3 is the language spec underlying Java 5 (aka JDK 1.5)

Language Compatibility

- Language is highly compatible with previous versions of Java
- All programs that compiled under JLS2 also compile under JLS2 with the same meaning
  - Exception: “enum” is no longer allowed as identifier
- Some program texts that did not compile under JLS2 are legal under JL3
- Existing 1.4 class files will link and run as before with 1.5 class libraries

What would we like people to learn

- Appreciate the role of having strong API specifications
- View API from different perspectives
  - Specification
  - Implementer
  - Client
- Make people aware of the danger of overspecification
  - API is a cover story to prevent you from having to tell the truth
- Wiki hub for Eclipse API material

  http://wiki.eclipse.org/index.php/API_Central

Designing APIs == making laws

- Consider which side of road one drives on
- Think back to when there was no convention
- Slowdowns when oncoming carts meet
- Do I pass on (my) left or right?
- Individuals acting locally cannot improve things much
- Significant improvement requires convention
- Convention must be universally adopted to be effective
- Convention overrides desires of individuals
- Convention must choose left vs right
- Everyone passes on left would work fine
- Everyone passes on right would also work fine
- Convention must make arbitrary choice
- Once convention is in widespread use, passing speeds pick up
- Becomes downright dangerous to not follow convention
- Becomes important everyone knows about convention
- Becomes hard to rethink arbitrary choice once made

Recap: API Design

My eyes are dim I cannot see.
I have not got my specs with me.
I have not got my specs with me.
---*The Quartermaster's Song*

API specifications

- APIs are interfaces with specified and supported behavior

API specs

- API specs play many key roles

  A. Tell client what they need to know to use it
  B. Tell an implementor how to implement it
  C. Tell tester about key behaviors to test
  D. Determines blame in event of failure

Lessons learned

- API is not just public methods

No specs. No API.

References

- *Requirements for Writing Java API Specifications*
  http://java.sun.com/products/jdk/javadoc/writingapispecs/index.html
- *How to Write Doc Comments for the Javadoc Tool*
  http://java.sun.com/products/jdk/javadoc/writingdoccomments/index.html

Appropriate level of specification detail

- Is the specification too specific or detailed, making it difficult to evolve later on?
- Is the spec too vague, making it difficult for clients to know the correct usage?
- Is the API designed to be implemented or extended by clients?

API Contract language

- The language used in an API contract is very important
- Changing a single word can completely alter the meaning of an API
- It is important for APIs to use consistent terminology so clients learn what to expect

API Contract language

- RFC on specification language: http://www.ietf.org/rfc/rfc2119.txt
- **Must**, **must not, required, shall**: it is a programmer error for callers not to honor these conditions. If you don't follow them, you'll get a runtime exception (or worse)
- **Should**, **should not, recommended**: Implications of not following these conditions need to be specified, and clients need to understand the trade-offs from not following them
- **May, can**: A condition or behavior that is completely optional

API Contract language

Some Eclipse project conventions:

- **Not intended**: indicates that you won't be prohibited from doing something, but you do so at your own risk and without promise of compatibility. Example: "This class is not intended to be subclassed"
- **Fail, failure:** A condition where a method will throw a checked exception
- **Long-running**: A method that can take a long time, and should never be called in the UI thread
- **Internal use only:** An API that exists for a special caller. If you're not that special caller, don't touch it

Specs for Subclassers

- Subclasses may
  - "**implement**" - the abstract method declared on the subclass must be implemented by a concrete subclass
  - "**extend**" - the method declared on the subclass must invoke the method on the superclass (exactly once)
  - "**re-implement**" - the method declared on the subclass must not invoke the method on the superclass
  - "**override**" - the method declared on the subclass is free to invoke the method on the superclass as it sees fit
- Tell subclasses about relationships between methods so that they know what to override

Compatibility

It’s the same old story
Everywhere I go,
I get slandered,
Libeled,
I hear words I never heard
In the bible
And I’m one step ahead of the shoe shine
Two steps away from the county line
Just trying to keep my customers satisfied,
Satisfied.

---Simon & Garfunkel, *Keep the Customer Satisfied*

Compatibility

- **Contract** – Are existing contracts still tenable?
- **Binary** – Do existing binaries still run?
- **Source** – Does existing source code still compile?

Contract compatibility

Before:


```
/**
 * Returns the current display.
 * @return the display; never null
 */
public Display getDisplay();
```

After:

```
/**
 * Returns the current display, if any.
 * @return the display, or null if none
 */
public Display getDisplay();
```

- Not contract compatible for callers of getDisplay
- Contract compatible for getDisplay implementors

Contract compatibility

- Weaken method preconditions – expect less of callers
  - Compatible for callers; breaks implementors
- Strengthen method postconditions – promise more to callers
  - Compatible for callers; breaks implementors
- Strenghten method preconditions – expect more of callers
  - Breaks callers; compatible for implementors
- Weaken method postconditions – promise less to callers
  - Breaks callers; compatible for implementors

Binary compatibility lessons

- It is very difficult to determine if a change is binary compatible
- Binary compatibility and source compatibility can be very different
- You can't trust the compiler to flag non-binary compatible changes

- Reference: Gosling, Joy, Steele, and Bracha, *The Java Language Specification*, Third Edition, Addison-Wesley, 2005; chapter 13 Binary Compatibility
  http://java.sun.com/docs/books/jls/third_edition/html/binaryComp.html

- Reference: *Evolving Java-based APIs,* rev 1.1
  http://wiki.eclipse.org/index.php/Evolving_Java-based_APIs

Evolving APIs

- Techniques for evolving APIs
- Techniques for writing APIs that are evolvable

Techniques for enabling API evolution

- Use abstract classes instead of interfaces for non-trivial types if clients are allowed to implement/specialize
- Separate service provider interfaces from client interfaces
- Separate concerns for different service providers
- Hook methods
- Mechanisms for plugging in generic behavior (IAdaptable) or generic state, such as getProperty() and setProperty() methods

Autoboxing, Variable Arity

To avoid unexpected effects,
Feed your function the args it expects,
With an arity count
In the proper amount,
Or you'll find that your program objects.
--- mephistopheles, www.oedilf.com

If you have a procedure with 10 parameters,
you probably missed some.
Alan Perlis

Auto-boxing

- Integer bigX = (Integer) 5; // boxing conversion
  - Integer bigX = Integer.valueOf(5); // how it's compiled
- Integer bigX = 5; // auto-boxing
- int littleX = (int) bigX; // unboxing conversion
  - int littleX = bigX.intValue(); // how it's compiled
- int littleX = bigX; // auto-unboxing

Language feature has no real impact on API design or evolution

Variable arity methods

- void main(String… args) {…} // variable arity method
  - void main(String[] args) {…} // how it's compiled
- main("A", "B", "C“) // variable arity method invocation
  - main(new String[] { "A", "B", "C“ }) // how it's compiled

- Pros for use in APIs
  - More convenient invocations for clients
  - Works even better with auto-boxing
- Cons for use in APIs
  - Hidden garbage array objects
  - Even more hidden garbage with auto-boxing

Introducing variable arity methods

```
1. void main(T… args)   // variable arity method
2. void main(T[] args)  // fixed arity method
3. void main(T a0)      // fixed arity method
```

- Change T to T…
  - Breaks compatibility
- Change T[] to T…
  - Compatible
  - Compiler warnings if method is overridden/implemented

Evolving variable arity methods

1. void main(T… args) // variable arity method
2. void main(T[] args) // fixed arity method
3. void main(T a0) // fixed arity method

- Change T… to T
  - Breaks compatibility
- Change T… to T[]
  - Breaks compatibility
  - Binary compatible
  - Not source code compatible - invocations may no longer compile

Enumerations

Enums

- Enumeration types are a class type with self-typed constants

```
public enum Direction = {NORTH, EAST, SOUTH, WEST};
```

- Direction.NORTH is of type Direction
- Constants are canonical instance - can be compared with ==

- Pros for use in APIs
  - More strongly typed than ints
- Cons for use in APIs
  - Less flexible than ints

Evolving enums

- Enum constant names are significant at runtime
  - Direction.NORTH.name() returns “NORTH“
  - Direction.valueOf(“NORTH“) returns Direction.NORTH
- Order of enum constants is significant at runtime
  - Direction.values() returns new Direction[] { Direction.NORTH, Direction.EAST, Direction.SOUTH, Direction.WEST) };
- Rename enum constant
  - Breaks binary compatibility
- Delete enum constant
  - Breaks binary compatibility
- Reorder enum constants
  - Liable to break contact compatibility
- Add enum constant
  - Liable to break contact compatibility

Annotations

Annotations

- Annotation types are special form of interface
  - Methods are called elements

```
/* © Copyright 2007 IBM Corp.  All rights reserved.  This source code is made available under the terms of the Eclipse Public License, v1.0. */
public @interface  LongOp {}                // marker annotation type
public @interface ServiceType {             // simple annotation type
   enum Style { REST, RPC }
   Style value() default Style.REST;
}
public @interface Login {                   // annotation type
   String firstName();
   String lastName();
}

@ServiceType(ServiceType.Style.RPC)    // annotation
public interface MyShop {
   @LongOp                                  // annotation
   @Login(firstName="Jayne", firstName="Daoust")  // annotation
   public void open();

   @LongOp                                  // annotation
   public void close();
}
```


Annotations

- Impact on API design ??
- Use annotations to systematize and encode information about API

- Annotations are readable by
  - Tools that analyze source code
    - annotation.RetentionPolicy.SOURCE
  - Tools that analyze class files
    - annotation.RetentionPolicy.CLASS
  - Program itself using reflection
    - annotation.RetentionPolicy.RUNTIME

Evolving annotation types

- Annotation types - follow general guidelines for non-implementable interfaces
- Add annotation type element
  - If element specifies default value
    - Compatible
  - If element does not specify default value
    - Breaks compatibility
- Delete annotation type element
  - Breaks compatibility
- Rename annotation type element
  - Breaks compatibility
- Change type of annotation type element
  - Breaks compatibility
- Add default class for annotation type element
  - Compatible
- Change default clause for annotation type element
  - Compatible
- Delete default clause for annotation type element
  - Breaks compatibility

Evolving annotations

- Adding or removing annotations has no effect on the correct linkage of class files by the Java virtual machine
- But…
- Annotations exist to be read via reflective APIs for manipulating annotations
  - No uniform answer as to what will happen if a given annotation is or is not present on an API element (or non-API element, for that matter)
  - Depends entirely on the specifics of the annotation and the mechanisms that are processing those annotations
- Parties that declare annotation types should try to provide helpful guidance for their customers

Generic Types

I like my lyrics to feel conversational and
truthful, as if we're having real talk.
I don't really like generic lyrics.
---Meredith Brooks

Generic Types

- **Generic types** are classes or interfaces with *type variables*


```
public class Stack<E> {                    // generic type
   public void push(E element);
   public E pop();
}
```

- **Parameterized types** supply actual type arguments
  - Reference types only – no primitive types


```
Stack<String> stringStack                  // parameterized type
   = new Stack<String>();
Stack<Date> integerStack                   // parameterized type
   = new Stack<Date>();

stringStack.push("A");
String s1 = stringStack.pop();
String s1 = (String) stringStack.pop();    // how it's compiled

stringStack.push(new Date());              // compile error
Integer i1 = stringStack.pop();            // compile error
```


Type Bounds

- Type variables may have bounds


```
public class NumberStack<E extends Number> {
   public void push(E element);
   public E pop();
}

NumberStack<Integer> integerStack
   = new NumberStack<Integer>();

NumberStack<Float> floatStack
   = new NumberStack<Float>();

NumberStack<String> stringStack
   = new NumberStack<String>();                    // compile error
```


Wildcard types

- Consider


```
interface Collection <E> {
   boolean containsAll(Collection<✠✠✠✠> c);
   ...
}

Collection<Number> myCollection;
Collection<Integer> yourCollection;
myCollection.containsAll(yourCollection);

Compare:
    boolean containsAll(Collection c);                  // raw type
    boolean containsAll(Collection<Object> c);          // too restrictive
    boolean containsAll(Collection<E> c);               // too restrictive
    boolean containsAll(Collection<?> c);               // just right
```


Generic Types

- Pros for use in APIs
  - Permits strong typing in certain situations that would otherwise be loosely typed
    - More errors detected at compile-time type
    - More convenient for callers
    - More convenient for implementers
  - Dovetail with Java Collections API
- Cons for use in APIs
  - None if done well
- Neither Pro Nor Con
  - Performance

Evolving Generified API

Add type parameter	**Breaks compatibility** (unless type was not generic)
Delete type parameter	**Breaks compatibility**
Re-order type parameters	**Breaks compatibility**
Rename type parameters	**Binary compatible**
Add, delete, or change type bounds of type parameters	**Breaks compatibility**

- We strongly recommend you get it right the first time
- As often the case with API design, there is no second chance

Evolving APIs that use Generic Types

- Same rules as before:
  - Changing an argument type is like removing a method and adding a new one
  - Same for return types

Generification

- Introducing generic types into an existing API
- Possible to preserve compatibility
  - E.g., Java Collections API was generified in 1.5
- Language has special provisions for backwards compatibility
- Raw type – using generic type as if it were not generic
  - List beatles = Arrays.asList("John", "Paul", "George", "Ringo"); // raw
- Raw types are discouraged – compiler warnings by default
- Compatibility between old and new is based on **erasures**

Erasures

- The compiler replaces type variables so that all parameterized types share the same class or interface at runtime


```
public class Stack<E> {
   public void push(E Object element);
   public E Object pop();
}

Stack<String> stringStack;
Stack<Integer> integerStack;
```


Converting Raw to Parameterized Types

- Applies if
  - Raw types (e.g. Collections) appear in your API
  - Conversion is contract-compatible
- Return types: making stronger promises, always possible

  `public Map getArgs() -> public Map<String, String> getArgs()`
- Argument types: enforcing existing contracts at compile time

  `public void setArgs(Map m) -> public setArgs(Map<String, String> m)`
  - This is a binary compatible change (erasure is the same), BUT...
  - Map<String, String> is not equivalent to “Map with String keys and values”
  - Sometimes not easy to step up to stronger contract
  - For example, it is easy if they create the map themselves, but hard to do if they get it from somewhere else
  - Be careful not to require too much from your clients

Introducing Type Variables

- Applies if
  - Your API is like the collection framework (e.g. container types), or
  - You inherit from / delegate to a type that was generified, or
  - java.lang.Object appears in your API but clients need to downcast

- Return types: Relieving clients from having to downcast

```
interface IObservableValue { Object getValue(); }
    -> interface IObservableValue<V> { V getValue(); }
```

- Argument types: Enforcing contracts at compile time

```
interface IObservableValue { void setValue(Object value); }
    -> interface IObservableValue<V> { void setValue(V value); }
```

- Don't overdo it, generify cautiously
- Weigh type safety against complexity
- Be aware of ripple effect
- Problematic: Arrays, Fields

Arrays and Generic Types Are Different

- String[] is a subtype of Object[], but
  ArrayList<String> is not a subtype of ArrayList<Object>!

- Reason for this: Principle of substitutability
  A is a subtype of B if B can be substituted whenever an A is expected

- Consider:


```
public static void someMethod(List<Object> someList) {
  someList.add(new Object());
}
List<String> stringList = new ArrayList<String>();
someMethod(stringList);
```

type error

- Array types: String[] is a subtype of Object[], but you will get an ArrayStoreException if you try to store an Object in an array that was created as a String array

Array Types in API and Generification

- class ArrayList<E> {

```
  ...
    E[] toArray() {
      // how to implement this?
    }
    E[] toArray(E[] es) {
      // here you can use:
      Array.newInstance(es.getClass().getComponentType(), size());
    }
}
```

- Solution:
  - If arrays are pervasive in your API (as in Eclipse):
    Do not generify types that appear as array component types in your API
  - Otherwise, generify everything except problematic cases like the one above

Generic Methods

- This sort API is not very useful to clients:
  public static void sort(List<Object> list);
  Why? Because e.g. List<String> is not a subtype of List<Object>, clients would be overly constrained.

- Generic methods to the rescue:
  class SortUtil {
    public <E> void sort(List<E> list) { ... }
  }

- Can be invoked as follows:
  List<String> stringList = ...;
  SortUtil.<String>sort(stringList); *(oftentimes, type parameter can be omitted)*

- If the concrete type of E is not used in the body of sort(), you can write:
  public static void sort(List<?> list) { ... }

Generic Methods and Type Bounds

- Generic method with type constraint:
  ```
  class SortUtil {
    public <E extends Comparable> void sort(List<E> list) { ... }
  }
  ```
- If E is not important in the body of sort:
  ```
  class SortUtil {
    public void sort(List<? extends Comparable> list) { ... }
  }
  ```
  (remember that using List<Comparable> would be very restrictive for clients)

- However, consider this:
  ```
  class SortUtil {
    public <E> void sort(List<E> list, Comparator<E> comparator) { ... }
  }
  ```
- Requiring a Comparator<E> is restrictive, you should instead do this:
  ```
    public <E> void sort(List<E> list, Comparator<? super E> comparator)
  { ... }
  ```


Hidden casts

- **Not recommended:**

```
class Wrapper<T> {
   protected T wrapped;
}
class FileWrapper
            extends Wrapper<File> {

  public void mkdirs() {
    if (wrapped != null)
      wrapped.mkdirs();
   }
  public void createNewFile() {
    if (wrapped != null)
      wrapped.createNewFile();
   }

}
```

- **Will be translated to:**


```
class Wrapper {
   protected Object wrapped;
}
class FileWrapper
            extends Wrapper {

  public void mkdirs() {
    if (wrapped != null)
      ((File)wrapped).mkdirs();
   }
  public void createNewFile() {
    if (wrapped != null)
      ((File)wrapped).createNewFile();
   }

}
```


More Resources about Java 5 and APIs

- EMF long talks on Tuesday and Wednesday

API Tools

- Work in PDE Incubator to provide API tools
- Four general categories of tooling:
  - API Comparison
  - Bundle version checking
  - Usage discovery
  - Usage validation

API Comparison

- Create XML-based snapshot of the API of a given bundle or project
- Produce a report on API changes between two snapshots
- Identifies potentially breaking changes (not perfect, there are various corner cases)
- Uses API difference analysis to suggest appropriate version number changes

Uses for API Comparison

- Catch breaking API changes early
- Helps in writing migration documentation for clients in cases where breaking changes are necessary
- Useful as input for New & Noteworthy, API documentation

More information on API tools

- In CVS at dev.eclipse.org/cvsroot/eclipse/pde-incubator/api-tooling/
- http://wiki.eclipse.org/index.php/PDE_UI_Incubator_ApiTools

Autobox/Arity Quiz 1: Is this compatible?

Before:


```
public class A {
   public void foo(String... x) {
   }
}
```

After:

```
public class A {
   public void foo(String... x) {
   }
   public void foo(String x, String... y) {
   }
}
```


Autobox/Arity Quiz 2: What does it print?


```
public class Sum {
   public int length(int... x) {
          return Arrays.asList(x).size();
   }

   public int length(String... x) {
          return Arrays.asList(x).size();
   }

   public static void main(String[] arguments) {
          System.out.print(new Sum().length(1, 2, 3, 4));
          System.out.print(new Sum().length("1", "2", "3", "4"));
   }
}
```

A) 11
B) 44
C) 14
D) ClassCastException

Generics Quiz 1: Is This Compatible?

Before:


```
public class A {
   public void foo(Collection c) {…}
}
```

After:

```
public class A<T> {
   public void foo(Collection<T> c) {…}
}
```


Generics Quiz 2: Is This Compatible?

Before:


```
public class A {
   public void foo(Collection<String> c) {…}
}
```

After:

```
public class A {
   public void foo(Collection c) {…}
}
```


Generics Quiz 3: Is This Compatible?

Before:


```
public class A {
   public final void foo(Collection<String> c) {…}
}
```

After:

```
public class A<T> {
   public final void foo(Collection<Object> c) {…}
}
```


Generics Quiz 4: Is This Compatible?

Before:


```
public class A<T> {
   public void foo(Collection<T> c) {…}
}
```

After:

```
public class A<T,E> {
   public void foo(Collection<T> c) {…}
}
```


Generics Quiz 5: Is This Compatible?

Before:


```
public class A {
   public void foo(Collection<Number> c) {…}
}
```

After:

```
public class A<T extends Number> {
   public void foo(Collection<T> c) {…}
}
```


Generics Quiz 6: What does this print?


```
static class A<T extends A<T>> {
   public T ping() {
          return (T) this;
   }
}
static class B extends A<B> {
   public B pong() {
          return this;
   }
}
public static void main(String... args) {
   System.out.println(new
   B().ping().pong().getClass().getSimpleName());
}
```

A) A
B) B
C) Compile error
D) ClassCastException

Generics Quiz 7: What does this print?

A) null
B) java.lang.Object@1a2b4c1d
C) Compile error
D) ClassCastException


```
public class Doit {
   public static class A<T> {
          T val = (T)new Object();
   }
   public static void main(String... arguments) {
          A<String> a = new A<String>();
           if (a.val != null)
                  System.out.println(a.val);
   }
}
```


Legal Notices

- IBM and Rational are registered trademarks of International Business Corp. in the United States and other countries
- Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both
- Other company, product, or service names may be trademarks or service marks of others

END

- Questions or Comments?

BACKUP SLIDES

Compatibility Quiz Material

- Collection -> Collection<E>
- Map<K> -> Map<K,V>
- Collection<E> -> Collection
- foo(List<String> list) -> foo(List<Object> list)
- void containsAll(Collection<E> c) -> void containsAll(Collection<?> c)
  Compatible for callers
- void containsAll(Collection<Object> c)
  -> void containsAll(Collection<?> c)
  Compatible for callers
- double sum(Collection<Integer> c)
  -> double sum(Collection<? extends Number> c)
  compatible for callers

