Java: According to the Court

‘‘Sun's Java Technology is a collection of programming components that create a standard, platform-independent programming and runtime environment. Sun's Java Technology has two basic elements: the **Java programming environment** and the **Java runtime environment**.

‘‘The **Java programming environment** allows software developers to create a single version of program code that is capable of running on any platform which possesses a compatible implementation of the Java runtime environment. The Java programming environment comprises (1) Sun's specification for the Java language, (2) Sun's specification for the Java class libraries and (3) the Java compiler.

‘‘The **Java runtime environment** comprises the Java class libraries and the Java runtime interpreter. A system platform or browser program that implements the Java runtime environment can execute application programs developed using the Java programming environment. ’’

Example

```
void spin() {
    int i;
    for (i = 0; i < 100; i++) {
        ;        // Loop body is empty
    }
}
```

```
Method void spin()
 0 iconst_0     //Push int constant 0
 1 istore_1     //Store into local 1 (i=0)
 2 goto 8       //First time don't increment
 5 iinc 1 1     //Increment local 1 by 1 (i++)
 8 iload_1      //Push local 1 (i)
 9 bipush 100   //Push int constant (100)
11 if_icmplt 5  //Compare, loop if < (i<100)
14 return       //Return void when done

Bytecodes:
3 60 167 0 8 132 1 1 27 16 100 161 0 5 177
```

Java Design Goals

"The Java Language: A White Paper," Sun Microsystems:

"Java: a

simple,

object-oriented,

distributed,

interpreted,

robust,

secure,

architecture neutral,

portable,

high-performance,

multithreaded, and

dynamic language."

Java Language Elements

Primitives:

- Primitive Data
  booleans: **`true`**, **`false`**
  integers: **`byte`** (8 bits), **`short`** (16 bits),
  `int` (32 bits), **`long`** (64 bits)
  floating point: **`float`** (32 bits), **`double`** (64 bits)
- Objects

Means of Combination:

- Arrays, e.g. **`int xarr[] = {1, 2, 3};`**
- Method invocation

Means of Abstraction

- Variables, e.g. **`int x = 27;`**
- Classes
- Interfaces
- Packages

Java Class Model

- A Class defines
  - instance variables
  - constructors (for instance creation)
  - methods (invoked on instances)
  - class variables
  - class methods
- Method Invocation
- Object References
  - Primitive data are *pass-by-value*
  - Objects are *pass-by-reference*
- **Single** Inheritance

Class Example 1: Class as Data Structure

```
class Body {
    public long idNum;       //instance vars
    public String nameFor;
    public Body orbits;

    public static long nextID = 0;  //class var
}
```

Object (Variable) Declaration:

```
Body mercury;     // declare but no creation
```

Object Creation:

```
Body sun = new Body();
sun.idNum = Body.nextID++;
sun.nameFor = "Sol";
sun.orbits = null;

Body earth = new Body();
earth.idNum = Body.nextID++;
earth.nameFor = "Earth";
earth.orbits = sun;
```

Class Example 2: With Constructors, Instance Methods, and Class Methods

```
class Body {
    public long idNum;   //instance vars
    public String nameFor;
    public Body orbits;
    public long id() { return idNum; }
    private static long nextID = 0; //class var
    public long numBodies() {
      return nextID;
      }
    Body() { idNum = nextID++; }
    Body(String bodyName, Body orbits) {
        this(); //explicit construct invocation
        name = bodyName;
        this.orbits = orbist;
    }
}

  Body sun = new Body();
  sun.nameFor = "Sol";
  sun.orbits = null;
  Body earth = new Body("Earth", sun);
  System.out.println(earth.nameFor +
    " is body number" + earth.id() +
    " out of " + Body.numBodies());

Earth is body number 1 out of 2 bodies
```

Object References

```
Body sun = new Body("Sol", null);
Body earth = new Body("Earth", sun);
Body home = earth;
home.nameFor = "Home Sweet Home";

System.out.println("Earth is " +
    earth.nameFor);
```

```
Earth is Home Sweet Home
```

Garbage Collection

```
Body sun = new Body("Sol", null);
Body earth = new Body("Earth", sun);
```

```
sun = new Body("Center of Universe", null);
```

Question: is the old **sun** object now garbage?

```
earth = new Body("FlatEarth", sun);
```

Question: is the old **earth** object now garbage?
How about the old **sun** object?

Garbage Collection and**Finalization**

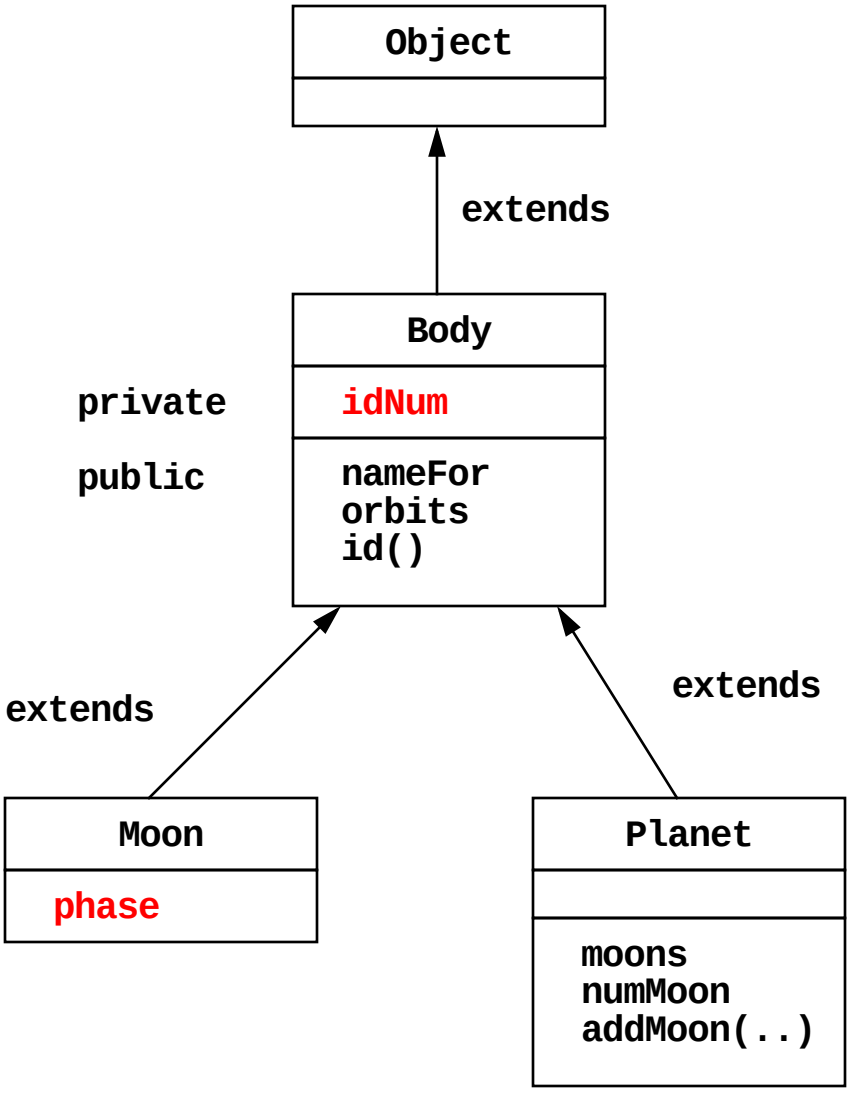
Finalization provides an opportunity to perform user-defined "clean up" operations just before an object is garbage collected:

```
public class ProcessFile {
    private Stream file;

    public ProcessFile(String path) {
        File = new Stream(path);
    }

    // ...

    public void close() {
        if (File != null) {
            File.close();
            File = null;
        }
    }

    protected void finalize() throws Throwable
        super.finalize();
        close();
    }
}
```

Inheritance

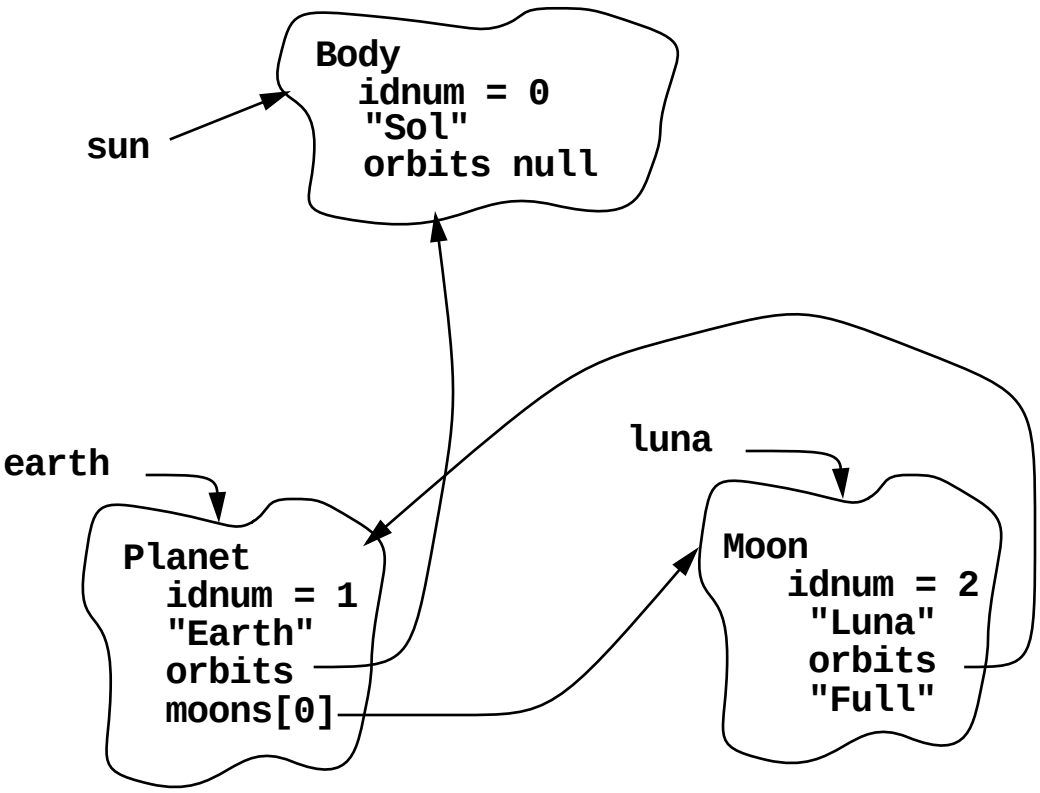
```
class Moon extends Body {
  public String phase;
  Moon(String name, Body orbits, String phase) {
    super(name, orbits);
    this.phase = phase;
    orbits.addMoon(this);
  }
}

class Planet extends Body {
  final int MAX_MOONS = 10;
  public Moon[] moons = new Moon[MAX_MOONS];
  public int numMoon = 0;
  public void addMoon(Moon m) {
    moons[numMoon++] = m;
  }
}

Body sun = new Body("Sol", null);
Planet earth = new Planet("Earth", sun);
Moon luna = new Moon("Luna", earth, "Full");
```

Java Inheritance



- Java enforces only single inheritance:
- Every class has exactly one superclass
  - except **`Object`** which is the base class

Java Instances



Java "Interfaces"

- The **`interface`** is a way to "standardize" methods and behaviors across classes without inheritance
  - **specification** of methods
  - does not include **implementation** for these methods

- Interfaces can have (multiple) sub-interfaces

Interface Example

```
public interface Drawable {
  public void setPosition(double x, double y);
  public void draw(DrawWindow dw);
  }

class Body implements Drawable {
    private double x_pos = 0.0, y_pos = 0.0;
    public long idNum;   //instance vars
    public String nameFor;
    public Body orbits;
    ...
    public void setPosition(double x, double y) {
      x_pos = x; y_pos = y;
    }
    public void draw(DrawWindow dw) {
      dw.drawCircle(x_pos, y_pos, 1.0);
    }
    ...
}
```

Java Packages

The Package is a means to scope names and provide/limit access to classes and methods within/outside the package:

1. Naming:
   - Universally(!) unique names for classes and methods:

```
java.lang.String.substring()
EDU.mit.www-mtl.boning.worlds.Body.numBodies()
```

   - One can use "shortened" names for classes within a package, or from "imported" packages:

```
import package;
import package.class;
import package.*;

import java.lang.*;   //implicit
```

2. Access/Inheritance: Various rules about access to packages, classes, and fields within classes
   - **`public`** classes are accessible within another package
   - **`private`** fields in a class only accessible to methods within that class
   - **`private protected`** fields in a class accessible to that class and subclasses of that class

Exception Handling

- **Exceptions** arise when methods are invoked on objects due to:
  - discovered internal state problems
  - errors with objects or data
  - discovery of contract violations
  - array access out of bounds
  - out of memory

- Exceptions are **thrown** by the system or by the programmer when an unexpected error condition is encountered.

- The exception is **caught** by surrounding code somewhere up the (dynamic) call chain designed to deal with the exception.

- Java exceptions are **objects** (instances of `java.lang.Error` or `java.lang.Exception`)
  - the programmer can add their own exceptions and exception handling

Exception Handling Code

```
try {
  // "Normal" operation of this block...
}
catch (SomeException e1) {
  // Handle SomeException...
}
catch (SomeOtherException e2) {
  // Handle SomeOtherException...
}
finally {
  // Code to do after successful try or
  // handled exception...
}
```

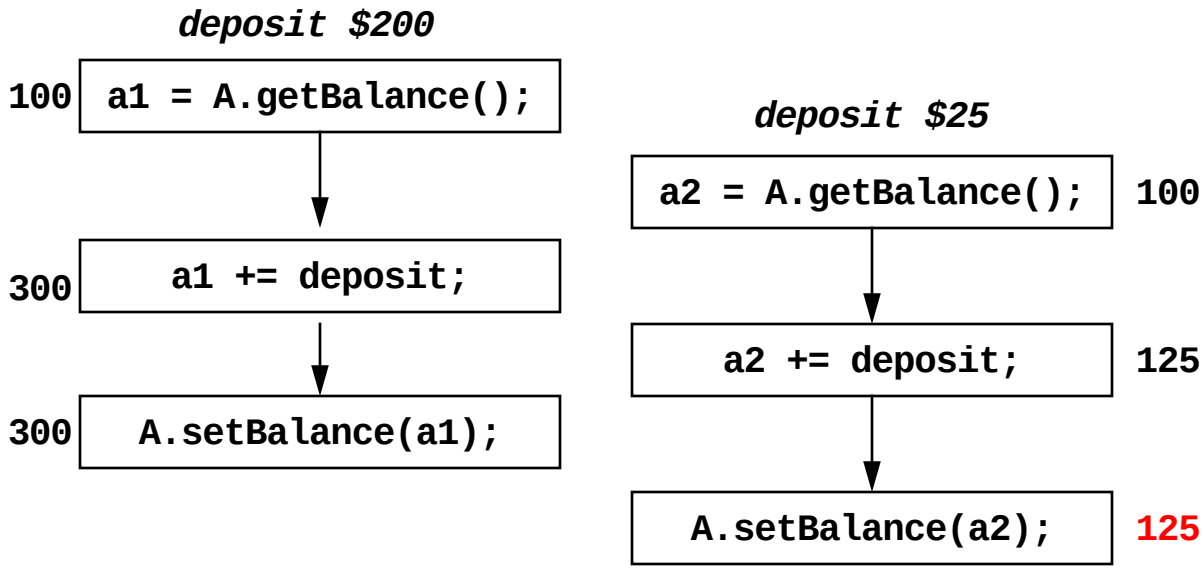
Example

```
boolean searchFor(String file, String word)
                throws StreamException
{
    Stream input = null;
    try {
        input = new Stream(file);
        while (!input.eof())
            if (input.next() == word)
                return true;
        return false;  // word not found
    } catch (FileNotFoundExcept e) {
        // Don't panic if file doesn't exist..
        return false;
    } finally {
        if (input != null)
            input.close();
    }
}
```

- Note: The finally block is always executed:
  - at the end of the try block (if no exceptions occurred)
  - before the "`return true`" or "`return false`" in the try block
  - at the end of a caught exception - unless the catch block itself changes the control (e.g. via a break or return)

Threads

- A **thread** is a **process** which can run concurrently with other threads



- **Synchronization** is provided by **locking on shared objects**:

```
Account A;
...
synchronized (A) {  // blocks if object A is locked
  double a1 = A.getBalance();
  a1 += 200.0;
  A.setBalance(a1);
  }

synchronized (A) {  // blocks if object A is locked
  double a2 = A.getBalance();
  a2 += deposit;
  A.setBalance(a2);
  }
```

Multithreading, Continued

- One can also declare synchronized methods:

```
class Account {
  private double balance;
  public Account(double initialDeposit) {
    balance = initialDeposit;
  }

  public synchronized double getBalance() {
    return balance;
  }

  public synchronized double
                deposit(double amount) {
    balance += amount;
  }
}
```

Wait and Notify

- Mechanisms are also provided to negotiate control and communicate between threads:

```
synchronized void doWhenCondition() {
  while (!condition)
    wait();  // halt thread until notified
  ... Do what needs doing when condition true
}


synchronized void changeCondition() {
  ... Change some values used in a condition
  notify(); // give up lock and notify threads
}
```

Wait/Notify Example

```
class Queue {
  Element head, tail;

  public synchronized void append(Element p) {
    if (tail == null)
      head = p;
    else
      tail.next = p;
    p.next = null;
    tail = p;
    notify(); // Let waiters know of arrival
  }

  public synchronized Element get() {
    while(head == null)
      wait(); // Wait for an element
    Element p = head; // Remember first element
    head = head.next; // Remove it from queue
    if (head == null) // Check for an empty Q
      tail = null;
    return p;
  }
}
```

Java Application Programming Interface (API)

- A standard set of classes that programmers can use

Some Examples:

- Networking Functionality (**`java.net`**)
  - **`URL`** class
- Graphical User Interfaces (**`java.awt`**)
- Input/Output Package (**`java.io`**)
  - **`Streams: InputStream, OutputStream, Piped, Filter`**
  - **`File`** handling
  - **`StreamTokenizer`** to parse streams
- Standard Utilities (**`java.util`**)
  - **`Enumeration`**
  - **`Vector`** (dynamically sized)
  - **`Stack`**
  - **`Dictionary`**
  - **`Hash Table`**
  - **`Observer/Observable`**
  - **`Date`**
  - **`Random`**
- Java Runtime System Access (**`java.lang`**)
  - Classes include: **`Runtime, Process, System, Math`**

Java Virtual Machine (JVM)

- Primitive data types are handled by JVM directly:
  - variants of opcodes for different types:
    `iconst, lconst, fconst, dconst`
  - type conversions
    `i2l, i2f, f2i, ...`
- Primitive operations (e.g. math, bit manipulation)
  - **`iadd, isub, idiv, imul; ior, ishr, ishl`**
- Stack oriented
  - operands to a JVM opcode are pushed/popped from an operand stack -- not registers
  - many opcodes deal with moving data (constants, references, addresses) between the stack and local variable stores:
    `iconst, bipush, iload, iaload, istore, swap, ...`
- Condition Testing and Flow control
  - **`if_icmpeq, if_icmpne, goto, jsr`**
- Support for Java class model:
  - data member access: **`getfield, putfield, putstatic`**
  - method invocation: **`invokevirtual, invokestatic, ...`**
  - method return values: **`return, ireturn, lreturn, ...`**
- Support for Exceptions: **`athrow`**
- Support for Synchronization: **`monitorenter, monitorexit`**
- Runtime system has a **Byte Code Verifier** on loaded classes
  - verify that language constraints are satisfied

Java Native Interface

- Issue: how can one access programs written in other languages (or underlying machine language)?
  - "All portability and safety of the code is lost"
  - Not appropriate for Applet code running on multiple hardware platforms
  - May be necessary for embedded system applications
- Need: A "standard" way to interface to native methods

- Language/Compiler **`native`** Keyword:

```
public native void unlock() throws IOEx-
ception;
```

Note that no Java implementation is present.

- JNI - Java Native Interface
  - public API to Java runtime interpreter
  - links Java code to native code through the JVM
  - provides native code with access to the JVM

Sun v. Microsoft

[...]
IV. ORDER

Since the court finds that Sun is likely to prevail on the merits and that it may suffer irreparable harm if Microsoft is not enjoined, a preliminary injunction is hereby issued against Microsoft [...] from:
[...]
(B) Selling or distributing, directly or indirectly, any software development tool or product containing or implementing computer program code copied or derived from any Sun copyrighted program code for the Java Technology; as that term is defined in the TLDA, including SDKJ 2.0, SDKJ 3.0 and VJ 6.0, ninety (90) days after the date of this order unless such product:

(1) includes a Java runtime implementation which supports Sun's JNI (including help files, header files, etc.) in a manner which passes the compatibility test suite accompanying the latest version of the Java Technology contained in, implemented by, or emulated by such product,

(2) has the default mode in the compiler configured such that (a) Microsoft's keyword extensions and compiler directives are disabled and (b) has the compiler mode switch such that it enables, rather than disables, such keyword extensions and compiler directives, and
[...]

Microsoft's "Delegation" Extension

```
public class SimpleExample extends JPanel {
  JButton button = new JButton("Hello, world");
  private void button_clicked(ActionEvent e) {
    System.out.println("Hello, world!");
}
...
public SimpleExample() {
  button.addActionListener(
    new ActionDelegate(this.button_clicked));
  add(button);
  ...
}
```

Sun's "Inner Classes"

```
public class SimpleExample extends JPanel {
  JButton button = new JButton("Hello, world");
  ...
  public SimpleExample() {
  button.addActionListener(
    new ActionListener() {
       public void actionPerformed(ActionEvent e)
        {System.out.println("Hello, World!");}
       });
    add(button);
    ...
}
...
```

Example: Sorting an Array

```
void sortIgnoreCase(String words[]) {
  Arrays.sort(words, new Comparator() {
    public int compare(Object o1, Object o2) {
      String s1 = ((String)o1).toLowerCase();
      String s2 = ((String)o2).toLowerCase();
      return s1.compareTo(s2);
    }
});
...
}
```