

J2SE™ 1.5

Taming the tiger - A developer's tour



Introduction

- ◆ Advisory Developer - delta.com
- ◆ 12 years experience
- ◆ Sun Certified Programmer and Web Component Developer for the Java™ 2 platform
- ◆ Open source developer – CacheCow, Edgar, Commando, WebDoctor
- ◆ Holds a B.Sc. (Hons) Computer Studies from Nottingham Trent University
- ◆ Interests include Java, Linux, web development
- ◆ Maintains blog at jason.blog-city.com
- ◆ Opinions expressed do not necessarily reflect those of Delta Air Lines, Inc. or Delta Technology Inc.



Contents

- ◆ Java™ - Pause for reflection
- ◆ The anatomy of Tiger
- ◆ Exploration of the new language features
 - JSR14 – Generics
 - JSR201 – Enumerations, Autoboxing, Enhanced for loops, static import, varargs
 - JSR175 – Metadata facility
- ◆ What else
- ◆ Recap





Java™ – Pause for reflection





Atlanta Java Users Group
October 2003

© Jason Chambers



The anatomy of Tiger

- ◆ JSR 176
- ◆ Next major release of J2SE™
- ◆ Comprises 15 component JSRs for major new features
 - New APIs
 - **New language features**
- ◆ Upgrades to existing features
- ◆ Bug fixes
- ◆ Performance improvements
- ◆ Scheduled for summer of 2004
- ◆ Focus on quality, stability and compatibility, performance, scalability, ease of development, monitoring and manageability, desktop client



Component JSRs

003 JMX™ Management API

013 Decimal Arithmetic

014 Generic Types

028 SASL

114 JDBC™ API Rowsets

133 New Memory Model

163 Profiling API

166 Concurrency Utilities

174 JVM™ Software Monitoring
and Management

175 Metadata

199 Compiler APIs

200 Pack Transfer Format

201 Language Updates

204 Unicode Surrogates

206 JAXP 1.3



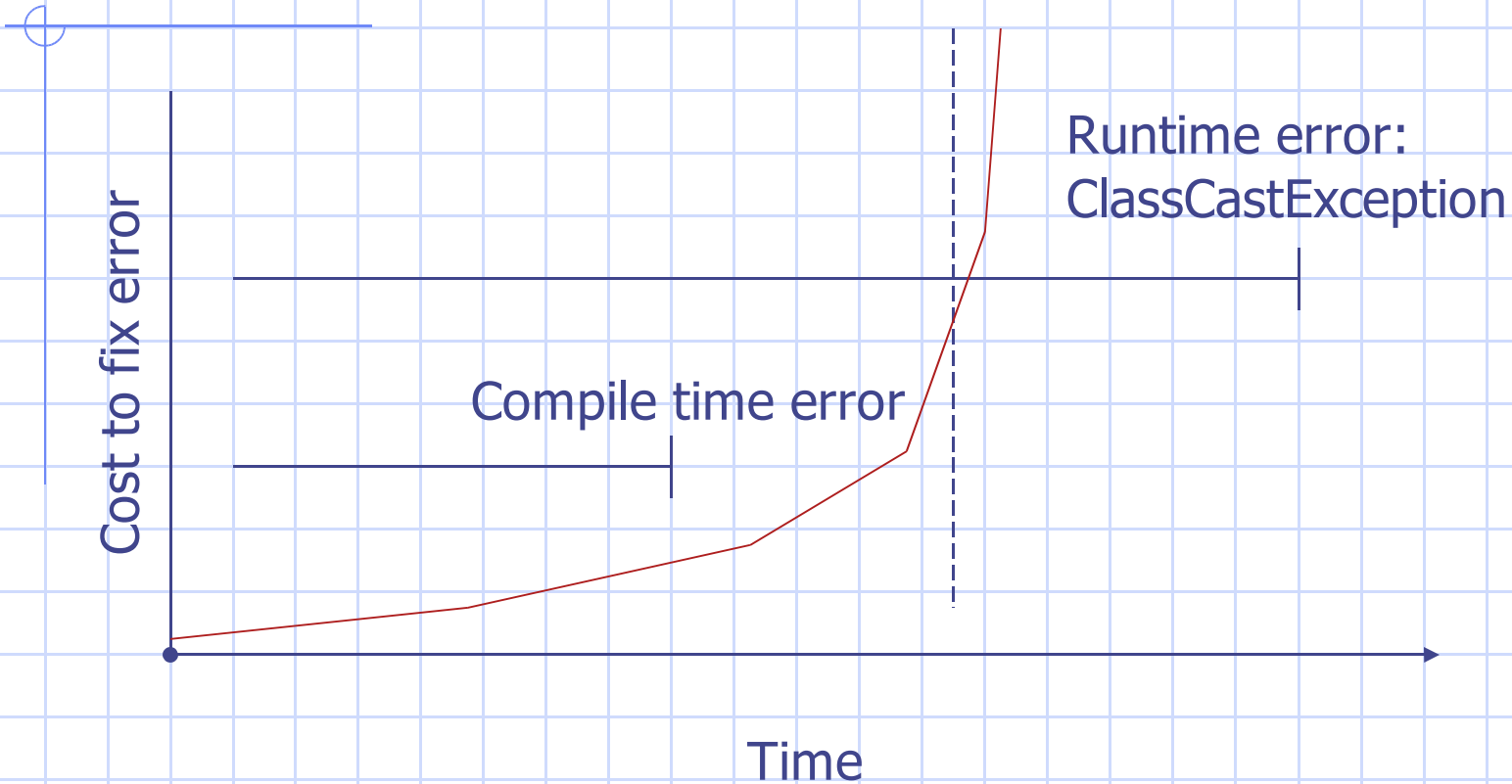
JSR 014 Generic Types

- ◆ Aka Parametric Polymorphism
- ◆ Classes, Interfaces and Methods can be parameterized by Types
- ◆ In OOP class we learned a class is a blueprint for an object
- ◆ A generic class is a blueprint for a class
- ◆ Have been around for years in other languages C++ (templates), Functional languages, Eiffel and so on
- ◆ Why? Improve quality, re-use and to find defects earlier
- ◆ Syntax familiar to C++ developers
- ◆ JSR 14 – 4 years in the making
- ◆ Controversial - it does introduce opportunities for greater source code complexity
- ◆ Frequently requested by the developer community



JSR 014 Generic Types

Rover launch date



JSR 014 Generic Types

Usage example – my intention is to have a list of words

◆ Today (no Generics)

```
// This list to contain Strings only.. Or else  
List words = new ArraryList();  
// Add to the list  
words.add(new Integer(10));  
// Pull from the list  
String aWord = (String)words.get(0); *1
```

◆ *1 - Results in `java.lang.ClassCastException`



JSR 014 Generic Types

Usage example (cont'd)

◆ Future (with Generics)

```
// This list to contain Strings only  
List<String> words = new ArrayList<String>(); *1  
// Add to the list  
words.add(new Integer(10)); *2  
// Pull from the list  
String aWord = words.get(0); *3
```

- ◆ *1 - Intent of this List is obvious – to both human and machine
- ◆ *2 - Compilation error
- ◆ *3 - No need to downcast



JSR 014 Generic Types

Example – How to define generic classes/interfaces

◆ Interface

```
interface List<T> {  
    void add(T x);  
    T get(int index);  
}
```

◆ Implementation

```
class LinkedList<T> implements List<T> {..}
```

◆ Usage

```
List<Passenger> passengers =  
    new LinkedList<Passenger>;
```



JSR 014 Generic Types

Example – Generic methods

◆ Declaration – type parameters precede return type

```
static <Elem> void swap(Elem[] a, int i, int j)
{
    Elem temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

◆ No special syntax for invocation

```
swap(strings, 1, 3);
swap(ints, 1, 2);
```



JSR 014 Generic Types

Example – Getting fancy with the type parameters

`<T1, T2>`

`<T extends Foo>`

`<T1, T2 implements Convertible<T1>>`



JSR 014 Generic Types

Exceptions

- ◆ Type parameters are not allowed in `catch` clauses
- ◆ They are allowed in `throws` lists:

```
interface Command<E> {  
    void execute() throws E;  
}
```



JSR 014 Generic Types

Implementation details

- ◆ Implementation of Generics in Java differs from C++ templates
- ◆ Type parameters are erased during the translation to JVM bytecodes
 - `(new Vector<String>).getClass().getName()` yields `java.util.Vector`
 - `(new Vector<Integer>).getClass().getName()` yields `java.util.Vector`
- ◆ The downcasts are inserted for you by the compiler
- ◆ No code bloat (like in C++)
- ◆ Compatibility is the driving force - No JVM changes



JSR 014 Generic Types

Implementation details

◆ Question – Is the following overloading of x legal?

```
void x(LinkedList<Integer> list);
```

```
void x(LinkedList<String> list);
```

◆ Answer – no because the generic types are erased





Generics demo time



JSR 201 Language Updates

- ◆ Enumerations
- ◆ Autoboxing
- ◆ Enhanced for loops
- ◆ Static import
- ◆ Varargs



JSR 201 - Enumerations

- ◆ A type whose legal values consist of a fixed set of constants
- ◆ Similar to C enumerated types.. But better in Java
- ◆ Typesafe in Java
- ◆ Not typesafe in C (essentially an int)
- ◆ New keyword introduced – enum
- ◆ Examples of enumerations – days of week, card suit, direction
- ◆ Enhances readability of code



JSR 201 - Enumerations

Without enum

```
class Direction {
    public static final int NOTSET = 0;
    public static final int NORTH = 1;
    public static final int SOUTH = 2;
    public static final int EAST = 3;
    public static final int WEST = 4;
}

class Aircraft {
    private int heading;
    void setHeading(int h) { heading = h;}
}

..
Aircraft.setHeading(Direction.NORTH);
Aircraft.setHeading(55); // Ouch - Will compile & run !!!!
```



JSR 201 - Enumerations

Without enum – typesafe enum pattern (Bloch)

```
class Direction {
    private final String name;
    private Direction(String name) {this.name = name;}
    public String toString() { return name; }
    public static final Direction NOTSET = new Direction("NOTSET");
    public static final Direction NORTH = new Direction("NORTH");
    public static final Direction SOUTH = new Direction("SOUTH");
    public static final Direction EAST = new Direction("EAST");
    public static final Direction WEST = new Direction("WEST");
}

class Aircraft {
    private Direction heading;
    void setHeading(Direction h) { heading = h;}
}

..
Aircraft.setHeading(Direction.NORTH);
Aircraft.setHeading(55); // Will not compile!
```



JSR 201 - Enumerations

Re-written using enum – typesafe and elegant

```
enum Direction { NOTSET, NORTH, SOUTH, EAST, WEST};

class Aircraft {
    private Direction heading;
    void setHeading(Direction h) { heading = h;}
}
..
Aircraft.setHeading(Direction.NORTH);
```



JSR 201 - Enumerations

Bonuses of the enum

- ◆ You can use them in the switch construct

```
switch (aircraft.getHeading()) {  
    case Direction.NORTH: ..  
    case Direction.SOUTH: ..  
    case Direction.EAST: ..  
    case Direction.WEST: ..  
}
```

- ◆ Easier to debug – the name is printed out not the number

```
System.out.println(aircraft.getHeading());
```

- ◆ Changing the order or adding new constants doesn't break anything

- ◆ You can use enum constants in collections (HashMap keys)



JSR 201 - Enumerations

Bonuses of the enum

- ◆ You can iterate over the enum values (more later)

```
for (Direction direction : Direction.VALUES)
    System.out.println(direction);
```

NOTSET

NORTH

SOUTH

EAST

WEST

- ◆ An enum declaration is a special kind-of class declaration – you can add methods if you want (be careful)





Enumerations demo time



JSR 201 - Autoboxing

- ◆ Java has a two-headed type system
 - Object references (classes, interfaces, arrays)
 - Primitives (int, float, boolean etc.)
- ◆ What if you want a collection of primitives?
- ◆ Conversion between the two is necessary yet cumbersome
- ◆ Wrapper types used (Integer<->int, Float<->float ..)
- ◆ Autoboxing provides automatic conversion between the primitive types and their corresponding wrapper types (boxing) and vice-versa (un-boxing)
- ◆ Compiler creates the conversion for you



JSR 201 - Autoboxing

◆ Without autoboxing

```
Map daysInMonth = new HashMap();  
daysInMonth.put("January", new Integer(31));
```



JSR 201 - Autoboxing

◆ With autoboxing (and Generics thrown in)

```
Map<String,Integer> daysInMonth = new  
    HashMap<String,Integer>();  
daysInMonth.put("January", 31);
```

◆ Typesafe & elegant

◆ No need to explicitly create a wrapper object




JSR 201 - Autoboxing

Boxing and un-boxing in a nutshell

```
public class Test {  
    public static void box(Integer i) {  
        System.out.println(i);  
    }  
    public static void unbox(int i) {  
        System.out.println(i);  
    }  
    public static void main(String[] args) {  
        box(1);  
        unbox(new Integer(5));  
    }  
}
```





Autoboxing demo time



JSR 201 – Enhanced for loops

◆ A fairly common thing to do in Java

```
Iterator iter = words.iterator();  
while (iter.hasNext()) {  
    String s = (String)iter.next  
    // Do something the element  
    System.out.println(s);  
}
```

◆ Cumbersome

◆ Have to create an Iterator object



JSR 201 – Enhanced for loops

◆ Re-written using the enhanced for loop

```
for (String s : words) {  
    // Do something with element  
    System.out.println(s);  
}
```

◆ Tighter

◆ Harmonious with generics – no need to cast

◆ Read the : operator as in



JSR 201 – Enhanced for loops

◆ Also works with arrays

```
String[] words = { "abc", "bill", "fred" };  
for (String s : words)  
    System.out.println(s);
```

◆ And the values of enumerated data types

```
for (Direction direction : Direction.VALUES)  
    System.out.println(direction);
```





Enhanced for loops demo time



JSR 201 – Static imports

- ◆ Today we use the import statement to import classes and interfaces from other packages
- ◆ This device stops us from having to fully-qualify them in code

```
java.util.List<String> = new  
    java.util.ArrayList<String>();
```

◆ Much better:

```
import java.util.List;  
import java.util.ArrayList;  
...  
List<String> = new ArrayList<String>();
```



JSR 201 – Static imports

- ◆ Today's import only goes so far as the class/interface level.
- ◆ We have to explicitly qualify static class members

```
double d = Math.sqrt(4);  
System.out.println(d);  
System.out.println(Math.PI);
```

- ◆ Much better:

```
import static java.lang.Math.*;  
...  
double d = sqrt(4);  
System.out.println(d);  
System.out.println(PI);
```





Static imports demo time



JSR 201 – Varargs

- ◆ Use when you have a method that is to take an arbitrary number of parameters
- ◆ Today, you would use an array
- ◆ Creating/initializing arrays is cumbersome

```
Object[] arr1 = {new Date(), new Integer(10),  
    new Integer(5)};  
methodWithoutVarArgs("Hello", arr1);
```

- ◆ Let the compiler do it for you

```
methodWithVarArgs("Hello", new Date(), 10, 5);
```



JSR 201 – Varargs

◆ Declaration

```
varargsMethod(String pattern, Object... args)
```

◆ Type of args is Object[]

◆ args has to be the last argument to the method

```
varargsMethod(Object... Args, String s)
```

◆ ... is invalid



JSR 201 – Varargs

```
public static void xxx (Object... arguments) {  
    // arguments comes through as an Object[]  
    System.out.println(arguments.length);  
  
    // Which means you can iterate over it  
    for (Object o : arguments)  
        System.out.println(o.getClass().getName());  
}
```





Varargs demo time



JSR 201 – Varargs

- ◆ Varargs and autoboxing – ingredients for simple formatted I/O popular with C/C++ developers
- ◆ Does this bring back memories?

```
out.printf("There are %d days in a week", 7);
```

- ◆ Simple formatted I/O promised in 1.5



New language features summary

◆ JSR14 – Generics

- Classes/Interfaces/Methods parameterized by types
- Similar (syntactic) feel to C++ templates
- Compiler inserts casts for you
- Push more errors to compile-time

◆ JSR201

- Enumerations – `enum season {spring, summer, autumn, winter}`
- Autoboxing – conversions between wrappers and primitives
- Enhanced for loops – `for (String s : collection)`
- Static import – avoid having to qualify static members
- Varargs – compiler generates args array for you



Opinion

- ◆ New language features integrated well with each other and existing language
- ◆ Difficult to keep 2 million developers happy – impossible to keep 10 million developers happy
- ◆ The challenge – simplicity versus language functionality
- ◆ Language design is hard – especially for general purpose languages
- ◆ The more features you add – the more ways you introduce to do things – the opportunities for complexity increase
- ◆ All-in-all – welcome additions to the language



JSR 175 Metadata

- ◆ Goal is to make J2EE easier
- ◆ Annotate classes methods and fields
- ◆ In C# parlance - referred to as attributes
- ◆ In Java parlance – referred to as metadata
- ◆ The metadata is picked up by other tools to generate boilerplate code
- ◆ For example, declare a class as being Persistent, or perhaps declare it as being exposed as a web service
- ◆ XDoclet



JSR 175 Metadata

◆ Scope of JSR 175:

- Language extension to allow metadata to be supplied for classes, interfaces, methods and fields
- Format and standard way for tools to access the metadata (build/deployment/run time?)

◆ Out of scope:

- The actual metadata are defined elsewhere e.g. JSR 181 defines the metadata for Web Services



JSR 175 Metadata

◆ Java Example

```
import javax.xml.rpc.*;

public class ReservationService {
    @Remote public Confirmation issueTicket(Reservation reservation) {
        ..
    }
}
```

◆ C# example

```
public class ReservationService: WebService {
    [WebMethod]
    public Confirmation issueTicket(Reservation reservation) {
        ..
    }
}
```





What else in Tiger?



JSR 003 JMX™ Specification

- ◆ Java™ Management Extensions
- ◆ Monitoring - is my application/device/service healthy?
- ◆ Recovery - make it healthy
- ◆ Expose resources/functionality as MBeans
- ◆ Resources include connection pools, thread pools, application resources
- ◆ Using a system management dashboard can control the resources. E.g. turn on logging, make the application change direction, grow a connection pool
- ◆ Applicable to all who are interested in keeping their systems running (most of us)



JSR 013 Decimal Arithmetic Enhancement

- ◆ `java.math.BigDecimal`
- ◆ Adding floating point arithmetic
- ◆ Allowing use of decimal numbers for general purpose arithmetic
- ◆ Financial applications



JSR 028 SASL

- ◆ Simple Authentication and Security Layer (SASL)
- ◆ Standard for connection-based protocols such as LDAP and IMAP
- ◆ SASL specifies challenge-response protocol in which data is exchanged between client and server for the purposes of authentication.
- ◆ JSR 028 defines the API for SASL with a pluggable authentication framework
- ◆ Applicable more for infrastructure developers than application developers
- ◆ SASL is to authentication what JSSE is to confidentiality & integrity



JSR 114 JDBC Rowset

- ◆ Makes it easy to send tabular data between tiers and components
- ◆ RowSet encapsulates a set of rows that have been retrieved from a tabular data source (e.g. database table, spreadsheet etc.)
- ◆ A RowSet object is simply a ResultSet object that can function as a JavaBeans component (events, properties etc.)
- ◆ As a JavaBeans component, a RowSet can be created and configured at design-time
- ◆ How else does a RowSet differ from a ResultSet?
 - A RowSet can update it's rows while disconnected from the datasource – and be synchronized when re-connected
 - Provides greater flexibility and scalability



JSR 114 JDBC Rowset

```
CachedRowSet crset = new CachedRowSet();  
// Configure the RowSet  
crset.setCommand("SELECT * FROM DUTYFREE");  
crset.setDataSourceName("jdbc/coffeesDB");  
crset.setUsername("juanvaldez");  
crset.setPassword("espresso");  
// Setup event notification  
crset.addRowSetListener(shoppingbasket);  
// Get a connection, execute and populate the RowSet  
crset.execute();  
// Don't need the connection anymore! - Captain ready for takeoff  
while (crset.next()) {  
    //System.out.println(crset.getString("ITEM"));  
}  
// Make a change to the 3rd and 4th columns of the 4th row  
crset.absolute(4);  
crset.updateFloat(3, 9.49f);  
crset.updateInt(4, 500);  
crset.updateRow(); // Event triggered  
// When we land, we want to sync our changes with the database  
crset.acceptChanges();
```



JSR 133 Java™ Memory Model and Thread Specification Revision

- ◆ No new APIs
- ◆ Revisions to “The Java Language Specification” and “The Java Virtual Machine Specification”
- ◆ To clarify the semantics of threads, locks, volatile variables and data races



JSR 163 Profiling API

- ◆ A mechanism and APIs for extracting time and space profiling information from a running Java™ virtual machine.
- ◆ Includes a wire protocol and remote API
- ◆ Supercedes JVMPI (experimental/flawed design)
- ◆ Targeted at tool vendors



JSR 166 Concurrency Utilities

- ◆ Java provides low-level threading primitives such as synchronized blocks, `Object.wait` and `Object.notify`
- ◆ Insufficient for many concurrent programming tasks
- ◆ 166 provides implementation of well-established higher-level concurrency abstractions
- ◆ Semaphore, Mutex, Latch, ReadWrite Locks, Channels etc.



JSR 174 Monitoring and Management Specification for the JVM

- ◆ Standard API for monitoring of garbage collection, memory, threads, heap etc.
- ◆ Run-time controls – minimum heap size, verbose GC on demand, garbage collection, thread creation
- ◆ Support for JMX (003)
- ◆ Functional overlap with Profiling API (163)



JSR 199 Java™ Compiler API

- ◆ For those that write code that compiles other code
- ◆ E.g. Assume you were to write a JSP engine, or you were writing an IDE
- ◆ Today, stick tools.jar on your CLASSPATH and:

```
int compileReturnCode =  
    com.sun.tools.javac.Main.compile( new String[]  
        {"MyClass.java"});
```

- ◆ Problem is javac is filesystem based
- ◆ What if my code is in memory or another location?
- ◆ What if I want my output to go somewhere else?



JSR 200 Network Transfer Format for Java™ Archives

- ◆ Make the deployment of Java applications faster and more network efficient (smaller)
- ◆ JAR format is not so efficient at compression (byte-level compression only)
- ◆ May use the Pack format



JSR 204 Unicode Supplementary Character Support

- ◆ 16 bits not enough room anymore (since Unicode 3.1)
- ◆ Need to support supplementary characters which fall outside of the range U+0000-U+FFFF
- ◆ 204 will support the supplementary characters in a way that is backwards compatible



Summary

003 JMX™ Management API

013 Decimal Arithmetic

014 Generic Types

028 SASL

114 JDBC™ API Rowsets

133 New Memory Model

163 Profiling API

166 Concurrency Utilities

174 JVM™ Software Monitoring
and Management

175 Metadata

199 Compiler APIs

200 Pack Transfer Format

201 Language Updates

204 Unicode Surrogates

206 JAXP 1.3



Further reading

- ◆ JCP <http://www.jcp.org>
- ◆ Discussion of the new language features
<http://developer.java.sun.com/developer/community/chat/JavaLive/2003/jl0729.html>
- ◆ Conversation with Bloch http://java.sun.com/features/2003/05/bloch_qa.html
- ◆ J2SE Roadmap from JavaOne 2003
<http://servlet.java.sun.com/javaone/resources/content/sf2003/conf/sessions/pdfs/1540.pdf>
- ◆ New language features from JavaOne 2003
<http://servlet.java.sun.com/javaone/resources/content/sf2003/conf/sessions/pdfs/3072.pdf>

