



JDK5.0:Language New Features

陈金洲 (Michael Chen) <http://webuc.net/mechiland>

Opened on Jan 12th,2005 Draft 1 released on Jan 15th,2005

I hold this document as my own copyright declaration. spread it for non-profit is allowed but contact me if you need a business license.



Synopsis

- Generics
- Enhanced **for** Loop
- Autoboxing/Unboxing
- Typesafe Enums
- Varargs
- Static Import
- Annotations
- Reasons upgrading to JDK5.0



Generics





Generics

- Problem: When take an element out of a Collection, you have to cast its type. It's inconvenient, unsafe.
`Cat cat = (Cat)case.get(1);`
- Solution: Generics provides a way that communicate the type of a collection to the compiler.
- Benefit: The code is more clearer and safer.



Generics Sample

```
// 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();
}
```

```
// Removes the 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();
    }
}
```



Generics Benefits

- The code is more clearer, safer
- More importantly, move apart of the specification of a method from the comment or ABC documents to its signature.
 - Report instance: returns all [],not list.
- The program will not throw ClassCastException.
- Improve readability and robustness especially in large program.
- You should use generics everywhere if you can



Generics Usage

- Collections
- Holder classes, such as `WeakReference` and `ThreadLocal`:
 - `java.lang.ref.WeakReference<T>`
 - `java.lang.ThreadLocal<T>`
 - `java.lang.Class<T>`

```
<T extends Annotation> T getAnnotation(Class<T> annotationType);  
Author a = Othello.class.getAnnotation(Author.class);
```



Inside Generics

- Implemented by *type erasure*:
 - present only at compile time
 - erased by compiler after.
 - provides interoperability between generic code and legacy code.
- It's like C++ template, but totally different – not generate a new class nor permit “template metaprogramming”



Enhanced for loop



Enhanced **for** Loop

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

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



Find the error:

```
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(), i.next()));
```

```
// Fixed, though 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()));
}
```



An enhanced way

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

You can use it in arrays also:

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



Summary for Enhanced for loop

- Generally, It's more flexible.
- Use for-each loop any time you can



Autoboxing/unboxing



Autoboxing/Unboxing

- You cannot put an int (or other primitive value) into a collection
- It's not possible to calculate the sum between an int and an Integer directly.



Autoboxing demo

```
import java.util.*;
// Prints a frequency table of the words on the 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 to do the watusi
{be=1, do=1, if=1, is=2, it=2, me=1, the=1, to=3, up=1, watusi=1}
```



Boxing/unboxing hints

- Ignore the distinction between `int` and `Integer`
- If `autounbox` a null `Integer`, a `NullPointerException` will be thrown.
- `==` operator equals be the same.
- There are performance costs with boxing/unboxing.
- Do not use them for scientific computing or other performance-sensitive code.



Typesafe enums



Typesafe Enums

- You must remember this:

// int Enum Pattern – has severe problems!

```
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;
```

- It has many problems.



Problems of int enum pattern

- Not typesafe: Since a season is just an int.
- No namespace: should provide prefix such as SEASON_
- Brittleness: they are compile time constants.
- Printed values are uninformative.



Solutions by Effective Java, #21

- Provide a class like below:

```
Public class Season implements Comparable{
    private Season(String name) {this.name = name;}
    public static final Season SPRING = new Season("spring");
    public static final Season SUMMER = new Season("summer");
    public static final Season FALL = new Season("fall");
    public static final Season WINTER = new Season("winter");
    // other issues...
}
```

- Problem: quite verbose, hence error prone, cannot be used in switch statement.

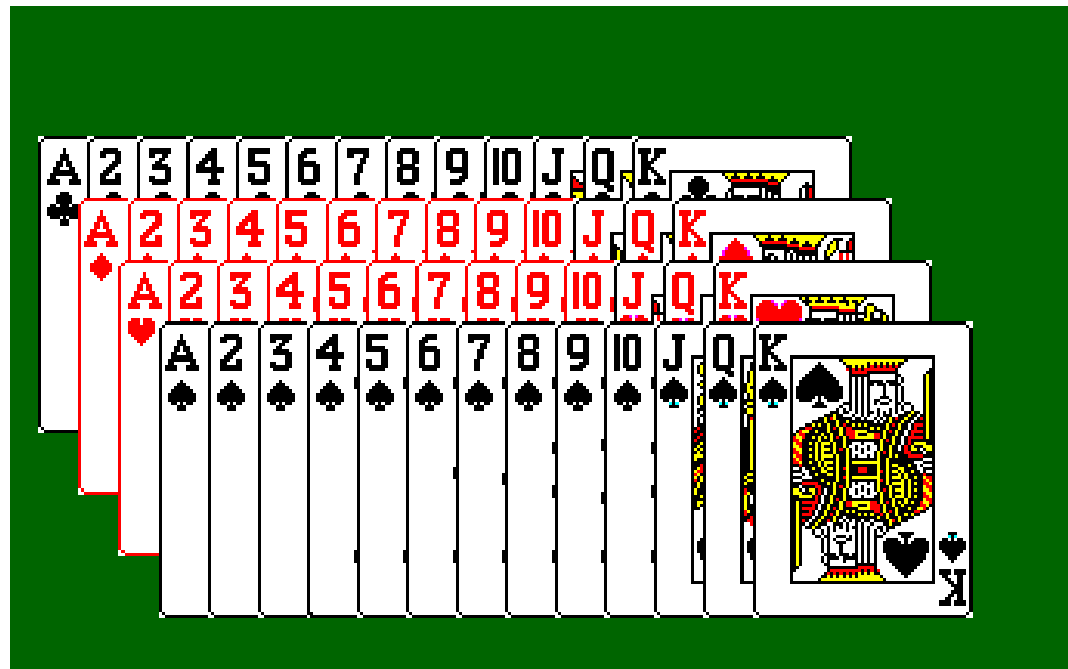


Enum In JDK5.0

- Looks like C, C++ and C#:
`enum Season { WINTER, SPRING, SUMMER, FALL }`
- It's far more powerful.
 - Defines a full-fledged class
 - Solve all the problems above
 - Can add arbitrary methods and fields
 - Can implement arbitrary interfaces.
 - Provide high-quality impl. Of Object.
 - Comparable and Serializable.
 - Support switch

Enum Sample #1: Card

- DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE
- CLUBS, DIAMONDS, HEARTS, SPADES



Enum Sample#1

```
import java.util.*;
public class Card {
    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; }
    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);
    }
}
```

The toString facility

Enum.values()

Enum Sample#1

```
import java.util.*;
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;
    }
}
```

```
$ java Deal 4 5
```

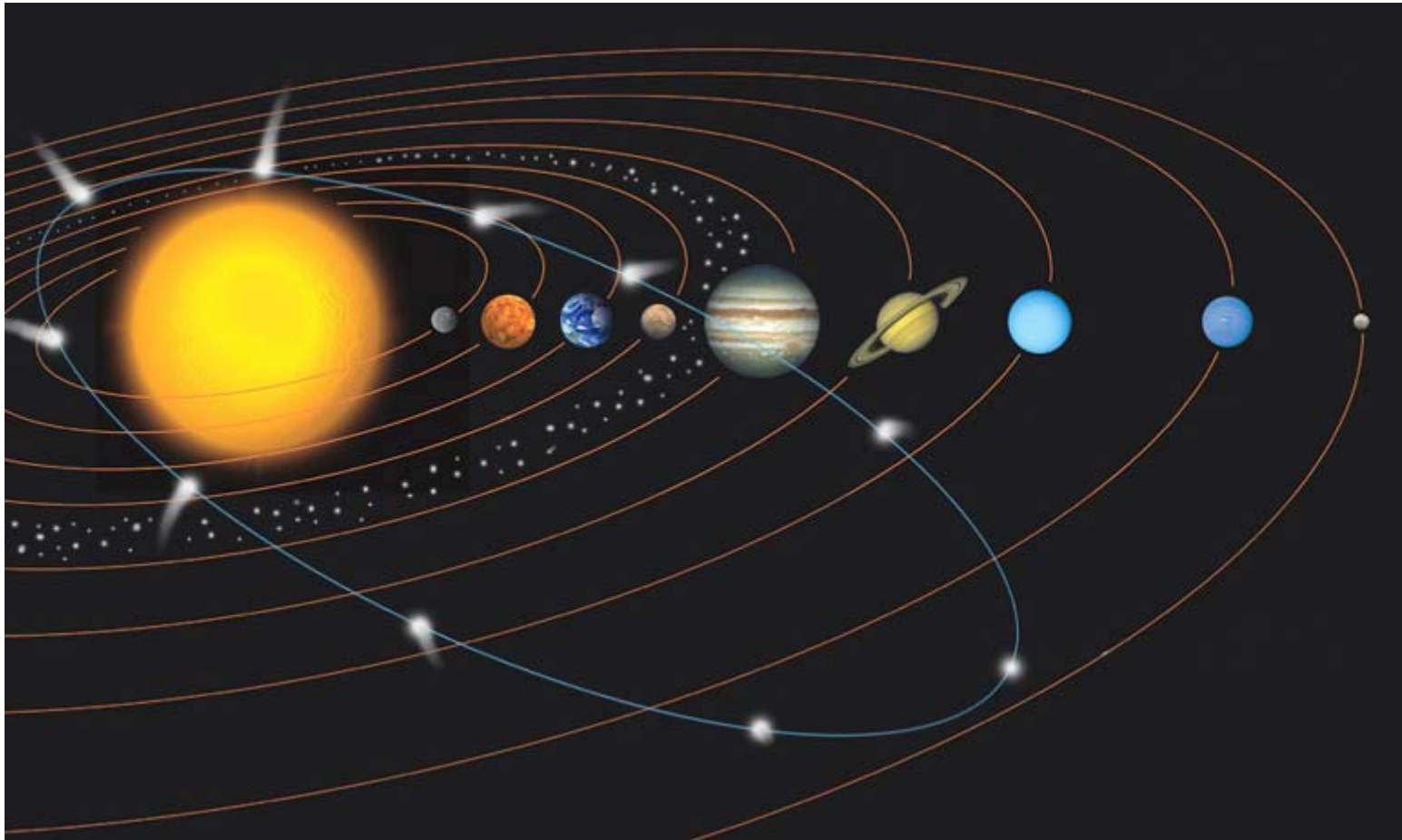
```
[FOUR of HEARTS, NINE of DIAMONDS, QUEEN of SPADES, ACE of SPADES, NINE of SPADES]
[DEUCE of HEARTS, EIGHT of SPADES, JACK of DIAMONDS, TEN of CLUBS, SEVEN of SPADES]
[FIVE of HEARTS, FOUR of DIAMONDS, SIX of DIAMONDS, NINE of CLUBS, JACK of CLUBS]
[SEVEN of HEARTS, SIX of CLUBS, DEUCE of DIAMONDS, THREE of SPADES, EIGHT of CLUBS]
```



Behavior for Enums?

- It's possible to add behavior and data to the enums.
- Below will show a demo.

Enum Sample2: SolarSystem



Enum Sample#2

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6),
    EARTH (5.976e+24, 6.37814e6),
    MARS (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27, 7.1492e7),
    SATURN (5.688e+26, 6.0268e7),
    URANUS (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO (1.27e+22, 1.137e6);

    private final double mass; // in kilograms
    private final double radius; // in meters

    Planet(double mass, double radius) {
        this.mass = mass; this.radius = radius; }
    private double mass() { return mass; }
    private double radius() { return radius; }
    // universal gravitational constant (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;
    double surfaceGravity() { return G * mass / (radius * radius); }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity(); }
}
```

```
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 72  
Your weight on MERCURY is 27.198548  
Your weight on VENUS is 65.159935  
Your weight on EARTH is 72.000000  
Your weight on MARS is 27.269077  
Your weight on JUPITER is 182.200142  
Your weight on SATURN is 76.753119  
Your weight on URANUS is 65.169158  
Your weight on NEPTUNE is 81.959621  
Your weight on PLUTO is 4.814961
```



More about behavior

- Can be go further: each enum type has its own implementation.
- Think about the Operand $+$, $-$, $*$, $/$

Enum Sample#3

```
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);
    }
}
```

```
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);
}
```



Enum sample #3 result

```
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
```



java.util.EnumSet, EnumMap

- Code:

```
enum Day {SUNDAY, MONDAY, TUESDAY,  
WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}
```

- EnumSet.range:

```
for (Day d: EnumSet.range(Day.MONDAY, Day.FRIDAY))  
    System.out.println(d);
```

- EnumSet.of:

```
EnumSet.of(Day.SUNDAY, Day.MONDAY);
```

- Enum.ordinal

- Same as EnumMap



Summery of Enum

- Use it Any time you need a fixed set of constants.
- Not necessary for the constants stay fixed for all time, such as PI
- Designed to allow for binary compatible evolution of enum types.



Varargs



Varargs (Variant arguments)

- As MessageFormat, you should always provide an array:

```
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);
```

- The new declaration:

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

usage:

```
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");
```

- Try the new System.out.printf(...); (you may remember the old C/C++...)



Summery for Varargs

- As a client, take advantage of them whenever the API offers them. Important use: reflection, message format, printf.
- As an api designer, use it sparingly only when the benefit is truly compelling.
- In general, should not overload a varargs method, or it will be difficult for programmers to figure out which overloading gets called.



Static Import



Static Import

- In order to access static members, you should use such as `Math.PI`, `Math.cos()`
- `import static java.lang.Math.PI`, or
- `import static java.lang.Math.*`
- Usage like this:
`double r = cos(PI * theta);`



Summary for static import

- Use it very sparingly!!!
- Use it ONLY when you have visit a static constant very frequently.
- Otherwise, it's will be Unreadable, difficult to maintain.
- No IDE support function static import yet.
(till Eclipse 3.1M4)




Annotations



Annotations

- Many API require boilerplate code,
 - JAX-RPC web service, you need a paired interface and implementation.
- Some api require “side files” in parallel with programs.
 - JavaBean, the BeanInfo class
 - EJB, require a deployment descriptor(ejb-jar.xml)
- Annotations in code will be more convenient and less error-prone.



Annotations: already had

- Java platform has various ad hoc annotation mechanisms
 - transient modifier
 - @deprecated tag
- Xdoclet: attribute oriented programming:
 - EJB, Hibernate, Struts, ...usage.



Annotations Cont.

- Do not directly affect program semantics
- Affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program.
- Can be read from source files, class files, or reflectively at run time.



Define an annotation type

- Similar to normal interface, with a “@” preceded.
- Each method declaration defines an element of the annotation type, must not have any parameters or throw clause.
- Return types: `primitives`, `String`, `Class`, `enums`, `annotations`, `arrays` of the above types.
- Method can have default values.



An example anno. type

```
/**
 * Describes the Request-For-Enhancement(RFE) that led
 * to the presence of the annotated API element.
 */
public @interface RequestForEnhancement {
    int id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date();
    default "[unimplemented]";
}
```



Anno. Usage example

```
@RequestForEnhancement(  
    id = 2868724,  
    synopsis = "Enable time-travel",  
    engineer = "Mr. Peabody",  
    date = "4/1/3007"  
)  
public static void travelThroughTime(Date destination) { ... }
```



Annotation type

- **No elements : marker**

```
public @interface Test { }
```

```
@Test public void method1() {...}
```

- **One single element: should be value**

```
public @interface Copyright { String value(); }
```

```
@Copyright("2005 Some copyright declarations.")  
public class HelloWorld{ ... }
```



Test framework based on annotation

```
import java.lang.annotation.*;  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface Test { }
```

- It's marker.
- Itself annotated (meta-annotations)
- First: to be retained by the VM
- Second: can be used to only method.



Sample program

```
public class Foo {  
    @Test public static void m1() { }  
    public static void m2() { }  
    @Test public static void m3() {  
        throw new RuntimeException("Boom");  
    }  
    public static void m4() { }  
    @Test public static void m5() { }  
    public static void m6() { }  
    @Test public static void m7() {  
        throw new RuntimeException("Crash");  
    }  
    public static void m8() { } }  
}
```




Testing tool

```
import java.lang.reflect.*;
public class RunTests {
    public static void main(String[] args) throws Exception {
        int passed = 0, failed = 0;
        for (Method m : Class.forName(args[0]).getMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                try { m.invoke(null); passed++;
                } catch (Throwable ex) {
                    System.out.printf("Test %s failed: %s %n", m, ex.getCause());
                    failed++;
                }
            }
        }
        System.out.printf("Passed: %d, Failed %d%n", passed, failed);
    }
}
```



Run result

```
$ java RunTests Foo  
Test public static void Foo.m3() failed: java.lang.RuntimeException: Boom  
Test public static void Foo.m7() failed: java.lang.RuntimeException: Crash  
Passed: 2, Failed 2
```




Annotation usage: EJB3(1) (from jboss impl.)

```
import javax.ejb.Stateless;  
@Stateless public class CalculatorBean implements  
    CalculatorRemote, CalculatorLocal { ... }
```

```
import javax.ejb.Remote;  
@Remote public interface CalculatorRemote extends Calculator { }
```

- No ejb-jar.xml to maintain.
- My opinion: can be go further such as:
@Stateless (remote=true, local=true)
public class CalculatorBean ...



EJB3(2) injection

```
@Stateful public class ShoppingCartBean implements ShoppingCart {
    private HashMap<String, Integer> cart = new HashMap<String, Integer>();

    @Inject(jndiName = "org.jboss.tutorial.injection.bean.Calculator")
    private Calculator calculator;
    private Calculator set;

    @EJB(name="calculator", jndiName = "org.jboss.tutorial.injection.bean.Calculator")
    public void setCalculator(Calculator c) { set = c; }

    public void buy(String product, int quantity) {
        if (cart.containsKey(product)) {
            int currq = cart.get(product);
            currq = calculator.add(currq, quantity);
            cart.put(product, currq); } else { cart.put(product, quantity); } }

    public HashMap<String, Integer> getCartContents() { return cart; }
}
```

- The same design appeared in Tapestry 3.1 Hivemind integration
- Spring will be over?



EJB3(3) entity bean

```
@Entity  
@Table(name = "PURCHASE_ORDER")  
public class Order implements java.io.Serializable {...}
```

```
@Id(generate = GenerationType.AUTO)  
public int getId() { return id; }
```

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)  
@JoinColumn(name = "order_id")  
public Collection<LineItem> getLineItems() { return lineItems; }
```

Annotations make EJB more easier.



Summery for Annotations

- Reduce the maintain(always dirty, and error prone) jobs.
- Design proper annotation types it if you are SA, use the types if you are client.
- Note: define a clear document to describe the runtime behavior. Otherwise there may be puzzles.



JDK5.0 in general

- In general, they are all compiler “tricks”.
- To byte-code compatible for the prior versions.
- reduce the chance of error occurrences.
- More clear, flexible, simple, and easier.
- Use it if there is a 5.0 compatible J2EE container, or can use 5.0 on desktop
- Ready for the JDK5.0.

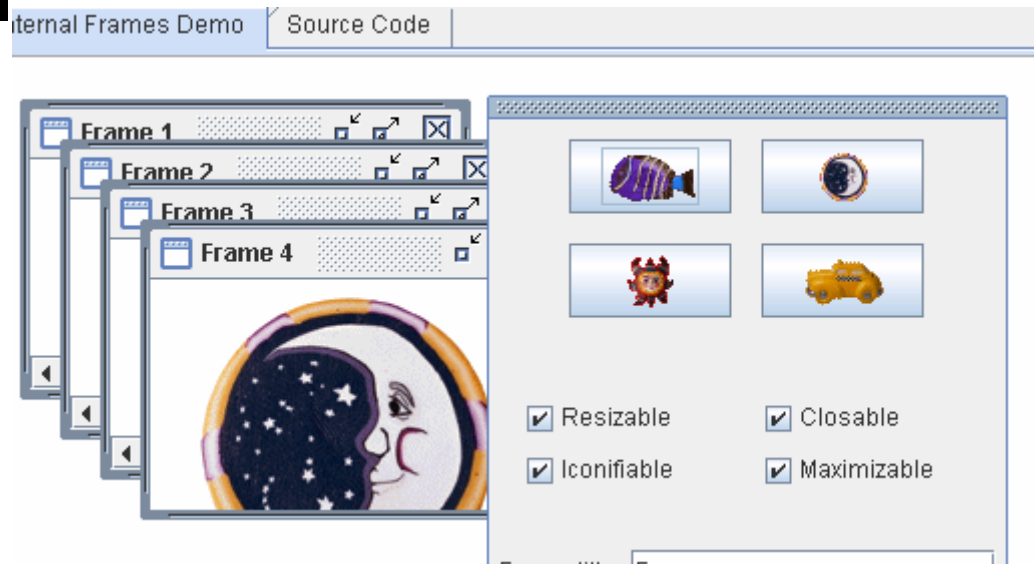


Appendix: 4 reasons upgrading to jdk5.0 (other features from sun)

- You application already works on 5.0
- Faster
- Reduced development time.
- Ready for mission-critical systems

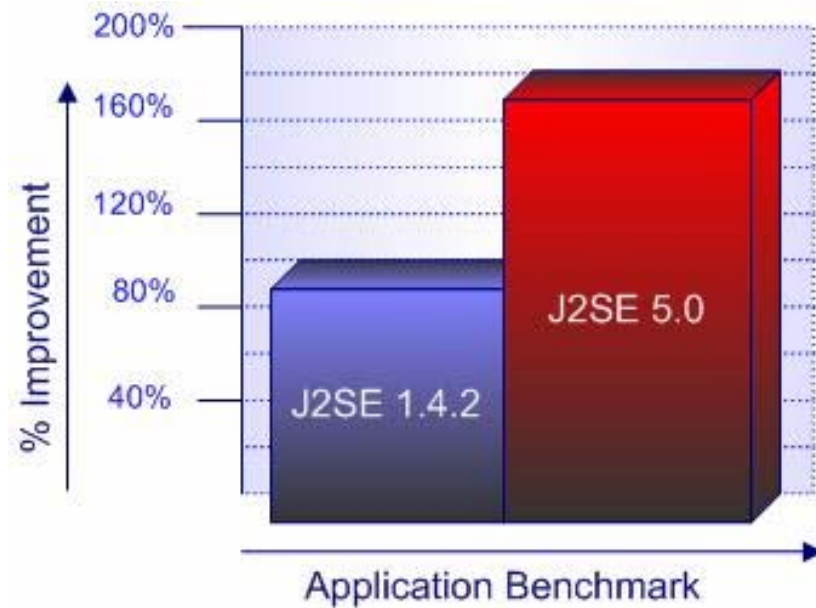
Your application already works on 5.0

- Former apps can run on 5.0
- Improved performance
- Monitoring and manageability
- New look and feel



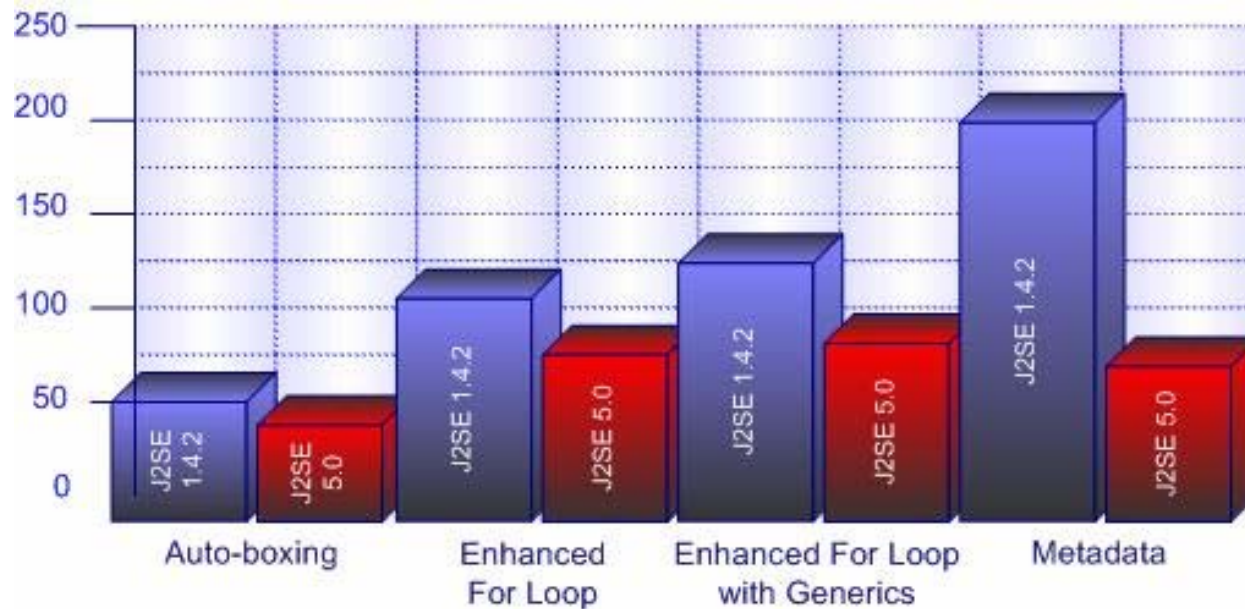
It is faster

- Reduced startup time
- 64-bit performance
- the JVM is now self-configuring and self-tuning on server-class machines.



Reduced development time

- Reduce development time
- Improve error checking savings.





Ready for mission-critical system

- Scalability on both client and server
- Quality: been test on various system and platforms.
- Deployment: can be monitoring, profiling more easily.



Thanks!