How is Java different from other languages

Less than you think:

Java is an imperative language (like C++, Ada, C, Pascal)
Java is interpreted (like LISP, APL)
Java is garbage-collected (like LISP, Eiffel, Modula-3)
Java can be compiled (like LISP)
Java is object-oriented (like C++, Ada, Eiffel)

A successful hybrid for a specific-application domain
A reasonable general-purpose language for non-real-time applications

- **Work in progress**: language continues to evolve
- C# is latest, incompatible variant
Original design goals (white paper 1993)

Simple
Object-oriented (inheritance, polymorphism)
Distributed
Interpreted
multithreaded
Robust
Secure
Architecture-neutral

• a language with threads, objects, exceptions and garbage-collection can’t really be simple!
Portability

Critical concern: write once-run everywhere

Consequences:
- Portable interpreter
- definition through virtual machine: the JVM
- run-time representation has high-level semantics
- supports dynamic loading
- (+) high-level representation can be queried at run-time to provide reflection
- (-) Dynamic features make it hard to fully compile, safety requires numerous run-time checks
Contrast with conventional systems languages (C, C++, Ada)

Conventional Imperative languages are fully compiled:

- run-time structure is machine language
- minimal run-time type information
- language provides low-level tools for accessing storage
- safety requires fewer run-time checks because compiler (least for Ada and somewhat for C++) can verify correctness statically.

Languages require static binding, run-time image cannot be easily modified
Different compilers may create portability problems
Omissions

No operator overloading (syntactic annoyance)

No enumerations (using final constants is clumsy)

No separation of spec and body

No generic facilities until latest language release.
A new construct: interfaces

Allow otherwise unrelated classes to satisfy a given requirement

Orthogonal to inheritance

- inheritance: an A is-a B (has the attributes of a B, and possibly others)
- interface: an A can-do X (and other unrelated actions)

- better model for multiple inheritance

More costly at run-time (minor consideration)
public interface Comparable {
    public int compareTo (Object x) throws ClassCastException;

    // returns -1 if this < x,
    //          0 if this = x,
    //          1 if this > x

};

// Implementation has to cast x to the proper class.
// Any class that may appear in a container should implement Comparable, so that container can support sorting.
class Doodler extends Thread {
    // override the basic method of a thread
    public void run() {
        ... // scribble something
    }
}

Doodler gary = new Doodler();
gary.start(); // calls the run method
The runnable interface allows any object to have dynamic behavior

class Simple_Gizmo { …}
class Active_Gizmo extends Simple_Gizmo
    implements Runnable {
        public void run ( ) {…}
    }

    a thread can be constructed from anything that runs:
    Thread thing1 = new Thread (new Active_Gizmo ( ));
    Thread thing2 = new Thread (new Active_Gizmo ( ));
    thing1.start( ); thing2.start ( );
Interfaces and event-driven programming

A high-level model of event-handling:

- graphic objects generate events
  - mouse click, menu selection, window close...
- an object can be designated as a handler
  - a listener, in Java terminology
- an event can be broadcast to several handlers
  - several listeners can be attached to a source of events
- a handler must implement an interface
  - actionPerformed, keyPressed, mouseExited..
Built-in interfaces for event handlers

```java
public interface MouseListener {
    void mousePressed   (MouseEvent event);
    void mouseReleased  (MouseEvent event);
    void mouseClicked   (MouseEvent event);
    void mouseEntered   (MouseEvent event);
    void mouseExited    (MouseEvent event);
}
```

Typically, handler only needs to process a few of the above, and supply dummy methods for the others
Adapters: a coding convenience

class mouseAdapter implements mouseListener
{
    public void mousePressed (MouseEvent event) { } ;
    public void mouseReleased (MouseEvent event) { } ;
    public void mouseClicked (MouseEvent event) { } ;
    public void mouseEntered (MouseEvent event) { } ;
    public void mouseExited (MouseEvent event) { } ;
};
class MouseClickListener extends MouseAdapter {
    public void mouseClicked (MouseEvent event {...});
    // only the method of interest needs to be supplied
}
Events and listeners

class Calm_Down extends JFrame {
    private JButton help := new JButton ("HELP!!!");
    // indicate that the current frame handles button clicks
    help.addActionListener (this);

    // if the button is clicked the frame executes the following:
    public void actionPerformed (ActionEvent e) {
        if (e.getSource () == help) {
            System.out.println ("can’t be that bad. What’s the problem?"));
        }
    }
}
Event handlers and nested classes

- Inner classes make it possible to add local handlers to any component

  ```java
  class reactive_panel extends JPanel {  // a swing component
      JButton b1;
      public reactive_panel (Container c) {
          b1 = new JButton ("flash");
          add (b1);
      }
  }

  private class MyListener implements ActionListener {
      public void actionPerformed (ActionEvent e) { ...}
  }
  MyListener ml = new MyListener ( );
  b1.addActionListener (ml);
  ```
Event handlers without classes

Need to connect low-level event detected by hardware/OS to user code.

**Ada**: designate operation in protected type

**C#**: use delegation (really a function member)
Interrupt handlers in Ada

```ada
protected watch is
  -- Single object with locks
  procedure update;
  pragma Interrupt_handler (update);
  pragma Attach_Handler (update, SIGSEGV);
  -- Place pointer to procedure in interrupt vector
end watch;

Watch must be declared at the library level (not nested within a subprogram).
```
Delegates and events in C#

Delegates are access_to_subprogram types with special operations

```csharp
delegate string MessageFormat();
// can be instantiated with any parameterless procedure

class Message {
    ...
    
    public static string Title();
    ...

    MessageFormat format = new MessageFormat(Message.Title);
    // format can be stored, passed as parameter, or invoked.
```
Event generation

An event is declared with the delegate type that can handle it:

delegate void UpdateEventHandler ();
class Subject {
    private int data;
    public void SetData (int value){ ... Changed ( )};
    public event UpdateEventHandler Changed;

    // whenever a Subject is modified, it will trigger an event
    // in its registered observers.
Event Handling

class observer {
    private Subject subject;  // an observer watches one object
    public Observer (Subject s) {
        // attach to it, and register for callback
        subject = s;
        s.Changed += new UpdateEventHandler (this.Update);
    }
    public void Update () {System.Console.Write.....}
}
// a change in the subject will trigger the callback
**Introspection, reflection, and typeless programming**

```java
public void DoSomething (Object thing) {
    // what can be do with a generic object?
    if (thing instanceof gizmo) {
        // we know the methods in class Gizmo
        ....
    }
}
```

*instanceof* requires an accessible run-time descriptor in the object.

Reflection is a general programming model that relies on run-time representations of aspects of the computation that are usually **not available to the programmer**.

More common in Smalltalk and LISP.
Reflection and metaprogramming

Given an object at run-time, it is possible to obtain:
- its class
- its fields (data members) as strings
- the classes of its fields
- the methods of its class, as strings
- the types of the methods

It is then possible to construct calls to these methods

- This is possible because the JVM provides a high-level representation of a class, with embedded strings that allow almost complete disassembly.
Reflection classes

java.lang.Class
  Class.getMethod ( ) returns array of method objects
  Class.getConstructor (Class[ ] parameterTypes)
    • returns the constructor with those parameters

java.lang.reflect.Array
  Array.newInstance (Class componentType, int length)

java.lang.reflect.Field

java.lang.reflect.Method
  • All of the above require the existence of run-time objects that describe methods and classes
Reflection and Beans

The beans technology requires run-time examination of foreign objects, in order to build dynamically a usable interface for them.

Class Introspector builds a method dictionary based on simple naming conventions:

```java
public boolean isCoffeeBean(); // is... predicate
public int getRoast(); // get... retrieval
public void setRoast(int darkness); // set... assignment
```
An endless supply of libraries

- The power of the language is in the large set of libraries in existence. The language is successful if programmers find libraries comfortable:
  - JFC and the Swing package
  - Pluggable look and Feel
  - Graphics
  - Files and Streams
  - Networking
  - Enterprise libraries: CORBA, RMI, Serialization, JDBC