

Taming the Tiger

java.sun.com/javaone/sf

Joshua Bloch, Distinguished Engineer
Neal Gafter, Senior Staff Engineer
Sun Microsystems



Watch Out for Tigers!

- J2SE™ 1.5 platform
- Code name “Tiger”
- Beta2—available for download now!
- A major theme—ease of development
- Seven new language features



New Language Features

- Generics
- For-each loop
- Autoboxing/Unboxing
- Enums
- Varargs
- Static Import
- Annotations (metadata)

Unifying Theme: Developer-Friendliness

- Make programs clearer, shorter, and safer
 - Linguistic support for common idioms
 - Compiler, not programmer, writes boilerplate code
- Interact well with one another, existing features
- New features do not:
 - Sacrifice compatibility
 - Compromise the spirit of the language

Goal of This Talk

Enable you to start using J2SE 1.5 platform today!

Familiarize you with new language features

Show you how they work together

Show you common idioms

Show you when and when *not* to use them

Outline

- I. Generics (with for-each and autoboxing)
- II. Enums (with all of the above features)
- III. Varargs
- IV. Static import

I. Generics

- Allow you to specify element type of collection
- Enforce specification at compile time
- “Stronger typing with less typing”
- Go way beyond collections

A Simple Example With For-each

```
void cancelAll(Collection<TimerTask> c) {  
    for (TimerTask task : c)  
        task.cancel();  
}
```

- Clearly specifies the element type
 - Checked at compile time
- The loop is concise and readable

We've *Generified* Collection Interface

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);

    boolean add(E o);
    boolean remove(Object o);

    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
}
```

Generics, Autoboxing, and For-each

```
// Print a frequency table of words on command line
public class Frequency {
    public static void main(String[] args) {
        Map<String, Integer> m = new TreeMap<String, Integer>
        ();
        for (String word : args) {
            Integer freq = m.get(word);
            m.put(word, (freq == null ? 1 : freq + 1));
        }
        System.out.println(m);
    }
}
```

```
$ java Frequency if it is to be it is up to me
{be=1, if=1, is=2, it=2, me=1, to=2, up=1}
```

Generics and Reflection

- Class `Class` has been generified
 - Class literal `Foo.class` is of type `Class<Foo>`
- Enables compile-time type-safe reflection
 - `Foo foo = Foo.class.newInstance();`
- Enables strongly typed static factories

```
<T extends Annotation>
```

```
    T getAnnotation(Class<T> annotationType);
```

```
Author a = Othello.class.getAnnotation(Author.class);
```

When to Use Generics

- Any time you can
 - Unless you need to run on pre-1.5 VM
- The extra effort in generifying code is worth it!
 - Increased clarity and type safety

When to Use For-each

- Any time you can!
 - Really beautifies code
- You *can't* use for-each for these cases
 - Removing elements as you traverse collection
 - Modifying the current slot in an array or list
 - Iterating over multiple collections or arrays

When to Use For-each in APIs

- A class should implement **Iterable**
 - When API defines a sequence of values
 - Unless a Collection or array is sufficient
- Can retrofit existing classes
- Can always provide an **Iterable adapter**

When to Use Autoboxing

- When there is an “impedance mismatch” between reference types and primitives
 - For example, numerical value into collection
- *Not* appropriate for scientific computing
 - Performance sensitive numerical code
- An `Integer` is not a substitute for an `int`

II. Enums

- Linguistic support for enumerated type
- Advanced object-oriented features
 - Can add methods and fields to enums
- Much clearer, safer, more powerful than existing alternatives

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

Immutable Playing Card

```
public class Card implements Serializable {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,
        SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    private final Rank rank;
    private final Suit suit;
    private Card(Rank rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public Rank rank() { return rank; }
    public Suit suit() { return suit; }
    public String toString() { return rank+" of "+suit;}
}
```

Deck Factory

```
private static final List<Card> protoDeck =  
    new ArrayList<Card>();  
  
static {  
    for (Suit suit : Suit.values())  
        for (Rank rank : Rank.values())  
            protoDeck.add(new Card(rank, suit));  
}  
  
public static ArrayList<Card> newDeck() {  
    return new ArrayList<Card>(protoDeck);  
}
```

Deal—Program to Exercise Card

```
public class Deal {
    public static void main(String args[]) {
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);
        List<Card> deck = Card.newDeck();
        Collections.shuffle(deck);
        for (int i=0; i < numHands; i++)
            System.out.println(deal(deck, cardsPerHand));
    }

    public static ArrayList<Card> deal(List<Card> deck, int n)
    {
        int deckSize = deck.size();
        List<Card> handView = deck.subList(deckSize-n,
deckSize);
        ArrayList<Card> hand = new ArrayList<Card>(handView);
        handView.clear();
        return hand;
    }
}
```

Watch It Go

```
$ java Deal 4 5
```

```
[TEN of SPADES, NINE of SPADES, JACK of CLUBS,  
ACE of CLUBS, JACK of HEARTS]
```

```
[KING of HEARTS, DEUCE of HEARTS, FOUR of CLUBS,  
SIX of HEARTS, DEUCE of CLUBS]
```

```
[KING of CLUBS, SIX of DIAMONDS, SEVEN of  
HEARTS, THREE of HEARTS, EIGHT of DIAMONDS]
```

```
[FOUR of HEARTS, QUEEN of SPADES, FIVE of  
DIAMONDS, SIX of CLUBS, QUEEN of CLUBS]
```

Adding Data and Computation

```
public enum Planet {
    VENUS(4.8685e24, 6051.8e3),
    EARTH(5.9736e24, 6378.1e3),
    MARS(0.64185e24, 3397e3);

    final double mass;    // in kilograms
    final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public static final double G = 6.67300E-11;
    double surfaceGravity()
        { return G * mass / (radius * radius); }
    double surfaceWeight(double otherMass)
        { return otherMass * surfaceGravity(); }
}
```

Watch It Go

```
public static void main(String[] args) {
    double earthWeight = Double.parseDouble(args[0]);
    double mass = earthWeight/EARTH.surfaceGravity();
    for (Planet p : Planet.values()) {
        System.out.printf("Your weight on %s is %f%n",
                          p, p.surfaceWeight(mass));
    }
}
```

```
$ java Planet 100
```

```
Your weight on VENUS is 90.525837
```

```
Your weight on EARTH is 100.000000
```

```
Your weight on MARS is 37.878168
```

Adding Constant-Specific Functionality

```
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    // Do arithmetic op represented by this constant
    double eval(double x, double y) {
        switch(this) {
            case PLUS:    return x + y;
            case MINUS:   return x - y;
            case TIMES:   return x * y;
            case DIVIDE:  return x / y;
        }
        throw new AssertionError("Unknown op: " + this);
    }
}
```

Adding Constant-Specific Methods

```
public enum Operation {
    PLUS {double eval(double x, double y) {return x+y;}},
    MINUS {double eval(double x, double y) {return x-y;}},
    TIMES {double eval(double x, double y) {return x*y;}},
    DIVIDE{double eval(double x, double y) {return x/y;}};

    // Do arithmetic op represented by this constant
    abstract double eval(double x, double y);
}
```

Watch It Go

```
public static void main(String args[]) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    for (Operation op : Operation.values())
        System.out.printf("%f %s %f = %f%n",
                           x, op, y, op.eval(x, y));
}
```

```
$ java Operation 4 2
```

```
4.000000 PLUS 2.000000 = 6.000000
```

```
4.000000 MINUS 2.000000 = 2.000000
```

```
4.000000 TIMES 2.000000 = 8.000000
```

```
4.000000 DIVIDE 2.000000 = 2.000000
```

EnumSet and EnumMap

- **EnumSet**—high-performance **Set** for enums
 - Implemented as bit-vector
- Provides range iteration

```
for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
```
- Replaces traditional bit-flags

```
EnumSet.of(Style.BOLD, Style.ITALIC)
```
- **EnumMap**—Analogous **Map** for enum keys
 - Implemented as array

When to Use Enums

- Natural enumerated types
 - Days of week, phases of moon, seasons, compass points, colors in spectrum
- Other sets where you know all possible values
 - Choices on Menu, Rounding modes, command line flags, opcodes in instruction set
 - Can add more in new version of API
- As a replacement for flags (**EnumSet**)

III. Varargs

- A method that takes an arbitrary number of values requires you to create an array
- Varargs automates and hides the process
 - Can retrofit existing APIs
 - Upward compatible with existing clients

```
public static String format(String pattern,  
                             Object... arguments);
```

```
String result = MessageFormat.format(  
    "At {1,time} on {1,date}, there was {2} on planet "  
    + "{0,number,integer}.", 7, new Date(),  
    "a disturbance in the Force");
```

Synergy Between Varargs and Reflection

Example—Simple Test Framework

```
public static void main(String[] args) {
    int passed = 0;
    int failed = 0;
    for (String className : args) {
        try {
            Class c = Class.forName(className);
            c.getMethod("test").invoke(c.newInstance());
            passed++;
        } catch (Exception ex) {
            System.out.printf("%s failed: %s%n",
                               className, ex);
            failed++;
        }
    }
    System.out.printf("passed=%d; failed=%d%n",
                      passed, failed);
}
```

When to Use Varargs

- In APIs, sparingly
 - Only when the benefit is compelling
 - Don't overload a varargs method
- In clients, when the API supports them
 - Reflection, message formatting, `printf`

IV. Static Import

- Clients must qualify static members with class name (`Math.PI`)
- To avoid this, some programmers put constants in interface and implement it
 - **Bad!** — “Constant Interface Antipattern”
- Static import allows unqualified access to static members without extending a type

```
import static java.lang.Math.*;
```

```
r = cos(PI * theta);
```

When to Use Static Import

- Very sparingly!
- Only when tempted to abuse inheritance
- Overuse makes programs unreadable

Conclusion

- This release is all about you, the programmer
- Better programs with less effort
- Try it today!
 - Beta-2 release: <http://java.sun.com/j2se/1.5.0>

Taming the Tiger

java.sun.com/javaone/sf

Joshua Bloch, Distinguished Engineer
Neal Gafter, Senior Staff Engineer
Sun Microsystems



Additional Program Examples

List adapter for primitive int array

```
public static List<Integer> intList(final int[] a) {  
    return new AbstractList<Integer> () {  
        public Integer get(int i) { return a[i]; }  
        public Integer set(int i, Integer val) {  
            Integer oldVal = a[i];  
            a[i] = val;  
            return oldVal;  
        }  
        public int size() { return a.length; }  
    };  
}
```

Card Factory—Preserves Singleton

```
private static Map<Suit, Map<Rank, Card>> table =
    new EnumMap<Suit, Map<Rank, Card>>(Suit.class);
static {
    for (Suit suit : Suit.values()) {
        Map<Rank, Card> suitTable =
            new EnumMap<Rank, Card>(Rank.class);
        for (Rank rank : Rank.values())
            suitTable.put(rank, new Card(rank, suit));
        table.put(suit, suitTable);
    }
}
public static Card valueOf(Rank rank, Suit suit) {
    return table.get(suit).get(rank);
}
```

Pinochle Deck Factory

```
private static final List<Card> pinochleDeck =
    new ArrayList<Card>();

static {
    for (int i=0; i<2; i++) // Two of each card
        for (Suit suit : Suit.values())
            for (Rank rank : EnumSet.range(Rank.NINE, Rank.ACE))
                pinochleDeck.add(new Card(rank, suit));
}

public static ArrayList<Card> pinochleDeck() {
    return new ArrayList<Card>(pinochleDeck);
}
```

How Do Checked Wrappers Work?

```
public class SafeWrapper<T> {
    private Class<T> klass;
    private T value;
    public SafeWrapper(T initialValue, Class<T> klass) {
        this.klass = klass;
        set(initialValue);
    }
    public void set(T newValue) {
        if (!klass.isInstance(newValue))
            throw new ClassCastException();
        value = newValue;
    }
    public T get() { return value; }
}
```

Iterator Adapter: Codepoints in a String

```
Iterable<Integer> codePoints(final String s) {
    return new Iterable<Integer>() {
        public Iterator<Integer> iterator() {
            return new Iterator<Integer>() {
                int nextIndex = 0;
                public boolean hasNext() {
                    return nextIndex < s.length();
                }
                public Integer next() {
                    int result = s.codePointAt(nextIndex);
                    nextIndex += Character.charCount(result);
                    return result;
                }
                public void remove()
                    { throw new UnsupportedOperationException(); }
            };
        }
    };
}
```

Taming the Tiger

java.sun.com/javaone/sf

Joshua Bloch, Distinguished Engineer
Neal Gafter, Senior Staff Engineer
Sun Microsystems

