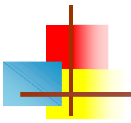
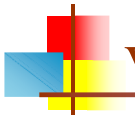


Java 1.5

New Features

25-Feb-07



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

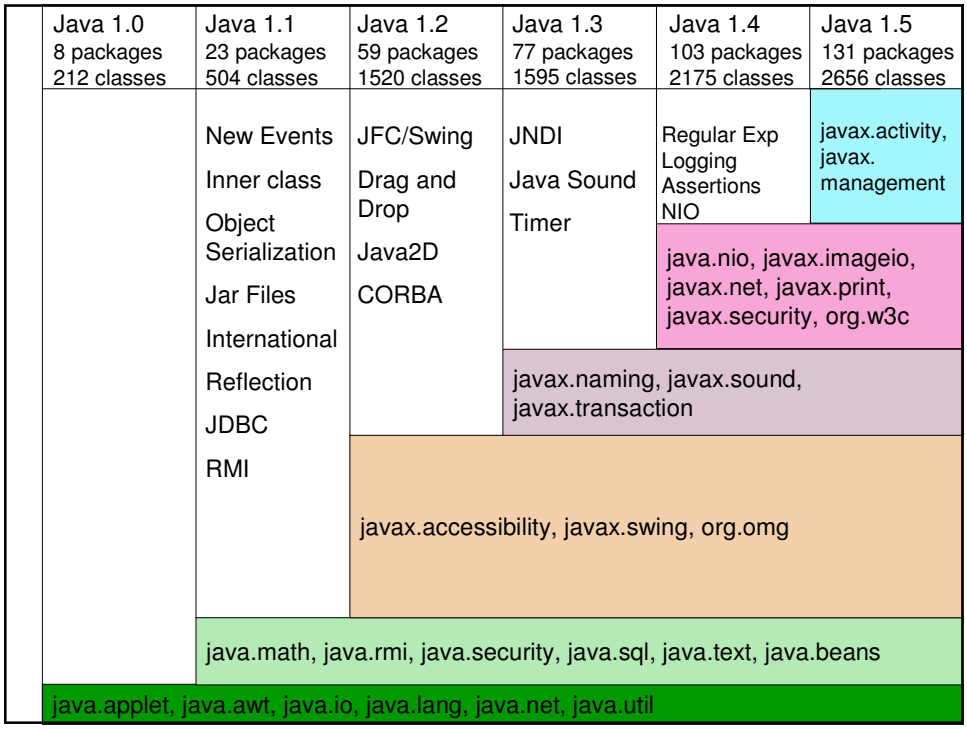
Java 1.4: More additions and the assert statement

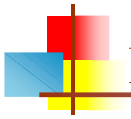
} Java 2

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

} Java 5.0

2





New features

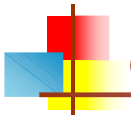
- Generics
  - Compile-time type safety for collections without casting
- Enhanced for loop
  - Syntactic sugar to support the Iterator interface
- Autoboxing/unboxing
  - Automatic wrapping and unwrapping of primitives
- Typesafe enums
  - Provides all the well-known benefits of the Typesafe Enum pattern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

4

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

5



Generics

- A generic is a method that is recompiled with different types as the need arises
- The bad news:
  - Instead of saying: ArrayList words = new ArrayList();
  - You'll have to say:
    ```
    ArrayList<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

6

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);
    }
    ```

7

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>

8

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());
        }
    }
    ```

9

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

10
  ```

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;
}
```

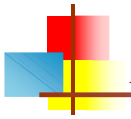
- For a collection class that has an Iterator, instead of

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

you can now say

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

11



Auto boxing and unboxing

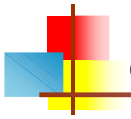
- Previously, Java didn't let you use a primitive value where an object is required--you need a "wrapper"
  - `myVector.add(new Integer(5));`
- Similarly, you can't use an object where a primitive is required--you need to "unwrap" it
  - `int n = ((Integer)myVector.lastElement()).intValue();`
- Java 1.5 makes this 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`

12

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 }

13



enums are classes

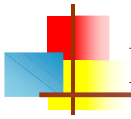
- An enum is actually a new type of class
  - You can declare them as inner classes or outer classes
  - You can declare variables of an enum type and get type safety and compile time checking
    - Each declared value is an instance of the enum class
    - Enums are implicitly public, static, and final
    - 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

14

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

15



Enums *really are* classes

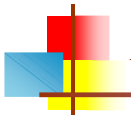
```
public enum Coin {
    // enums can have instance variables
    private final int value;
    // An enum can have a constructor, but it isn't public
    Coin(int value) { this.value = value; }
    // Each enum value you list really calls a constructor
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);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

16

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

17



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, "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*** }

18

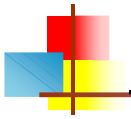
Static import facility

- 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*);

19



java.util.Scanner

- Finally, Java has a fairly simple way to read input
  - **Scanner sc = new Scanner(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

20

java.util.Formatter

- Java now has a way to produce formatted output, based on the C printf statement
- ```
  String line;
  int i = 1;
  double v = 3.14159;
  while ((line = reader.readLine()) != null) {
     System.out.printf("Line %d: %s %3.2f %n", i++, line, v);
  }
  ```
- There are about 45 different format specifiers (such as %d and %s), most of them for dates and times

21

Additional features

- Annotations
  - Allow you to mark methods as overridden, or deprecated, or to turn off compiler warnings for a method
  - You can create other kinds of annotations
- Threading
  - There are many new features for controlling synchronization and threading, new ways of using semaphore

22
  ```

Metadata

- Many APIs require a fair amount of boilerplate. For example, when you define a JAX-RPC Web Service you provide both an interface and an implementation class:

```
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) {
      ...
  }
}
```

23

Metadata

- With the metadata facility, you don't have to write all of this yourself. You just annotate the code to let a tool know which methods are remotely accessible, and the tool generates the above code:

```
import javax.xml.rpc.*;
public class CoffeeOrder {
   @Remote public Coffee [] getPriceList() {
      ...
   }
   @Remote public String orderCoffee(String name, int quantity)
{
      ...
   }
}
```

24

Closing comments

- Java 1.5 was released in September 2004
- I've just touched on the new features
- Most of the Java 1.5 extensions are designed for ease of use, but unfortunately not for ease of learning
- All things change...
  - They just change a lot faster in computer science
  - If you want to be a software developer you will have to keep up with many of these changes on your own!
- Recommendation: Install and learn Java 5

25