eclipseCON 2006

Tutorial: Designing Eclipse APIs

Boris Bokowski, John Arthorne, Jim des Rivières

IBM Rational Software
Ottawa, Canada

 March 21, 2006

Tutorial schedule (provisional)

8:30-8:45	Welcome + Setup
8:45-9:15	Stack example (2 exercises)
9:15-9:45	API critique (1 critique of turtle)
9:45-10:15	More advanced API design (1+ exercise)
10:15-10:30	Break
10:30-11	API compatibility quiz
11-11:30	API evolution (1 exercise using turtle)
11:30-12	API process (no exercises)

Goals

- Learn about and practice API design
- Build API design community
- Give API designers an opportunity to meet and interact

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
- Encourage use of a Wiki hub for Eclipse API material

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

API specifications

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

Exercise: First baby specs

- **Write Javadoc specs for this API class**

```
package org.eclipsecon.stackexample.myspecs;
public final class Stack {
   public Stack();
   public void push(Object v);
   public Object pop();
   public Object peek();
   public boolean isEmpty();
   public int search(Object o);
}
```


Question: What do we spec for constructor?

```
/**
 *        ???
 */
public Stack();
```


Question: What do we spec for constructor?

```
/**
  * Creates a new empty stack.
  */
public Stack();
```

- Initial conditions

Question: What do we spec for methods?

```
/**
 *
 *              ???
 *
 *
 */
public void push(Object o);
```


Question: What do we spec for methods?

```
/**
 * Adds the given object to the top of this stack.
 *
 * @param o
 *            the object to push; may be <code>null</code>
 */
public void push(Object o);
```

- Method specs include
  - Purpose
  - Parameters
  - Postconditions

Question: What do we spec for methods?

```
/**
 *
 *          ???
 *
 *
 *
 */
public Object pop();
```


Question: What do we spec for methods?

```
/**
 * Removes and returns the object on the top of this stack.
 * The stack must not be empty.
 *
 * @return the object popped off the stack; may
 *                be <code>null</code>
 */
public Object pop();
```

- Method specs include
  - Preconditions
  - Results

Question: What do we spec for methods?

```
/**
 *
 *
 *              ???
 *
 *
 *
 *
 *
 *
 *
 */
public int search(Object o);
```


Question: What do we spec for methods?

```
/**
 * Returns the position of the given object on this stack. The position is
 * the distance from the top of the stack or, equivalently, the number of
 * calls to {@link #pop()} required to uncover the object. If the same
 * object occurs more than once, the position returned is that of the one
 * closest to the top. Returns -1 if the object is not present. The method
 * uses {@link #equals(Object)} to compare objects.
 *
 * @return the distance of the given object from the top of the stack, or -1
 *         if the object is not present
 */
public int search(Object o);
```

- Method specs include
  - Other important details

Question: What do we spec for class?

```
/**
 *
 *            ???
 *
 *
 */
public final class Stack {
```


Question: What do we spec for class?

```
/**
 * Represents a stack (last-in-first-out).
 * <p>
 * This class is not thread-safe.
 * </p>
 */
public final class Stack {
```

- Class specs include
  - Purpose
  - General usage

Exercise: The Impy and Testy show

- Form 2-person teams: dubbed “Impy” and “Testy”
- Impy will implement Stack
  - Objective: Good implementation conforming to spec
  - Zero defects
- Testy will implement tests for Stack
  - Objective: Check that Stack implementation conforms to spec
  - Detect as many non-conforming implementation as possible
  - Pass any conforming implementation
- Starting with same spec and working independently
  - No communication between Impy and Testy at this stage

Exercise

- Impy
  - Edit this file
    - package org.eclipsecon.stackexample.myimpl;
    - Class Stack
  - Wait for Testy to finish (no reading ahead ☺)
- Testy
  - Edit this file
    - package org.eclipsecon.stackexample.mytests;
    - Class StackTests
  - Wait for Impy to finish

Exercise

- Impy and Testy work together now

- Put Impy's Stack implementation and Testy's StackTest together
- Run StackTest
- Fix any failures

Exercise

- Doublecheck implementation by running reference tests
  - package org.eclipsecon.stackexample.reftests;
  - Run RefTestMyImpl
- Doublecheck tests against reference impl
  - In StackTests
  - Import org.eclipsecon.stackexample.refimpl.Stack;
  - Run StackTests

eclipseCON 2006

Discussion

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

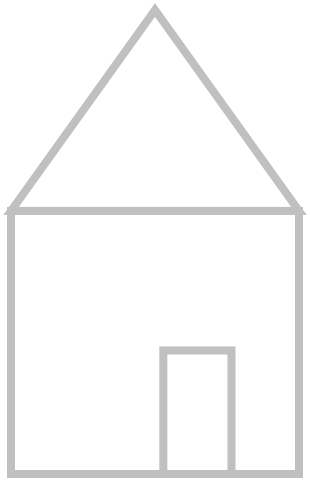
API critique: What to look for when reviewing an API.

- Introductory sentence summarizing purpose
- Pre-conditions
- Post-conditions
- Capturing of argument and result objects
- Specifying failure
- Side effects
- Concurrency
- Event ordering
- Callbacks

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?

Exercise: You be the judge



- Review and critique the turtle API
- API is in org.eclipsecon.turtle, and org.eclipsecon.turtle.swt
- To help uncover API problems, write a simple program that uses the turtle API to draw a house
- Use org.eclipsecon.turtle.examples.RandomExample to see how to set up your program

Sketch map of API

```
package org.eclipsecon.turtle;
public interface ITurtle {
    public void go(double amount);
    public void penDown();
    public void penUp();
    public void reset();
    public void turn(double amount);
public interface ITurtleRunnable {
    public void run(ITurtle t);
public interface ITurtleRunner {
    public void execute(TurtleState initialState, ITurtleRunnable turtleRunnable);
public class TurtleFactory {
    public static ITurtle createTurtle(TurtleState state, TurtlePen pen);
public class TurtlePen {
    public void drawLine(double x1, double y1, double x2, double y2);
public class TurtleState {
    public TurtleState();
    public TurtleState(double x, double y, double direction, boolean isDown);
    public double getDirection();
    public double getX();
    public double getY();
    public boolean isPenDown();
package org.eclipsecon.turtle.swt;
public class SWTTurtleRunner implements ITurtleRunner {
    public static SWTTurtlePen createSWTTurtlePen(GC gc);
    public void execute(TurtleState initialState, ITurtleRunnable turtleRunnable);
package org.eclipsecon.turtle.printing;
public class PrintingTurtleRunner implements ITurtleRunner {
    public void execute(TurtleState initialState, ITurtleRunnable turtleRunnable);
```


API Review: ITurtle

- Should perhaps be a class to allow specialization
- Unit for go()? Specified, or left open?
- Unit for turn()? Needs to be specified
- Negative values for go(), negative angles for turn()
- What happens when penDown is called and the pen is already down? Same for penUp()
- Reset() is not very useful without being able to find out the initial or current state
- Are there any boundaries to the area in which the turtle can go? What if we go(Double.MAX_VALUE) several times?

API Review: ITurtleRunnable

- Can clients implement this interface?
- Difficult to implement without knowing initial state of the turtle
- Any restrictions on the turtle state after the runnable?
- Is the implementor allowed to continue manipulating the turtle after the run method has returned?

API Review: ITurtleRunner

- Can clients implement this interface?
- When will the runnable be executed, and in what thread?
- The specification falls into the trap of describing the implementation, rather than stating what the caller needs to know. Does the client care that it will create a turtle before calling the runnable? Maybe some implementations will want to reuse turtle instances
- Can execute be called multiple times on a single instance?

API Review: TurtleFactory

- Shouldn‘t allow instantiation of the factory – should make the default constructor private
- It doesn‘t need to say the turtle will be new – this prevents future caching and reuse of turtle instances

API Review: TurtlePen

- Who is this API directed at? Can clients subclass it?
- Constructor should be specified explicitly
- Behaviour of the default implementation of drawLine should be specified
- Is drawLine intended to be overwritten or extended? I.e., must subclasses call super.drawLine()?

API Review: TurtleState

- Who is this API directed at? Can clients subclass it?
- State for default constructor is not specified
- Should specify whether a TurtleState is immutable (current implementation is immutable, which can be a very useful property)
- Is the state linked to a particular turtle? Will it change when the turtle changes?
- Default constructor could be replaced by a singleton instance, since all default locations are the same

API Review: SWTTurtleRunner

- Spec what this implementation of execute does:
  - Must run in SWT's UI thread
  - Can only be called once because it creates a display

API Review: SWTTurtlePen

- Trick question – this isn‘t API

API problems that make life difficult for clients

- All interfaces, but no constructor
- Spec too vague
- Check assumptions about defaults
- Internationalization: specify if strings are visible to the end-user
- Non-functional requirements
  - thread safety
  - progress reporting
  - being able to cancel long-running operations
  - nesting, composeability
- Completeness (e.g., add but no remove)
- Names (avoid overly long names for elements that are used a lot)

API problems that make life difficult for implementations

Making promises that are hard to keep:

- Real-time promises
- Promises about the order of operations - prevents future concurrency
- Over-specifying precision of results (returns the number of nanoseconds since the file was changed)
- Returning a data structure that promises to remain up to date indefinitely (leaking live objects)

API problems that make life difficult for implementations

- Exposing unnecessary implementation details
- Factory methods: specifying that it "returns a new instance" rather than "returns an instance" - prevents future caching/pooling
- Specifying the whole truth, rather than just the truth
- Specifying all failure cases (instead of "reasons for failure include...")
- Specifying that an enum or similar pool of types is a closed set - prevents adding entries later (eg: resource types)
- Specifying precise data structures of return types (HashSet, rather than just Set or Collection)

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

Advanced Topics –subclassing, listeners

There Are TwoTurtle APIs

- The Turtle API serves two customers
  - clients: Implement ITurtleRunnable, use ITurtle
  - turtle providers: Implement ITurtleRunner, subclass TurtlePen
- These two aspects are not clearly separated
  - May lead to confusion (mostly for turtle clients)
  - What could we do to improve this?

Exercise: Refactoring the Turtle API

Goal

- Make suggestions for separating client API from turtle provider API

Time

- 3 minutes

Suggested steps

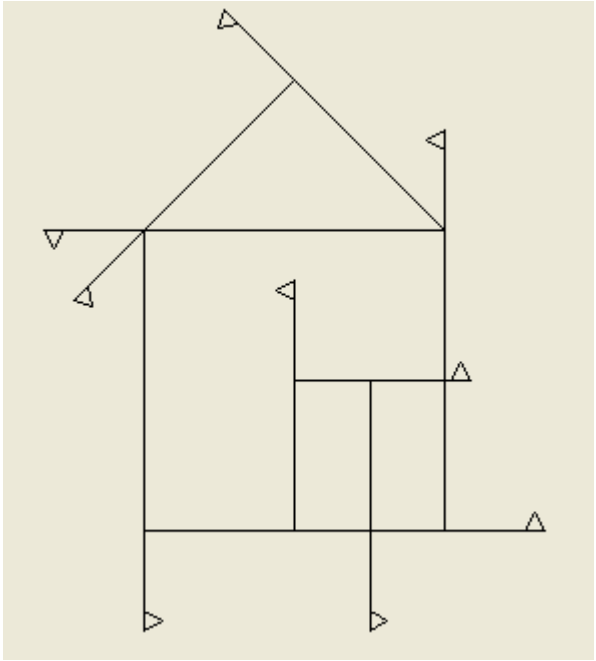
- For each class/interface:
  - who calls the methods
  - who (if any) will extend or implement it
- Suggest refactorings that improve the separation, e.g.
  - different packages?
  - protected members?

Sketch of API changes

```
package org.eclipsecon.turtle;
public interface ITurtle {
    public void go(double amount);
    public void penDown();
    public void penUp();
    public void reset();
    public void turn(double amount);
public interface ITurtleRunnable {
    public void run(ITurtle t);
public interface ITurtleRunner {   /** not intended to be implemented by clients, subclass TurtleRunner */
    public void execute(TurtleState initialState, ITurtleRunnable turtleRunnable);
public class TurtleFactory {
    public static ITurtle createTurtle(TurtleState state, TurtlePen pen);
public class TurtlePen {
    public void drawLine(double x1, double y1, double x2, double y2);
public class TurtleState {
    public TurtleState();
    public TurtleState(double x, double y, double direction, boolean isDown);
    public double getDirection();
    public double getX();
    public double getY();
    public boolean isPenDown();
package org.eclipsecon.turtle.provider;
public class TurtleRunner implements ITurtleRunner {
    protected static class TurtlePen {
        public void drawLine(double x1, double y1, double x2, double y2);
    }
    protected final ITurtle createTurtle (TurtleState initialState, TurtlePen pen);
    public void execute(TurtleState initialState, ITurtleRunnable turtleRunnable);
package org.eclipsecon.turtle.swt;
public class SWTTurtleRunner extends TurtleRunner {
    public SWTTurtleRunner();
```


Exercise: Listening to the turtle

- **Write Javadoc specs for a turtle listener**
  - should be notified about (at least)
    - go
    - penUp/penDown
    - turn
    - reset
  - Example
    - draw a flag on each turn
  - OK to change the API and break turtle providers, but existing runnables should stay unchanged
  - Consider adding listeners to ITurtleRunner, not ITurtle



Sketch of API changes

```
ITurtleRunner
  public void addTurtleListener(ITurtleListener l);
  public void removeTurtleListener(ITurtleListener l);

package org.eclispsecon.turtle.listener;
public interface ITurtleListener {
  public void handleTurtleEvent(TurtleEvent e);
}
public final class TurtleEvent extends EventObject {
  public static final int TURTLE_BEGIN = 1;
  public static final int TURTLE_END = 2;
  public static final int TURTLE_GO = 3;
  public static final int TURTLE_PEN_CHANGE = 4;
  public static final int TURTLE_RESET = 5;
  public static final int TURTLE_TURN = 6;
  public TurtleEvent(ITurtle source, int eventType, Object arg);
  public ITurtle getTurtle();
  public int getEventType();
  public Object getArgument();
}
```


Question: What do we spec for addListener?

```
/**
 *
 *
 *
 *
 *
 *
 *
 *
 * @param listener
 */
public void addTurtleListener(ITurtleListener listener);
```


Question: What do we spec for addListener?

```
/**
 *
 *
 *
 *
 *
 *
 *
 *
 * @param listener
 */
public void addTurtleListener(ITurtleListener listener);
```

- Can a listener be registered more than once?
- May listeners be added during event notification? Will they be notified of the current event?

Question: What do we spec for addListener?

```
/**
 * Adds the given listener to the list of turtle listeners. This
 * method has no effect if the listener is already registered.
 * <p>
 * Listeners added during event notification will only be notified of
 * the next event.
 * </p>
 *
 * @param listener
 */
public void addTurtleListener(ITurtleListener listener);
```

- Can a listener be registered more than once?
- May listeners be added during event notification? Will they be notified of the current event?

Question: What do we spec for removeListener?

```
/**
 *
 *
 *
 *
 *
 *
 *
 *
 * @param listener
 */
public void removeTurtleListener(ITurtleListener listener);
```


Question: What do we spec for removeListener?

```
/**
 *
 *
 *
 *
 *
 *
 *
 *
 * @param listener
 */
public void removeTurtleListener(ITurtleListener listener);
```

- Can a listener be removed if it has not been added before?
- May listeners be removed during event notification? Will they be notified of the current event?

Question: What do we spec for removeListener?

```
/**
 * Removes the given listener from the list of turtle listeners. This
 * method has no effect if the listener was not registered.
 * <p>
 * Listeners removed during event notification will still be notified of
 * the current event.
 * </p>
 *
 * @param listener
 */
public void removeTurtleListener(ITurtleListener listener);
```

- Can a listener be removed if it has not been added before?
- May listeners be removed during event notification? Will they be notified of the current event?

Question: What do we spec for execute (clients)?

```
/**
 *
 *
 *
 *
 *
 * @param initialState   The starting state of the turtle
 * @param turtleRunnable   The turtle program to run
 */
public void execute(TurtleState initialState, ITurtleRunnable
   turtleRunnable);
```


Question: What do we spec for execute (clients)?

```
/**
 *
 *
 *
 *
 *
 * @param initialState   The starting state of the turtle
 * @param turtleRunnable   The turtle program to run
 */
public void execute(TurtleState initialState, ITurtleRunnable
   turtleRunnable);
```

- Is a different thread used for notification?
- May listeners call back into ITurtleRunner or ITurtle?

Question: What do we spec for execute (clients)?

```
/**
 * Creates a turtle with the given initial state, and runs the given
 * turtle program with that turtle. Listeners will be notified of the
 * beginning and end of the execution, and of changes to the turtle
 * as they occur. Listeners are notified synchronously.
 *
 * @param initialState   The starting state of the turtle
 * @param turtleRunnable   The turtle program to run
 */
public void execute(TurtleState initialState, ITurtleRunnable
   turtleRunnable);
```

- Is a different thread used for notification?
- May listeners call back into ITurtleRunner or ITurtle?

Provider spec: TurtleRunner.execute(…)

```
/**
 * {@inheritDoc}
 * <p>
 * Implementers must call
 * {@link #executeAndNotify(TurtleState,
   org.eclipsecon.turtlelistener.provider.TurtleRunner.TurtlePen,
   ITurtleRunnable)}
 * to run the given runnable, they should not call the given turtle
 * runnable's run method directly.
 * </p>
 */
public void execute(TurtleState initialState, ITurtleRunnable
   turtleRunnable);
```


Provider spec: TurtleRunner.executeAndNotify(…)

```
/**
 * Runs the given turtle runnable using the given turtle, firing off
 * events to turtle listeners as they occur.
 *
 * @param turtle
 *            The turtle to use
 * @param turtleRunnable
 *            The turtle program to run
 */
protected final void executeAndNotify(TurtleState initialState,
        TurtlePen turtlePen, ITurtleRunnable turtleRunnable);
```


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

Question: What do we spec for ITurtleListener?

```
/**
 *
 *
 *
 */
public interface ITurtleListener {

   public void handleTurtleEvent(…);

}
```


Question: What do we spec for ITurtleListener?

```
/**
 * A turtle listener is notified of changes to turtles in the context of
 * an ITurtleRunner. This interface is intended to be implemented
 * by clients.
 */
public interface ITurtleListener {

   public void handleTurtleEvent(…);

}
```


Question: What do we spec for handleTurtleEvent?

```
/**
 *
 *
 *
 *
 *
 *
 *
 * @param turtleEvent
 */
public void handleTurtleEvent(TurtleEvent turtleEvent);
```


Question: What do we spec for handleTurtleEvent?

```
/**
 *
 *
 *
 *
 *
 *
 *
 *
 * @param turtleEvent
 */
public void handleTurtleEvent(TurtleEvent turtleEvent);
```

- Where and how may arguments be used?
- Event objects are extensible, explicit passing of values is not
- What happens if listeners call back into ITurtleRunner or ITurtle?

Question: What do we spec for handleTurtleEvent?

```
/**
 * Handle the given turtle event. The event object and the objects it
 * references may only be used until this method returns.
 * <p>
 * Note that calling methods on the turtle that is the source of this
 * event will cause recursive listener notification.
 * </p>
 *
 * @param turtleEvent
 */
public void handleTurtleEvent(TurtleEvent turtleEvent);
```

- Where and how may arguments be used?
- Event objects are extensible, explicit passing of values is not
- What happens if listeners call back into ITurtleRunner or ITurtle?

Question: What do we spec for TurtleEvent (1)?

Question: What do we spec for TurtleEvent (1)?

```
/**
 * Event object describing an event that happened to a turtle.
 */
public final class TurtleEvent extends EventObject {
   /** @return the turtle */
   public ITurtle getTurtle() { … }

   /** @return the eventType, one of the event type constants */
   public int getEventType() { … }

   /** @return the argument */
   public Object getArgument() { … }
```


Question: What do we spec for TurtleEvent (2)?

```
/** Event type constant describing that a turtle is about to be
 *  used. The argument object holds the initial state of the turtle of
 *  type TurtleState. */
public final static int TURTLE_BEGIN = 1;

/** Event type constant describing that a turtle is no longer used.
 * The value of the argument object is undefined. */
public final static int TURTLE_END = 2;

/** Event type constant describing that a turtle performed a go().
 *  The argument object holds the argument to go() as a Double. */
public final static int TURTLE_GO = 3;
```

- Has the change already happened when the listener is notified, or is it about to happen? (Chance of vetoing)

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 quiz

- Is the code snippet a binary compatible change?
- Is it source compatible?

Binary compatibility #1

Before:

```
public class Test {
   public void foo() {
      System.out.print(“Yes”);
   }
}
```

After:

```
public class Test {
   public void foo() {
      System.out.print(“Oui”);
   }
}
```

- Binary compatible
- Method bodies do not affect binary compatibility

Binary compatibility #2

Before:

```
public class Test {
    public void foo() {}
    public void bar() {}
}
```

After:

```
public class Test {
    public void foo() {}
}
```

- Not binary compatible
- Not source compatible
- Deleting methods is a breaking change

Binary compatibility #3

Before:

```
public class Test {
    public void foo() {}
}
```

After:

```
public class Test {
    public void foo(int flags) {}
```

- Not binary compatible
- Parameters are part of the method signature

Binary compatibility #4

Before:

```
public class Test {
    public void foo(String s) {}
}
```

After:

```
public class Test {
    public void foo(Object o) {}
}
```

- Not binary compatible
- Source compatible

Binary compatibility #5

Before:

```
public class Test {
    public void foo(Object o) {}
}
```

After:

```
public class Test {
    public void foo(Object o) {}
    public void foo(String s) {}
}
```

- Binary compatible
- When source references are recompiled they may be bound to the new method
- Will cause errors in source references with a null argument, such as test.foo(null)

Binary compatibility #6

Before:

```
public class Super {
    public void foo(String s) {}
}

public class Sub extends Super {
    public void foo(String s) {}
}
```

After:

```
public class Super {
    public void foo(String s) {}
}

public class Sub extends Super {
}
```

- Binary compatible
- A different method will be called at runtime when method “void foo(String) is invoked on an object of type “Sub”

Binary compatibility #7

Before:

```
public class Test {
    public static final int x = 5;
}
```

After:

```
public class Test {
    public static final int x = 6;
}
```

- Not binary compatible
- Constant values that can be computed by the compiler are in-lined at compile time. Referring code that is not recompiled will still have the value “5” in-lined in their code

Binary compatibility #8

Before:

```
public class Test {
    public static final
      String s = “foo”.toString();
}
```

After:

```
public class Test {
    public static final
      String s = “bar”. toString();
```

- Binary compatible
- Constant value cannot be computed at compile-time

Binary compatibility #9

Before:

```
package org.eclipse.internal.p1;
public class Super {
   protected void foo(String s) {}
}

package org.eclipse.p1;
public class Sub extends Super {
}
```

After:

```
package org.eclipse.internal.p1;
public class Super {
}

package org.eclipse.p1;
public class Sub extends Super {
}
```

- Not binary compatible
- Protected members accessible from an API type are API
- Is this valid if class Sub is final or says clients must not subclass?

Binary compatibility #10

Before:

```
public class E extends Exception {}

public class Test {
    protected void foo() throws E {}
}
```

After:

```
public class E extends Exception {}

public class Test {
    protected void foo() {}
}
```

- Binary compatible
- There is no distinction between checked and unchecked exceptions at runtime
- Not source compatible because catch blocks in referring methods may become unreachable

Binary compatibility #11

Before:

```
public class A {
    public void foo(String s) {}
}
public class C extends A {
    public void foo(String s) {
        super.foo(s);
    }
}
```

After:

```
public class A {
    public void foo(String s) {}
}
public class B extends A {}
public class C extends B {
    public void foo(String s) {
        super.foo(s);
    }
}
```

- Binary compatible
- The super-type structure can be changed as long as the available methods and fields don't change

Binary compatibility #12

Before:

```
public class Test {
}
```

After:

```
public class Test {
    public Test(String s) {
    }
}
```

- Not binary or source compatible
- When a constructor is added, the default constructor is no longer generated by the compiler. References to the default constructor are now invalid
- You should **always** specify at least one constructor for every API class to prevent the default constructor from coming into play (even if it is private)
- A constructor generated by the compiler also won't appear in javadoc

Binary compatibility #13

Before:

```
public class Test {
    public void foo() {}
}
```

After:

```
public class Test {
    public boolean foo() {
        return true;
    }
}
```

- Not binary compatible because the return type is part of the method signature
- Source compatible only if the return type was previously void

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* *http://www.eclipse.org/eclipse/development/java-api-evolution.html*

Evolving APIs

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

Example: Color your turtle

- Start with the original turtle example
- Evolve the API to add the notion of pen color
- Must maintain binary compatibility and contract compatibility for clients

Approaches to adding color to turtle

```
public interface IColorTurtle extends ITurtle {
   public void setColor(int red, int green, int blue);
}
```

- Doesn’t pollute the existing ITurtle interface with an option that all turtles might not implement
- This solution doesn’t scale – new sub-interface needed for every new attribute

Simple color turtle solution

```
public class PenColor {
    public PenColor(int red, int green, int blue) {…}
    public int getBlue() {…}
    public int getGreen() {…}
    public int getRed() {…}
}
public interface ITurtle {
    /**Sets the color of this turtle's pen. */
    public void setPenColor(PenColor color);
}
public class TurtlePen {
    public void setColor(PenColor color) {…}
}
```


Problems with simple solution

- What happens when this code is run?

```
TurtlePen pen = new TurtlePen();
TurtleState state = new TurtleState();
ITurtle t1 = TurtleFactory.createTurtle(state, pen);
ITurtle t2 = TurtleFactory.createTurtle(state, pen);
t1.setPenColor(PenColor.BLUE);
```

- What is the color of t2's pen? If a pen can be shared by two turtles, it is problematic that calling setPenColor on one turtle will change the pen color of another turtle

Solving the problem with shared pen state

- Copy the pen when it is added to a turtle to prevent sharing
- Or, make the setColor method non destructive

```
public interface ITurtle {
   public void setPenColor(PenColor color) {
     this.pen = pen.setColor(color);
   }
}
public class TurtlePen {
   public TurtlePen setColor(PenColor color) {…}
}
```

- Now altering the state of a turtle's pen does not alter the pen that may be shared with other turtles

What about the ITurtle reset method?

- Do we add color to the contract of the reset method?

/** Sets the position, direction **and pen color** of the turtle back to its initial state */
public void reset();

- Now we need to incorporate color into TurtleState:

```
public class TurtleState {
    public TurtleState(double x, double y, double dir, boolean down, PenColor color)
    public PenColor getPenColor()
}
```


Other considerations with color turtle

- Perhaps PenColor is too specific, if later we may need different line styles (dotted lines), join styles (miter, bevel, etc)
- We could have a PenStyle class instead of PenColor to encapsulate all mutable pen attributes

- What about old pens that haven't been adapted for color? Client contract must state that not all pens are capable of color
- Clients may want the ability to query if color is supported. Can add a query method to ITurtle that returns false for old pens, but new pens may return true:

```
public static final int ATTRIBUTE_COLOR = 1;
public boolean hasAttribute(int);
```


Summary of color turtle

- Must consider the whole picture when adding API
- Determine what needs to be added elsewhere in the API to make it consistent
- Ensure you handle backwards compatibility for clients and implementations

Techniques for evolving APIs

- Create extension interfaces, use naming convention (ITurtleExtension, ITurtle2)
- Wiegand's device
- Deprecate and try again
- Proxy that implements old API by calling new API

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

API Process

Process issues: Start small, start early, ...

- Always work with a client
- API first
- Start small
- Start early
- Only one chance (in each namespace)
- How to use deprecation
- Package naming conventions (provisional packages, SPI packages)

Plug-in Version Numbers

- major.minor.service.qualifier
- From release to release, the version number changes as follows:
  - When you break the API, increment the major number
  - When you change the API in an (binary) upwards compatible way, increment the minor number
  - When you make other changes, increment the service number
  - The qualifier is changed for each new build submission

"Provisional" API

- Before the API freeze
  - new API added is provisional by definition
- After the API freeze
  - Real API or internal code

