Java 5: New Features
(t.k.prasad@wright.edu)

Adapted from Material by
- David Matuszek, UPenn
- James Heliotis, RIT
- ...

More references:
- http://java.sun.com/developer/technicalArticles/releases/j2se15/

Versions of Java

Oak (1991): Designed for embedded devices
Java (1995): Original version (had applets)
Java 1.1 (1997): Adds inner classes and a completely new event-handling model
Java 1.2 (1998): Includes “Swing” but no new syntax
Java 1.3 (2000): Additional methods and packages, but no new syntax
Java 1.5 (2004): Generics, enums, new for loop, and other new syntax

Java 1.0
8 packages
212 classes
Java 1.1
23 packages
504 classes
Java 1.2
58 packages
1050 classes
Java 1.3
77 packages
1065 classes
Java 1.4
103 packages
2179 classes
Java 1.5
131 packages
2656 classes

New Events
- Inner class
- Object
- Serialization
- Jar Files
- International
- Reflection
- JDBC
- RMI

Java 1.0
- JFC/Swing
- Drag and Drop
- Java2D
- CORBA

Java 1.1
- JNDI
- Java Sound
- Timer

Java 1.2
- Regular Exp
- Logging
- Assertions
- NIO

Java 1.3
- java.nio, java.imego, java.net, java.print,
- java.security, org.w3c

Java 1.4
- java.naming, java.sound,
- java.transaction

Java 1.5
- java.accessibility, java.swing, org.omg

Java.math, java.rmi, java.security, java.sql, java.text, java.beans
java.applet, java.awt, java.io, java.lang, java.net, java.util
**Assertion facility**

- An assert statement has two permissible forms:
  - `assert expression1;`
  - `assert expression1 : expression2;`

- In each form, `expression1` is the boolean-typed expression being asserted. The expression represents a program condition that the developer specifically proclaims must be true during program execution. In the second form, `expression2` provides a means of passing a String message to the assertion facility.

```java
assert ref != null;
assert 0 < value : "Value must be non-negative: " + value;
assert ref.m1(parm) && count == (oldCount + 1);
```

**Informal Behavior**

- Evaluate `expression1`
  - If true
    - No further action
  - If false
    - And if `expression2` exists
      - Evaluate `expression2` and use the result in a single-parameter form of the AssertionError constructor
    - Else
      - Use the default AssertionError constructor

```java
public class Foo {
    public void m1( int value ) {
        assert 0 <= value;
        System.out.println( "OK" );
    }
    public static void main( String[] args ) {
        Foo foo = new Foo();
        System.out.print( "foo.m1(  1 ): " );
        foo.m1( 1 );
        System.out.print( "foo.m1( -1 ): " );
        foo.m1( -1 );
    }
}
```

```bash
$ java -ea Foo
foo.m1(  1 ): OK
foo.m1( -1 ): Exception in thread "main" java.lang.AssertionError
    at Foo.m1(Foo.java:5)
    at Foo.m1(Foo.java:15)
```

```java
public class Bar {
    public void m1( int value ) {
        assert 0 <= value ; "Value must be non-negative: " + value;
        System.out.println( "OK" );
    }
    public static void main( String[] args ) {
        Bar bar = new Bar();
        System.out.print( "bar.m1(  1 ): " );
        bar.m1( 1 );
        System.out.print( "bar.m1( -1 ): " );
        bar.m1( -1 );
    }
}
```

```bash
$ java -ea Bar
bar.m1(  1 ): OK
bar.m1( -1 ): Exception in thread "main" java.lang.AssertionError: Value must be non-negative: -1
    at Bar.m1(Bar.java:5)
    at Bar.m1(Bar.java:15)
```

```java
public class Foo {
    public void m1( int value ) {
        assert 0 <= value :
```
**Reason for changes (to get Java 5)**

- “The new language features (in Java 5) 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

**New features**

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

**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

**New for statement**

- The syntax of the new statement is
  - `for(type var : array) […]`
  - or `for(type var : collection) […]`
- Example:
  ```java
  for(float x : myRealArray) {
      myRealSum += x;
  }
  ```
  - For a collection class that has an Iterator, instead of
    ```java
    for (Iterator iter = c.iterator(); iter.hasNext(); )
    ((TimerTask) iter.next()).cancel();
    ```
  you can now say
    ```java
    for (TimerTask task : c)
        task.cancel();
    ```
Array Example

```java
import java.util.Arrays;
public class ArrayIterator {
    public static void main(String[] args) {
        System.out.print("Arg's lengths are ");
        for (String arg: args) {
            System.out.print(" "+arg.length());
        }
        System.out.println(");
        System.out.println("args.toString() = " + args.toString());
        System.out.println("Array.toString(args) = "+Arrays.toString(args));
    }
}
```

Generics
- A generic is a class that is recompiled with different types as needed
- The bad news:
  - Instead of: List words = new ArrayList();
  - Write: 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);
    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 (String s : listOfStrings) {
        System.out.println(s);
    }
Writing generic methods

- private void printListOfStrings(List<String> list) {
  for (String s : list) {
    System.out.println(s);
  }
}

- 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.

Type wildcards

- Here’s a simple (no generics) method to print out any list:

```java
private void printList(List list) {
  for (Iterator i = list.iterator(); i.hasNext(); ) {
    System.out.println(i.next());
  }
}
```

- The above still works in Java 5, but generates a warning.

- You should eliminate all errors and warnings in your final code, so you need to tell Java that any type is acceptable:

```java
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;
  }
}

Auto boxing and unboxing

- **Box**
  - An instance of a wrapper class that holds a value of a primitive type

- **Boxing**
  - Creating a box for a primitive value

- **Unboxing**
  - Removing the primitive value from a box
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 5 makes this automatic:
- 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
- There are some subtle issues with equality tests, though

Example: Assignment

```java
public class SimpleBoxing {
    public static void main(String[] args) {
        Integer n1 = new Integer(43);
        int i1 = n1;
        int i2 = 57;
        Integer n2 = i2 - 1 * n1 + i1;
        System.out.println(n1 + " " + n2 + " " + i1 + " " + i2);
    }
}
```

```java
Integer x = 6;
Integer y = 2*x;
```

generates the same byte code as

```java
Integer x = Integer.valueOf(6);
Integer y = Integer.valueOf(2 * x.intValue());
```
**Enumerations**

- An enumeration, or “enum,” is simply a set of constants to represent various values.
- Here’s the old way of doing it:
  ```java
  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:
  ```java
  enum Season { WINTER, SPRING, SUMMER, FALL }
  ```

**Enumeration Type Issues**

- Are they just integers?
  - Type safety
  - How are they input and output?
  - Brittleness in the face of changes
- Can they be treated like classes?
  - Methods
  - Extensions
- Name clashes?

**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.
  - Supports equals, “==”, compareTo, ordinal, etc.
  - Enums override `toString()` and provide `valueOf()`, `name()`.
  - Example:
    ```java
    enum Season { WINTER, SPRING, SUMMER, FALL }
    ```

```java
class TrafficLight {
  private enum Color { red, green, yellow };
  private Color light;
  public TrafficLight( String s ) {
    light = Color.valueOf( s );
  }
}
```

```java
public enum Coin {
  penny, nickel, dime, quarter;
  public static void show() {
    System.out.print( "The coins available are\n{" );
    for ( Coin c: Coin.values() )
      System.out.print( " " + c.name() );
    System.out.println( " }\" );
  }
}
```
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 must recompile, and then everything is OK again
- Enum printed values are informative
  - If you print an int enum you just see a number
- Because enums are objects, you can put them in collections
- Because enums are classes, you can add fields and methods

Enums really are classes

```java
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), QUARTER(25);
    // And, of course, classes can have methods
    public int value() { return value; }
    public static void main(String[] args) {
        int sum = 0;
        for (String arg : args)
            sum += Coin.valueOf(arg).value();
        System.out.println(sum + " cents");
    }
}
```

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

Enum Semantics

- Only one copy of each value (instance) of an Enum class is allowed.
  - The operator == works fine.
  - No new instances allowed.
  - No public constructors allowed.
  - No subclasses allowed (messy semantics).
- Each instance must be created.
  - Static initializers cannot refer to the instances.
**Useful Collection Classes**

- Optimized for enumerations:
  - **EnumSet**
    - Uses a bit set implementation
  - **EnumMap**
    - Used to store additional information for each enum literal
    - Often declared inside the enum itself

**Persistence Properties**

- **enums** implement **Serializable**.
- Their serialized form as a byte array does not change even if the list of literals is modified.
- This means that data in a file does not become obsolete, and network peers can be at different revision levels.
  - Robust in the face of modifications

**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)
- Call with foo(13, "eins", "zwei", "drei");
- 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.*;
- class Guacamole {
  - public static void main(String[] args) {
    - 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);
More examples

- Imports static members from a class
- Old Approach
  public interface MyConstant {
  ...
  }
  public class C1 implements MyConstants {
  ...
  }
  - Pollutes the meaning of interface
  - Confuses clients
- New Approach
  import static myPackage.MyConstants.*;
  public class C1 {
  ...
  }

Scanner Class Overview

- Introduced to write a quick-and-dirty Java program that uses console I/O.
- Works like StreamTokenizer
- Implements Iterator<String>
- Perl-like pattern matching available
- Constructors
  Scanner( File source )
  Scanner( InputStream source )
  Scanner( String source )

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

Testing Methods

- boolean hasNext()
- boolean hasNext( Pattern ptrn )
- boolean hasNextBoolean()
- boolean hasNextDouble()
- boolean hasNextInt()
- boolean hasNextInt( int radix )
- boolean hasNextLine()
A Sampling of Methods for More Advanced Use

IOException ioException()
    What was the last exception thrown by the underlying Readable object?
Scanner skip( Pattern ptrn )
    Throw away some input.
Scanner useDelimiter( Pattern ptrn )
    Change what is to be considered "white space".
Scanner useRadix( int radix )
    Change the default base for reading integers.

It works pretty much like C.

```java
public class VarArgs {
    public static void main( String[] args ) {
        System.out.printf( "%4d--%9.3e--%s--(%h)%n", 43, 6.02e23, "Hello, world!", new VarArgs() );
    }
}
```

43--6.020e+23--Hello, world!--(9cab16)

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

Additional features

- Annotations
  - Allows you to mark methods as overridden, or deprecated, or to turn off compiler warnings for a method
  - You can create other kinds of annotations, to eliminate boilerplate code
- Threading
  - There are many new features for controlling synchronization and threading (concurrency API)
- Profiling
Closing comments

- I've just skimmed the surface of the new features
- Most of the Java 5 extensions are designed for ease of use, but unfortunately not for ease of learning
- …