Forthcoming Java™ Programming Language Features

Joshua Bloch and Neal Gafter

Senior Staff Engineers
Sun Microsystems, Inc.

A Brief History of the Java™ Programming Language

- 1995 (1.0)—First public release
  - Hit a sweet spot!
- 1997 (1.1)—Nested classes added
  - Support for function objects
- 2001 (1.4)—Assertions added
  - Verify programmers understanding of code

Watch Out for Tigers!

- Java 2 Platform, Standard Edition Release 1.5
- Code name “Tiger”
- Beta—Early 2004
- A major theme—ease of development

Significant Language Changes Planned for Tiger

I. Generics
II. Enhanced for Loop ("foreach")
III. Autoboxing/Unboxing
IV. Typesafe Enums
V. Varargs
VI. Static Import
VII. Annotations

Unifying Theme—Developer-Friendliness

- Increase expressiveness
- Increase safety
- Minimize incompatibility
  - No substantive VM changes
  - All binaries, most sources run unchanged
  - New keywords kept to a minimum (1)

Disclaimer

- All subject to Java Community Process[SM]
  - JSR-014 Generics
  - JSR-175 Metadata (Annotations)
  - JSR-201 Remaining language changes
- For more information
  - http://www.jcp.org
- Participate!

I. Generics

- When you get an element from a collection, you have to cast
  - Casting is a pain
  - Casting is unsafe—casts may fail at runtime
- Wouldn’t it be nice if you could tell the compiler what type a collection holds?
  - Compiler could put in the casts for you
  - They’d be guaranteed* to succeed

* Offer void where prohibited by law. Price does not include dealer preparation and licensing. Your mileage may vary. Cash value 1/20c.

Filtering a Collection—Today

```
// Removes 4-letter words from c; elements must be strings
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (((String) i.next()).length() == 4)
            i.remove();
}

// Alternative form - a bit prettier?
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        String s = (String) i.next();
        if (s.length() == 4)
            i.remove();
    }
}
```

Filtering a Collection With Generics

```
// Removes 4-letter words from c
static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
        if (i.next().length() == 4)
            i.remove();
}
```

- Clearer and Safer
- No cast, extra parentheses, temporary variables
- Provides compile-time type checking

Generics Are Not Templates

- No code-size blowup
- No hideous complexity
- No “template metaprogramming”
- Simply provides compile-time type safety and eliminates the need for casts

II. Enhanced `for` Loop ("foreach")

- Iterating over collections is a pain
- Often, iterator unused except to get elements
- Iterators are error-prone
  - Iterator variable occurs three times per loop
  - Gives you two opportunities to get it wrong
  - Common cut-and-paste error
- Wouldn't it be nice if the compiler took care of the iterator for you?

Applying a Method to Each Element in a Collection—Today

```
void cancelAll(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        TimerTask tt = (TimerTask) i.next();
        tt.cancel();
    }
}
```

Applying Method to Each Element In a Collection With Enhanced `for`

```
void cancelAll(Collection c) {
    for (Object o : c)
        ((TimerTask)o).cancel();
}
```

- Clearer and Safer
- No iterator-related clutter
- No possibility of using the wrong iterator

Enhanced `for` Really Shines When Combined With Generics

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

- Much shorter, clearer and safer
- Code says exactly what it does

It Works For Arrays, Too

```
// Returns the sum of the elements of a
int sum(int[] a) {
    int result = 0;
    for (int i : a)
        result += i;
    return result;
}
```

- Eliminates array index rather than iterator
- Similar advantages

Nested Iteration Is Tricky

```
List suits = ...;
List ranks = ...;
List sortedDeck = new ArrayList();

// Broken - throws NoSuchElementException!
for (Iterator i = suits.iterator(); i.hasNext(); )
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(i.next(), j.next()));
```

Nested Iteration Is Tricky

```
List suits = ...;
List ranks = ...;
List sortedDeck = new ArrayList();

// Broken - throws NoSuchElementException!
for (Iterator i = suits.iterator(); i.hasNext(); )
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(i.next(), j.next()));

// Fixed - a bit ugly
for (Iterator i = suits.iterator(); i.hasNext(); ) {
    Suit suit = (Suit) i.next();
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(suit, j.next()));
}
```

With Enhanced `for`, It's Easy!

```
for (Suit suit : suits)
    for (Rank rank : ranks)
        sortedDeck.add(new Card(suit, rank));
```

III. Autoboxing/Unboxing

- You can’t put an `int` into a collection
  - Must use `Integer` instead
- It's a pain to convert back and forth
- Wouldn't it be nice if compiler did it for you?

Making a Frequency Table—Today

```
public class Freq {
    private static final Integer ONE = new Integer(1);

    public static void main(String[] args) {
        // Maps word (String) to frequency (Integer)
        Map m = new TreeMap();

        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE :
                  new Integer(freq.intValue() + 1)));
        }
        System.out.println(m);
    }
}
```

Making a Frequency Table With Autoboxing, Generics, and Enhanced `for`

```
public class Freq {
    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);
    }
}
```

IV. Typesafe Enums

Standard approach – `int` enum pattern

```
public class Almanac {
    public static final int SEASON_WINTER = 0;
    public static final int SEASON_SPRING = 1;
    public static final int SEASON_SUMMER = 2;
    public static final int SEASON_FALL   = 3;

    ...  // Remainder omitted
}
```

Disadvantages of `int` Enum Pattern

- Not typesafe
- No namespace - must prefix constants
- Brittle - constants compiled into clients
- Printed values uninformative

Current Solution – Typesafe Enum Pattern

- *"Effective Java Programming Language Guide"*
- Basic idea - class that exports self-typed constants and has no public constructor
- Fixes *all* disadvantages of **`int`** pattern
- Other advantages
  - Can add arbitrary methods, fields
  - Can implement interfaces

Typesafe Enum Pattern Example

```
import java.util.*;
import java.io.*;

public final class Season implements Comparable, Serializable {
    private final String name;
    public String toString()    { return name; }

    private Season(String name) { this.name = name; }

    public static final Season WINTER = new Season("winter");
    public static final Season SPRING = new Season("spring");
    public static final Season SUMMER = new Season(”summer");
    public static final Season FALL   = new Season("fall");

    private static int nextOrdinal = 0;
    private final  int ordinal = nextOrdinal++;

    public int compareTo(Object o) {
        return ordinal - ((Season)o).ordinal;
    }

    private static final Season[] PRIVATE_VALUES = { WINTER, SPRING, SUMMER, FALL };

    public static final List VALUES =
      Collections.unmodifiableList(
          Arrays.asList(PRIVATE_VALUES));

    private Object readResolve() {
        // Canonicalize
        return PRIVATE_VALUES[ordinal];
    }
}
```

Disadvantages of Typesafe Enum Pattern

- Verbose
- Error prone—each constant occurs 3 times
- Can't be used in **`switch`** statements
- Wouldn't it be nice if compiler took care of it?

Typesafe Enum Construct

- Compiler support for Typesafe Enum pattern
- Looks like traditional enum (C, C++, Pascal)
  - `enum Season { WINTER, SPRING, SUMMER, FALL }`
- Far more powerful
  - All advantages of Typesafe Enum pattern
  - Allows programmer to add arbitrary methods, fields
- Can be used in **`switch`** statements

Enums Interact Well With Generics and Enhanced `for`

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

List<Card> deck = new ArrayList<Card>();
for (Suit suit : Suit.values())
    for (Rank rank : Rank.values())
        deck.add(new Card(suit, rank));

Collections.shuffle(deck);
```

Would require *pages* of code today!

Enum With Field, Method and Constructor

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    Coin(int value) { this.value = value; }

    private final int value;

    public int value() { return value; }
}
```

Sample Program Using `Coin` Class

```
public class CoinTest {
    public static void main(String[] args) {
        for (Coin c : Coin.values())
            System.out.println(c + ":   \t"
                  + c.value() +"¢ \t" + color(c));
    }
    private enum CoinColor { COPPER, NICKEL, SILVER }
    private static CoinColor color(Coin c) {
        switch(c) {
          case PENNY:   return CoinColor.COPPER;
          case NICKEL:  return CoinColor.NICKEL;
          case DIME:
          case QUARTER: return CoinColor.SILVER;
          default: throw new AssertionError("Unknown coin: " + c);
        }
    }
}
```

Actual Output of Sample Program

```
PENNY:          1¢      COPPER
NICKEL:         5¢      NICKEL
DIME:           10¢     SILVER
QUARTER:        25¢     SILVER
```

V. Varargs

- To write a method that takes an arbitrary number of parameters, you must use an array
- Creating and initializing arrays is a pain
- Array literals are not pretty
- Wouldn’t it be nice if the compiler did it for you?
- Essential for a usable `printf` facility

Using `java.text.MessageFormat` —Today

```
Object[] arguments = {
    new Integer(7),
    new Date(),
    "a disturbance in the Force"
};

String result = MessageFormat.format(
    "At {1,time} on {1,date}, there was {2} on planet "
     + "{0,number,integer}.", arguments);
```

Using `MessageFormat` With Varargs

```
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");
```

Varargs Declaration Syntax

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

Parameter type of **`arguments`** is **`Object[]`**

Caller need not use varargs syntax

VI. Static Import Facility

Classes often export constants

```
public class Physics {
    public static final double
        AVOGADROS_NUMBER   = 6.02214199e23;
    public static final double
        BOLTZMANN_CONSTANT = 1.3806503e-23;
    public static final double
        ELECTRON_MASS      = 9.10938188e-31;
}
```

Clients must qualify constant names

```
double molecules = Physics.AVOGADROS_NUMBER * moles;
```

Wrong Way to Avoid Qualifying Names

```
// "Constant Interface" antipattern - do not use!
public interface Physics {
  public static final double
      AVOGADROS_NUMBER   = 6.02214199e23;
  public static final double
      BOLTZMANN_CONSTANT = 1.3806503e-23;
  public static final double
      ELECTRON_MASS      = 9.10938188e-31;
}

public class Guacamole implements Physics {
    public static void main(String[] args) {
        double moles = ...;
        double molecules = AVOGADROS_NUMBER * moles;
        ...
    }
}
```

Problems With Constant Interface

- Interface abuse—does not define type
- Implementation detail pollutes exported API
- Confuses clients
- Creates long-term commitment
- Wouldn’t it be nice if compiler let us avoid qualifying names without subtyping?

Solution—Static Import Facility

- Analogous to package import facility
- Imports the static members from a class, rather than the classes from a package
- Can import members individually or collectively
- Not rocket science

Importing Constants With Static Import

```
import static org.iso.Physics.*;

public class Guacamole {
    public static void main(String[] args) {
        double molecules = AVOGADROS_NUMBER * moles;
        ...
    }
}
```

`org.iso.Physics` now a class, not an interface

Can Import Methods as Well as Fields

- Useful for mathematics
- Instead of: `x = Math.cos(Math.PI * theta);`
- Say: `x = cos(PI * theta);`

Static Import Interacts Well With Enums

```
import static gov.treas.Coin.*;

class MyClass {
    public static void main(String[] args) {
        int twoBits = 2 * QUARTER.value();
        ...
    }
}
```

VII. Metadata (Annotations)

- Many APIs require a fair amount of boilerplate
  - Example: JAX-RPC web service requires paired interface and implementation
- Wouldn't it be nice if language let you annotate code so that tool could generate boilerplate?
- Many APIs require "side files" to be maintained
  - Example: bean has `BeanInfo` class
- Wouldn't it be nice if language let you annotate code so that tools could generate side files?

JAX-RPC Web Service—Today

```
public interface CoffeeOrderIF extends java.rmi.Remote {
    public Coffee [] getPriceList()
        throws java.rmi.RemoteException;
    public String orderCoffee(String name, int quantity)
        throws java.rmi.RemoteException;
}

public class CoffeeOrderImpl implements CoffeeOrderIF {
    public Coffee [] getPriceList() {
        ...
     }
    public String orderCoffee(String name, int quantity) {
        ...
    }
}
```

JAX-RPC Web Service With Metadata

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

public class CoffeeOrder {
    @Remote public Coffee [] getPriceList() {
        ...
    }
    @Remote public String orderCoffee(String name, int quantity) {
        ...
    }
}
```

Would You Like to Try it Today?

- All features available in early access compiler
  - http//developer.java.sun.com/developer/earlyAccess/adding_generics
- For documentation, see JSRs 14, 201, 175
  - http://www.jcp.org
- Try it out and send us feeback!

Conclusion

- Language has always occupied a sweet spot
  - But certain omissions were annoying
- In “Tiger” we intend rectify these omissions
- New features were designed to interact well
- Language will be more expressive
  - Programs will be clearer, shorter, safer
- We will not sacrifice compatibility

Joshua.Bloch@sun.com
Neal.Gafter@sun.com

Sun microsystems
We make the net work.