Seminar
"Selected Topics in Software Engineering"

What's new in Java 1.5?

Christopher Oezbek
Freie Universität Berlin, Department of CS
http://www.inf.fu-berlin.de/inst/ag-se/

Slides adapted from David Matuszek
http://www.cis.upenn.edu/~matuszek/cit591-2004/

Versions of Java

- Oak: Designed for embedded devices
- Java: Original, not very good version (but it had applets)
- Java 1.1: Adds inner classes and a completely new event-handling model

Java 1

- Java 1.2: Includes "Swing" but no new syntax
- Java 1.3: Additional methods and packages, but no new syntax
- Java 1.4: More additions and the assert statement

Java 2

- Java 1.5: Generics, enums, new for loop, and other new syntax

Java 5.0

- Watch out: It's called J2SE 5.0 not J5SE! :-)

Reason for changes

- "The new language features all have one thing in common: they take some common idiom and provide linguistic support for it. In other words, they shift the responsibility for writing the boilerplate code from the programmer to the compiler."

  --Joshua Bloch, senior staff engineer, Sun Microsystems

- In other words:
  - Wherever people were complaining too much that Java is too verbose, they made some adjustments.

New features

- Generics
  - Compile-time type safety for collections without casting
- Enhanced for loop
  - Syntactic sugar to support the Iterator interface
- Autoboxing/unboxing
  - Automatic wrapping and unwrapping of primitives
- Typesafe enums
  - Provides all the well-known benefits of the Typesafe Enum pattern
- Static import
  - Lets you avoid qualifying static members with class names
- Scanner and Formatter
  - Finally, simplified input and formatted output

Generics

- A generic is a method that is recompiled with different types as the need arises
- The bad news:
  - Instead of saying: List words = new ArrayList();
  - You'll have to say:
    List<String> words = new ArrayList<String>();
- The good news:
  - Replaces runtime type checks with compile-time checks
  - No casting; instead of
    String title = (String) words.get(i);
    you use
    String title = words.get(i);
- Some classes and interfaces that have been "genericized" are: Vector, ArrayList, LinkedList, Hashtable, HashMap, Stack, Queue, PriorityQueue, Dictionary, TreeMap and TreeSet

Generic Iterators

- To iterate over generic collections, it's a good idea to use a generic iterator

```
List<String> listOfStrings = new LinkedList<String>();
...
for (Iterator<String> i = listOfStrings.iterator(); i.hasNext(); ) {
    String s = i.next();
    System.out.println(s);
}
```

Writing generic methods

- ```
  private void printListOfStrings(List<String> list) {
      for (Iterator<String> i = list.iterator(); i.hasNext(); ) {
          System.out.println(i.next());
      }
  }
  ```

- This method *should* be called with a parameter of type List<String>, but it *can* be called with a parameter of type List
  - The disadvantage is that the compiler won't catch errors; instead, errors will cause a ClassCastException
  - This is necessary for backward compatibility
  - Similarly, the Iterator need not be an Iterator<String>
  ```

Type wildcards

- Here's a simple (no generics) method to print out any list:
  - ```
    private void printList(List list) {
        for (Iterator i = list.iterator(); i.hasNext(); ) {
            System.out.println(i.next());
        }
    }
    ```
- The above still works in Java 1.5, but now it generates warning messages
  - Java 1.5 incorporates lint (like C lint) to look for possible problems
- You should eliminate *all* errors and warnings in your final code, so you need to *tell* Java that any type is acceptable:
  - ```
    private void printListOfStrings(List<?> list) {
        for (Iterator<?> i = list.iterator(); i.hasNext(); ) {
            System.out.println(i.next());
        }
    }
    ```

Writing your own generic types

```
public class Box<T> {
    private List<T> contents;

    public Box() {
        contents = new ArrayList<T>();
    }

    public void add(T thing) { contents.add(thing); }

    public T grab() {
        if (contents.size() > 0) return contents.remove(0);
        else return null;
}
```

- Sun's recommendation is to use single capital letters (such as T) for types

Using Generics for more than Collections

- In C++ we can use Templates to do the following:

```
public class Communicate  {
      public <T> void speak(T speaker) {  speaker.talk(); }
      }
}
```

- This gets statically type-checked on use!

```
Communicate c = new Communicate();
c.speak(new Human());
c.speak(new Stone()); // -> compile time error!
```

- This latent typing does not work in Java. Solution:

```
interface Speaks { void speak(); }
public class Communicate  {
      public <T extends Speaks> void speak(T speaker) {
  speaker.speak(); }
}
```

New for-statement

- The syntax of the new statement is
  for(***type var*** : ***array***) {…}
  or for(***type var*** : ***collection***) {…}
- Example:

```
for(float x : myRealArray)
  myRealSum += x;
```

- For a collection class that has an Iterator, instead of

```
TimerTask tTask;
for (Iterator iter = c.iterator(); iter.hasNext(); tTask = (TimerTask)iter.next())
    tTask.cancel();
```

  you can now say

```
for (TimerTask task : c)
    task.cancel();
```

- Watch out: The for-Statement is not null-safe!

```
Collection<TimerTask> c = null;
for (TimerTask task : c)
    task.cancel();
```

Use the for-statement in your own classes

- Just implement Iterable<T> and the class is ready to go

```
class ForEachInteger implements Iterable<Integer> {
        public Iterator<Integer> iterator() { /*...*/ }
}

/*...*/

ForEachInteger forEach = new ForEachInteger();
for (Integer i : forEach){ /*...*/}
```

Auto boxing and unboxing

- Java won't let you use a primitive value where an object is required – you need a "wrapper"
  - myVector.add(new Integer(5));
- Similarly, you can't use an object where a primitive is required – you need to "unwrap" it
  - int n = ((Integer)myVector.lastElement()).intValue();
- Java 1.5 makes this automaticly:
  - Vector<Integer> myVector = new Vector<Integer>();
    myVector.add(5);
    int n = myVector.lastElement();
- Other extensions make this as transparent as possible
  - For example, control statements that previously required a boolean (if, while, do-while) can now take a Boolean

Auto-Boxing subtleties

- Auto-Boxing works also in comparisons (==, <, >, etc.)
- Watch out: No un-boxing for two reference-types!
- Literal cache for ints in the short range (-128,127)...

```
Integer i = 0; Integer j = 0;
System.out.println(i == j); // true
Integer i = 127; Integer j = 127;
System.out.println(i == j); // true
Integer i = 128; Integer j = 128;
System.out.println(i == j); // false
Integer i = new Integer(0); Integer j = new Integer(0);
System.out.println(i == j); // false
```

- ...and Booleans (more?)
- Conclusion: Don't compare two "Boxed"-instances for value equality => use equals

Auto-Boxing limitations

- Run-time overhead! Don't use for scientific computations and your genomic database.

```
for (Integer i = 0; i < Integer.MAX_VALUE ; i++){ }
```

  - Around 4x slower than with int.
- Primitives don't become objects in all situations:

```
Integer i = 3;
i.compareTo(2); // okay
3.compareTo(2); // This doesn't work!
```

- Remember that this works with strings:

```
"a String".length() // -> 8
```

- Watch out for null! Not a compile time error any more.

```
Integer i = null;
if (i == 0) // -> NullPointerException
```

Enumerations

- An enumeration, or "enum", is simply a set of constants to represent various values
- Here's the old way of doing it
  - public final int SPRING = 0;
    public final int SUMMER = 1;
    public final int FALL = 2;
    public final int WINTER = 3
- This is a nuisance, and is error prone as well
- Here's the new way of doing it:
  - enum Season { WINTER, SPRING, SUMMER, FALL }

Advantages of the new enum

- Enums provide compile-time type safety
  - int enums don't provide any type safety at all: season = 43;
- Enums provide a proper name space for the enumerated type
  - With int enums you have to prefix the constants (for example, seasonWINTER or S_WINTER) to get anything like a name space
- Enums are robust
  - If you add, remove, or reorder constants, you don't need to recompile clients that use your enum
  - Enum printed values are informative ("SPRING", "WINTER")
  - If you print an int enum, you just see a number
- Because enums have objects, you can put them in collections
- Because enums are classes, you can add fields and methods
- Inside the implement the singleton pattern (their constructor is private) *for each* value (there is only one SUMMER).

enums are classes

- An enum is actually a new type of class
- You can declare them as inner classes or outer classes
- You can declare variables of an enum type and get type safety and compile time checking
  - Each declared value is an instance of the enum class
  - Enums are implicitly public, static, and final
  - You can compare enums with either equals or ==
- enums extend java.lang.Enum and implement java.lang.Comparable
  - Hence, enums can be sorted
- Enums override toString() and provide valueOf()
- Example:
  - `Season season = Season.WINTER;`
  - `System.out.println(season );  // prints WINTER`
  - `season = Season.valueOf("SPRING"); // sets season to Season.SPRING`

Enums *really are* classes

```
public enum Coin {
    // enums can have instance variables
    private final int value;
    // An enum can have a constructor, but it isn't public
    Coin(int value) { this.value = value; }
    // Each enum value you list really calls a constructor
    PENNY(1), NICKEL(5), DIME(10),
    // You can even override methods on the value level
    QUARTER(25) { String toString(){ return "1/4" } };
    // And, of course, classes can have methods
    public int value() { return value; }
}
```

- Well okay, they are not completely classes. You cannot sub-class them because they are final...

Other features of enums

- values() returns an array of enum values
  - Season[] seasonValues = Season.values();
- switch statements can now work with enums
  - switch (thisSeason) { case SUMMER: …; default: …}
  - You *must* say case SUMMER:, *not* case Season.SUMMER:
  - It's still a very good idea to include a default case
- It is possible to define value-specific class bodies, so that each value has its own methods (see last example)

varargs

- You can create methods and constructors that take a variable number of arguments
  - public void foo(int count, String**...** cards) { ***body*** }
  - The "**...**" means *zero or more* arguments (here, zero or more Strings)
    - If zero arguments are passed, cards is an empty String[]-instance and not null.
  - Call with foo(13, "ace", "deuce", "trey");
  - Only the *last* argument can be a vararg
  - To iterate over the variable arguments, use the new for loop:
    for (String card : cards) { ***loop body*** }

Static import facility

- import static org.iso.Physics.*;

  …
  double molecules = AVOGADROS_NUMBER * moles;
- You no longer have to say Physics.AVOGADROS_NUMBER
- Are you tired of typing System.out.println(***something***); ?
- Do this instead:
  - import static java.lang.System.out;
  - out.println(***something***);
- Works with static constants, functions and (inner-)classes.

java.util.Scanner

- Finally, Java has a fairly simple way to read input
  - **Scanner sc = Scanner.create(System.in);**
  - boolean b = sc.nextBoolean();
  - byte by = sc.nextByte();
  - short sh = sc.nextShort();
  - int i = sc.nextInt();
  - long l = sc.nextLong();
  - float f = sc.nextFloat();
  - double d = sc.nextDouble();
  - String s = sc.nextLine();
- By default, whitespace acts as a delimiter, but you can define other delimiters with regular expressions

java.util.Formatter

- Java now has a way to produce formatted output, based on the C printf statement

- ```
  String line;
  int i = 1;
  while ((line = reader.readLine()) != null) {
      System.out.printf("Line %d: %s\n", i++, line);
  }
  ```

- There are about 45 different format specifiers (such as %d and %s), most of them for dates and times
  ```

New methods in java.util.Arrays

- Java now has convenient methods for printing arrays:
  - Arrays.toString(***myArray***) for 1-dimensional arrays
  - Arrays.deepToString(***myArray***) for multidimensional arrays
- Java now has convenient methods for comparing arrays:
  - Arrays.equals(***myArray***, ***myOtherArray***) for 1-dimensional arrays
  - Arrays.deepEquals(***myArray***, ***myOtherArray***) for multidimensional arrays
- It is important to note that these methods *do not* override the public String toString() and public boolean equals(Object) instance methods inherited from Object
  - The new methods are static methods of the java.util.Arrays class

Annotations

- Allow you to mark any program elements (methods, constructors, fields, local variables, packages, types and parameters).
  - For instance you can annotate methods as overridden, or deprecated, or to turn off compiler warnings for them.
- Especially useful to get rid of boilerplate code like J2EE deployment descriptors
- An answer to the popularity of XDoclet with added typing
- Annotations are defined using the @interface keyword:

```
public @interface RequestForEnhancement {
    int    id();
    String synopsis();
    String engineer() default "[unassigned]";
}
```

Annotations

- Annotations are put in front of declarations like public or static keywords:

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

- The method travelThroughTime is now annotated. Since annotations are accessible using reflection you can now programmatically find all methods on which "Mr. Peabody" was the change engineer.
- Sun provides the annotation processing tool apt to deal with annotations for "take over the world"-things.

An EJB Example

- Remember EJB 2.1?
- To write a EJB you typically need at least two interfaces (home and remote), one implementation class with several empty method implementations, and a deployment descriptor.
- With EJB 3.0 and "heavy" use of annotation:

```
import javax.ejb.*;
@Stateless @Remote public class HelloWorldBean {
   public String sayHello() {
      return "Hello World!!!";
   }
}
```

Annotations - Conclusion

- Sun hopes that people will use annotations to store information about program code (hence the name) and write program transformators based on that.
- Improvements could be expected for IDEs (for instance for GUI-generation), frameworks (like the previous J2EE example) and higher level coding tools (like AspectJ).
- As a meta-data-format (for instance for issue tagging) I believe it might be possible that we will see some usage in the programs of the average programmer, but for everything higher the technology seems to complicated.
  - But people are dreaming of meta-programming-systems again, so beware.
- With annotations Java has done a good job at catching up with C++ in the category of hacker features.

Additional features

- Threading
  - There are many new features for controlling synchronization and threading which increase performance and prevent that you have to write synchronization primitives yourself.
- Monitoring and Profiling API
  - It is now easier to observe what's going on inside the JVM
- Unicode 4.0
- RMI
  - **rmic** is no longer needed to generate stubs/skeletons.
- New skin for Swing

Conclusion

- Java 1.5 was released in September 2004
- Eclipse support with 3.1.0 due around Feb 2005
  - Get a 3.1.0 Release Candidate as >M4, if you cannot wait
- I've just touched on the new features...
  - ...and had a good slide-set to work from.
  - Don't expect that there are not some more gems hidden.
- Most of the Java 1.5 additions are designed for ease of use.
- If you know Java, you will learn them easily.
- For new learners the perspective might be different.
- I believe that Java has gained some usable features that make it a more well rounded language.
- If you are on a java project right now, switching will make you happy! =)

Resources

- David Matuszek - Initial set of slides http://www.cis.upenn.edu/~matuszek/cit591-2004
- Linda DeMichiel - EJB 3.0 and annotations http://www.theserverside.com/talks/videos/Symposium2004/EJBWorkInProgress/dsl/interview.html
- Anil Sharma - EJB 3.0 in a nutshell http://www.javaworld.com/javaworld/jw-08-2004/jw-0809-ejb.html
- Robert Schuster - Useful remarks on Autoboxing and Static Imports

Questions?

Thank you!