

A large, two-story, light-colored building with a red-tiled roof and a central tower, surrounded by green lawns and trees under a clear blue sky.

# MAHARISHI UNIVERSITY of MANAGEMENT

*Engaging the Managing Intelligence of Nature*

## Computer Science Department

### CS401 Modern Programming Practices (MPP)

Professor Renuka Mohanraj

© 2015 Maharishi University of Management, Fairfield, Iowa

All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi University of Management.

# Lecture 7: Interfaces in Java 8 and the Object Superclass

# Wholeness Statement

Java supports inheritance between classes in support of the OO concepts of inherited types and polymorphism. Interfaces support encapsulation, play a role similar to abstract classes, and provide a safe alternative to multiple inheritance.

Likewise, relationships of any kind that are grounded on the deeper values at the source of the individuals involved result in fuller creativity of expression with fewer mistakes.

# Overview

1. Java 8 introduces many new features in interfaces and applications of these features to the collections library
2. The Object class is the superclass of all Java classes. It offers several important methods (usable by any Java class), but to be useful, they must be overridden (in the right way).
3. Discussion on Inner class

# Java 8 Features of Interfaces

- Before Java 8, none of the methods in an interface had a method body; all were unimplemented.
- In Java 8, two kinds of implemented methods are now allowed: *default methods* and *static methods*. Both can be added to legacy interfaces without breaking code.

- A default method is a fully implemented method within an interface, whose declaration begins with the keyword `default`
- A static method in an interface is a fully implemented static method having the same characteristics as any static method in a class.

See Demos in package `lesson7.lecture.defaultmethods` and `lesson7.lecture.interfacestatic`

# New Programming Style

***Default Methods*** in an interface eliminate the need to create special classes that represent a default implementation of the interface.

- Examples from pre-Java 8 of default implementations of interfaces: List / Collection. [See JavaLibrary project in workspace]
- Now, in developing new code, it is possible in many cases to place these default implementations in the interface directly.

***Static Methods*** in an interface eliminate the need to create special utility classes that naturally belong with the interface.

- Examples from pre-Java 8 of how interfaces sometimes have companion utility classes (consisting of static methods):  
Collection / Collections [See JavaLibrary project]  
Path / Paths.



# Solution to Evolving API Problem

When you need to add new methods to an existing interface, provide them with default implementations using the new Java 8 default feature. Then

- legacy code will not be required to implement the new methods, so existing code will not be broken
- new functionality will be available for new client projects.

# Other Uses

## First Set of Examples:

`enums` can now “inherit” from another type

## Second Set of Examples:

`forEach` – default method in `Iterable`

# First Set of Examples:

## Review of Enums

- An *enumerated type* is a Java class all of whose possible instances are explicitly enumerated during initialization.
- Example:

```
public enum Size { SMALL, MEDIUM, LARGE};
```

**//usage:**

```
if(requestedSize==Size.SMALL)  
    applyDiscount();
```

- The enum `Size` (which is a special kind of Java class) has been declared to have just three instances, named `SMALL`, `MEDIUM`, `LARGE`.

# Review of Enums

Two important applications for enums:

1. Provide a list of related “constants” for an application
  - Weak programming practice: Create a class (or interface) containing constants, stored as public static final values

*Problem.* No compiler control over usage of constants

- Better approach when constants are related to each other: Represent constants as instances of an enumerated type.
2. Optimal, threadsafe implementation of the Singleton Pattern

# Example: Improve style of managing constants with enums

In the java.awt package there is a class Label, used to represent a label in a UI (built from the old AWT). It makes use of constants to designate alignment properties: left, center, right. This use of constants is flawed, but it is a commonly used style

```
//extracted from java.awt.Label
//Java library does it the bad way
public class Label {
    private String text;
    private int alignment;
    public Label(String text, int alignment) {
        this.text = text;
        setAlignment(alignment);
    }
    public synchronized void setAlignment(int alignment) {
        switch (alignment) {
            case AlignmentConstants.LEFT:
            case AlignmentConstants.CENTER:
            case AlignmentConstants.RIGHT:
                this.alignment = alignment;
                return;
        }
        throw new IllegalArgumentException(
            "improper alignment: " + alignment);
    }
    public String getText() {
        return text;
    }
    public int getAlignment() {
        return alignment;
    }
}
```

```
public class AlignmentConstants {
    /**
     * Indicates that the label should be left justified.
     */
    public static final int LEFT = 0;

    /**
     * Indicates that the label should be centered.
     */
    public static final int CENTER = 1;

    /**
     * Indicates that the label should be right justified.
     * @since JDK1.0t.
     */
    public static final int RIGHT = 2;
}
```

**Problem**: No compiler control over use of these constants. Could make the following call:

```
Label label = new Label("Hello", 23);
```

You won't know till you run the code that "23" is meaningless. The compiler sees that a value of the correct type has been used, but 23 should not be used here.

It is better to control the values passed in with the help of the compiler. This is accomplished using an enum to store constants, rather than collecting together a bunch of public static final integers.

# Improved Label Using enums

//Better way, not currently implemented

//in Java libraries

```
public class Label {
    private String text;
    private Alignment alignment;
    public Label(String text, Alignment alignment) {
        this.text = text;
        setAlignment(alignment);
    }
    public synchronized void setAlignment(Alignment alignment) {
        this.alignment = alignment;
    }
    public String getText() {
        return text;
    }
    public Alignment getAlignment() {
        return alignment;
    }
}
```

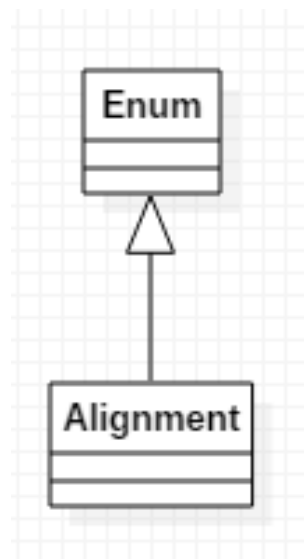
```
public enum Alignment {
    /**
     * Indicates that the label should be left justified.
     */
    LEFT,

    /**
     * Indicates that the label should be centered.
     */
    CENTER,

    /**
     * Indicates that the label should be right justified.
     * @since JDK1.0t.
     */
    RIGHT;
}
```

# Review of Enums

- In the example, each of the instances declared within the Alignment enum has type Alignment, which is a subclass of Enum. All enums implicitly extend java.lang.Enum. Therefore
  - Alignment is itself a *class*
  - Alignment may not inherit from any other class ( since there is no multiple inheritance).





# First Set of Examples:

## Using enums to Create Singletons

- A *singleton* class is a class that can have at most one instance
- Easy implementation using an `enum`:

```
enum MySingleton {  
    INSTANCE;  
    public void behavior() {}  
}  
  
//access it like this:  
MySingleton.INSTANCE.behavior();
```

Demo: `lesson7.lecture.singletons`



# First Set of Examples:

## In Java 8, Enums Can implement “Interface”

```
interface MyInterface {  
    default usefulThings() {  
        . . .  
    }  
}  
  
enum MySingleton implements MyInterface {  
    INSTANCE;  
    @Override  
    public void behavior() {  
        . . .  
        usefulThings();  
    }  
}
```

# Second Set of Examples: `forEach`

- The `Iterable` interface is part of the Collections API that is implemented by all collection classes, and supports iteration through a collection
- The only method in `Iterable` is `iterator()`, which returns an `Iterator`
- `Iterator` has two methods:
  - `hasNext()`
  - `next()`
- When a class (even user-defined) implements the `Iterable` interface, the “for each” construct can be used (and of course, an instance of `Iterator` is available).

See Demo: `lesson7.lecture.iterator`

New (Java 8) in the Iterable interface is a default method:

## **forEach**

Sample usage:

```
Consumer<String> consumer = new Consumer<String>() {  
    @Override  
    public void accept(String s) {  
        System.out.println(s);  
    }  
};  
System.out.println("-----using new forEach method-----");  
l.forEach(consumer);
```

Output:

```
-----using new forEach method-----  
Bob  
Steve  
Susan  
Mark  
Dave
```

See Demos:  
[lesson7.lecture.iterator](#)

Review inner classes:  
[Lesson7.lecture.reviewinner](#)

1. The `forEach` method applies the `Consumer` method `accept` to each element of the list.
2. In this example, the `accept` method just prints the value to the console.
3. `Consumer` is a new interface in Java 8, with just one abstract method `accept`, which accepts a single argument and produces no return value.

# Quick Review of Nested Classes I

```
public class Static {  
    private String name = "Joe";  
    private Pair p = new Pair();  
    {  
        p.first = 4;  
        p.second = 5;  
        System.out.println(p);  
    }  
    private void printHello() {  
        System.out.println("Hello" + name);  
    }  
    static class Pair {  
        int first;  
        int second;  
        Pair() {  
            //no access  
            //printHello();  
        }  
        public String toString() {  
            return "(" + first + ", " + second + ")";  
        }  
    }  
    public static void main(String[] args) {  
        (new Static()).printHello();  
    }  
}
```

**Static Nested Class**

```
public class Member {  
    private String name = "Joe";  
    private Pair p = new Pair();  
    {  
        p.first = 4;  
        p.second = 5;  
        System.out.println(p);  
    }  
    private void printHello() {  
        System.out.println("Hello " + name);  
    }  
    class Pair {  
        int first;  
        int second;  
        Pair() {  
            printHello();  
        }  
        public String toString() {  
            return "(" + first + ", " + second + ")";  
        }  
    }  
    public static void main(String[] args) {  
        new Member();  
    }  
}
```

**Member Inner Class**

# Quick Review of Nested Classes II

See Demo : [lesson7.lecture.reviewofinner.foureexamples](#)

```
public class Local {  
    private String name = "Joe";  
    public void printPair(int x, int y) {  
        class Pair {  
            int first;  
            int second;  
            Pair() {  
                printHello(name);  
            }  
            public String toString() {  
                return "(" + first + ", "  
                    + second + ")";  
            }  
        }  
        Pair p = new Pair();  
        p.first = x;  
        p.second = y;  
        System.out.println(p);  
    }  
    private void printHello(String n) {  
        System.out.println("Hello " + n);  
    }  
    public static void main(String[] args)  
        (new Local()).printPair(11, 3);  
}
```

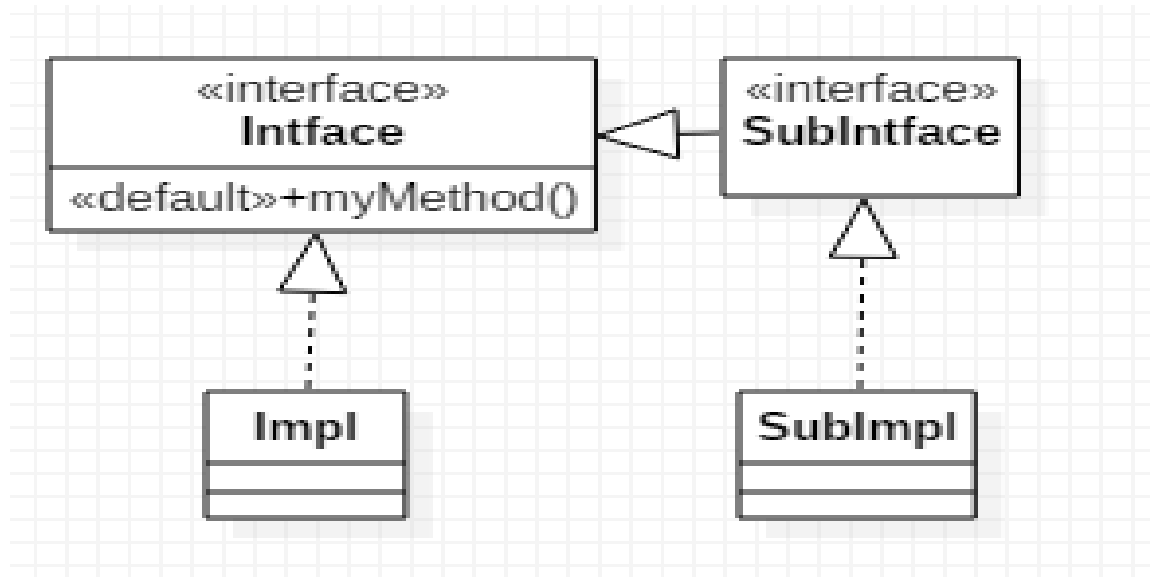
**Local Inner Class**

```
public class Anonymous {  
    interface IPair {  
        public void printHello();  
    };  
  
    private String name = "Joe";  
    public void printPair(int x, int y) {  
        (new IPair() {  
            int first = x;  
            int second = y;  
            public String toString() {  
                return "(" + first + ", " +  
                    second + ")";  
            }  
            public void printHello() {  
                System.out.println("Hello "  
                    name + "\n" + this);  
            }  
        }).printHello();  
    }  
    public static void main(String[] args)  
        (new Anonymous()).printPair(11, 3);  
}
```

**Anonymous Inner Class**

# Rules for Default Methods in an Interface

- If a class implements an interface with a default method, that class inherits the default method (or can override it).



- Potential clash if
  - two interfaces have the same method, or
  - one interface and a superclass have the same method



# Conflicts

```
Interface I1 {  
    default void display() {  
        System.out.println("Interface I1");  
    }  
}
```

```
Interface I2 {  
    default void display() {  
        System.out.println("Interface I2");  
    }  
}
```

```
// Interface conflicts  
Class Test implements I1, I2 {  
}
```

```
class C1 {  
    void display() {  
        System.out.println("Class C1");  
    }  
}
```

```
// Super class & Interface conflicts  
Class Test extends C1 implements I1 {  
}
```

- *Interface vs Interface* – clash! When two interfaces each have a method with the same signature:
  - If one of these is a default method, any implementer of both interfaces *must* override the method (or declare it as an abstract method) – can't simply do nothing.
  - If one of these is a default method, any *subinterface* of both interfaces must provide a default method (i.e. an implementation) of this method, or declare the method (even if unimplemented). (This rule requires no special effort.)
  - Note: Even in Java 7, it is not possible to implement two interfaces each of which has a method with the same signature but different return types.

- *Superclass vs Interface* — superclass wins! When a class extends a superclass and also implements an interface, and both super class and interface have a method with the same name, the superclass implementation wins — this is the version that is inherited by the class. The subclass/implementer is not required to override the shared method.
- See Demos in `lesson7.lecture.defaultmethodrules`

# Static Methods Do Not Clash

- Static methods defined in an interface are *not* inherited by implementers (this differs from the behavior for subclasses of a class)
- Therefore, if two interfaces implement static methods with the same signature, there is no clash to address when a class implements these interfaces.
- Static methods can always be accessed in a static way in such cases, but it is not related to inheritance.

See demo `lesson7.lecture.interfacestatic_clash`

# Main Point 1

Interfaces are used in Java to specify publicly available services in the form of method declarations. A class that implements such an interface must make each of the methods operational. Interfaces may be used polymorphically, in the same way as a superclass in an inheritance hierarchy. Because many interfaces can be implemented by the same class, interfaces provide a safe alternative to multiple inheritance. Java8 now supports static and default methods in an interface, which make interfaces even more flexible: For instance, enum can now “inherit” from other types and new public operations can be added to legacy interfaces without breaking code (as was done with the `forEach` method in the `Iterable` interface).

The concept of an interface is analogous to the creation itself – the creation may be viewed as an “interface” to the undifferentiated field of pure consciousness; each object and avenue of activity in the creation serves as a reminder and embodiment of the ultimate reality.

# Quiz - 1

1. Enum can implement Interface

a. True    b. False

2. What is the output of the following code?

```
interface i1 {  
    default int show() {  
        return 100;  
    }  
}  
  
abstract class A {  
    public int show() {  
        return 50;  
    }  
}  
  
public class TestClass extends A implements i1 {  
    public static void main(String args[]) {  
        TestClass t = new TestClass();  
        System.out.println(t.show());  
    }  
}
```

a. 100                      b. 50                      c. Run time Error                      d. Compile time error

# Overriding Methods in the Object Class

The `Object` class is the superclass of all Java classes, and contains several useful methods - in most cases, they are useful *only if* they are overridden.

- `toString()`
- `Equals()`
- `hashCode()`

# The toString() Method

- Every class automatically is equipped with a `toString` method (by inheritance), but the default implementation simply prints out the class name followed by a code for a memory location.

Example:

```
public class Pair {  
    public String first;  
    public String second;  
    public static void main(String[] args) {  
        Pair p = new Pair();  
        p.first = "Joe";  
        p.second = "Smith";  
        System.out.println(p.toString());  
    }  
}
```

- (Useless) output: `toString.Pair@19e0bfd`



- When `toString()` is overridden, it is possible to capture the state of the current instance of the class and send it to a log file or to the console. This can help in solving a problem after the code has been released, and in debugging during development. Note the `@Override` annotation.

```
public class Pair {  
    public String first;  
    public String second;  
    @Override  
    public String toString() {  
        return "(" + first + ", " + second + ")";  
    }  
    public static void main(String[] args) {  
        Pair p = new Pair();  
        p.first = "Joe";  
        p.second = "Smith";  
        System.out.println(p.toString());  
    }  
}
```

- Output:        (Joe, Smith)

# Overriding equals ( )

- Default implementation in Java is same as ==

`ob1.equals(ob2)` if and only if `ob1 == ob2`  
if and only if references point to the same object

Example:

```
class Person {  
    private String name;  
    Person(String n) {  
        name = n;  
    }  
}
```

Two `Person` instances should be "equal" if they have the same name. But using the default implementation of `equals`:

```
Person p1 = new Person("Joe");  
Person p2 = new Person("Joe");  
boolean same = (p1.equals(p2)); //same has value false
```

# Properties of equals() method

- The equals method implements an equivalence relation on non-null object references:
- It is *reflexive*: for any non-null reference value x, x.equals(x) should return true.
- It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is *consistent*: It should be consistent. If x.equals(y) returns true, it should keep returning true until the state of x or y is modified. If x.equals(y) returns false, it should keep returning false until the state of x or y is modified.
- *Comparison with null reference*: For any non-null reference value x, x.equals(null) should return false.
- The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

# Correct Way to Do It

```
//overriding equals method in the Person class
@Override
public boolean equals(Object aPerson) {
    if(aPerson == null) return false;
    if(!(aPerson instanceof Person)) return false;
    Person p = (Person)aPerson;
    boolean isEqual = this.name.equals(p.name)
    return isEqual;
}
```

Things to notice:

1. The argument to `equals` must be of type `Object` (otherwise, compiler error)
2. If input `aPerson` is null, it can't possibly be equal to the current instance of `Person`, so `false` is returned immediately
3. If runtime type of `aPerson` is not `Person` (or a subclass), there is no chance of equality, so `false` is returned immediately
4. After the preliminary special cases are handled, two `Person` objects are declared to be equal if and only if they have the same name.

# Alternative Approach

- To check that the aPerson object is of the right type we used instanceof operator. This is called the *instance-of strategy for overriding equals*.
- An alternative is to call getClass on aPerson to see if it matches the value of getClass for the current object, called the *same-classes-strategy for overriding equals*

```
@Override
public boolean equals(Object aPerson) {
    if(aPerson == null) return false;
    if(aPerson.getClass() != this.getClass()) return false;
    Person p = (Person)aPerson;
    boolean isEqual = this.name.equals(p.name);
    return isEqual;
}
```

*Potential problem with same-classes-strategy:* If a subclass of `Person` is introduced, subclass inherits the `equals` method but it always returns `false` when comparing a superclass instance with a subclass instance. For this reason, whenever same-classes strategy is used, you should either:

- 1. declare the superclass *final* (to prevent subclassing) or
- 2. override `equals` separately in the subclass

See Demos

`lesson7.lecture.overrideequals.equalclassesstrategyXX`

*Potential problem with instance-of-strategy:* If a subclass of `Person` is introduced, subclass inherits the `equals` method. If subclass overrides `equals`, then an *asymmetric equals* is created. For this reason, whenever instance-of strategy is used, you should either:

- 1. declare the superclass *final* (to prevent subclassing) or
- 2. require that every subclass relies on the superclass version of `equals()` (and does *not* override `equals()` separately)

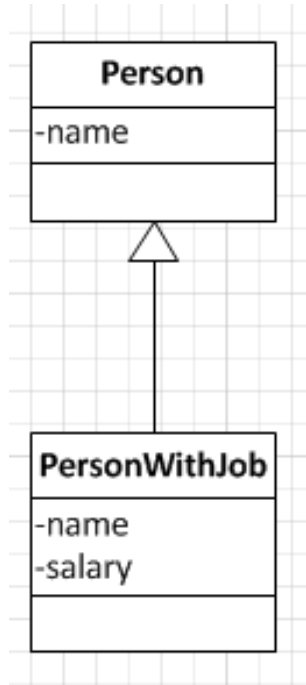
See Demos

`lesson7.lecture.overrideequals.instanceofstrategyXX`

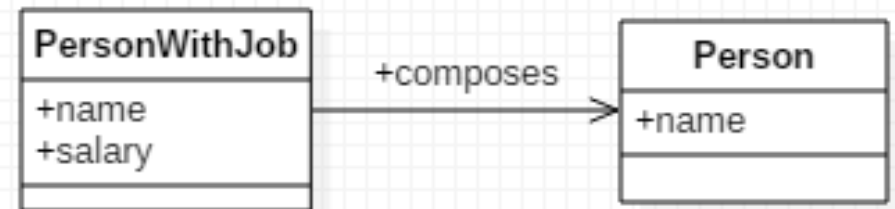
# A third alternative: Composition instead of Inheritance

- Using separate `equals` methods for superclass and subclass using either approach (equal-classes or instanceof strategy) is error-prone.
- Safe alternative: Replace inheritance with composition:

CHANGE



TO





# Exercise

Explain with an example how this way of “overriding” equals leads to logic errors in your code:

```
public class Person {  
    private String name;  
    public Person(String n) {  
        this.name = n;  
    }  
  
    public boolean equals(Person p) {  
        if(p == null) return false;  
        return q.name.equals(name);  
    }  
}
```

# Solution

This version of equals correctly compares two Person objects. The difficulty arises when the Object version of equals needs to be called, and there is no overriding version of equals. This happens (for example) when you want to test whether an object is contained in a List.

```
public class Main {  
    static ArrayList<Person> myList = new ArrayList<>();  
    static {  
        myList.add(new Person("Joe"));  
    }  
    public static void main(String[] args) {  
        Person p1 = new Person("Joe");  
        Person p2 = new Person("Joe");  
        System.out.println(p1.equals(p2)); //true  
        System.out.println(myList.contains(p1)); //false  
    }  
}
```

# Overriding hashCode ( )

1. Any implementation of the Hashtable ADT in Java will make use of the `hashCode ( )` function as the first step in producing a hash value (or table index) for an object that is being used as a key.
2. Default implementation of `hashCode ( )` provided in the Object class is not generally useful.

**Example:** We wish to use pairs (firstName, lastName) as keys for Person objects in a hashtable. (See Demo)

Demo illustrates that default `hashCode` method is not useful. By default, it simply gives a numeric representation of the memory location of an object. If two `Pair` objects, created at different times, are equal (using the `equals` method), we would expect them to have the same `hashCodes`, so that, after hashing, they are sent to the same table slot. But default `hashCode` method does not take into account the fields used by `equals` method, so equal `Pair` objects may be assigned different slots in the table.

3. **Conclusion:** *Whenever `equals` is overridden in a class, `hashCode` must also be overridden.*

# hashCode() Rules

- To use an object as a key in hashtable,
  - you must override `equals()` and `hashCode()`
  - the class on which object is based should be *immutable* (slide 45) – see demo `lesson7.lecture.hashcode.bad3`
- If  $k_1, k_2$  are keys and  $k_1.equals(k_2)$  then it must be true that  $[k_1.hashCode() == k_2.hashCode()]$   
This means that you must include the same information in your `hashCode` definition as you include in your implementation of `equals`.

## Creating a Hash Value from Object Data

(From Effective Java, 2<sup>nd</sup> Ed.)

You are trying to define a hash value for each instance variable of a class. Suppose `f` is such an instance variable.

- If `f` is boolean, compute `(f ? 1 : 0)`
- If `f` is a byte, char, short, or int, compute `(int) f`.
- If `f` is a long, compute `(int) (f ^ (f >>> 32))`
- If `f` is a float, compute `Float.floatToIntBits(f)`
- If `f` is a double, compute `Double.doubleToLongBits(f)` which produces a long `f1`, then return `(int) (f1 ^ (f1 >>> 32))`
- If `f` is an object, compute `f.hashCode()`

## Formula:

**Step 1.** Use the table above to produce a temporary hash of each variable in your class.

*Example:* You have variables `u`, `v`, `w`. Produce (using the chart above) temporary hash vals `hash_u`, `hash_v`, `hash_w`.

**Step 2.** Combine these temporary hashes into a final `hashCode` that is to be returned

*Example:*

```
int result = 17;  
result += 31 * result + hash_u;  
result += 31 * result + hash_v;  
result += 31 * result + hash_w;  
return result;
```

# Review: Making Your Classes Immutable

1. A class is immutable if the data it stores cannot be modified once it is initialized. Java's String and number classes (such as Integer, Double, BigInteger) are immutable. Immutable classes provide good building blocks for creating more complex objects. **Java 8:** LocalDate, as we saw earlier, is also immutable.
2. Immutable classes tend to be smaller and focused (building blocks for more complex behavior). If many instances are needed, a “mutable companion” should also be created (for example, the mutable companion for String is StringBuilder) to handle the multiplicity without hindering performance.
3. Guidelines for creating an immutable class (from *Effective Java*, 2<sup>nd</sup> ed.)
  - **All fields should be *private* and *final*.** This keeps internals private and prevents data from changing once the object is created.
  - **Provide *getters* but no *setters* for all fields.** Not providing setters is essential for making the class immutable.
  - **Make the class *final*.** (This prevents users of the class from accessing the internals of the class in another way – to be discussed in Lesson 6.)
  - **Make sure that getters do not return mutable objects.**

# Immutable Classes

## Mutable

```
public class IntHolder {  
    private int value;  
    public IntHolder(int value) {  
        this.value = value;  
    }  
    public int getValue() {  
        return value;  
    }  
    public String SetValue(int value) {  
        this.value = value;  
    }  
}  
// main method  
IntHolder holder = new IntHolder(10);  
int v = holder.getValue(); // return 10  
Holder.SetValue(20);  
v = holder.getValue(); // return 20
```

## Immutable

```
final public class IntHolder{  
    private final int value;  
    public IntHolder(int value) {  
        this.value = value;  
    }  
    public int getValue() {  
        return value;  
    }  
}  
// main method  
IntHolder holder = new IntHolder(10);  
int v = holder.getValue(); // return 10
```



## Main Point 2

All classes in Java belong to the inheritance hierarchy headed by the Object class.

Likewise, all individual consciousness inherit from the single unified field.

# Quiz 2

1. What is the return type of hashCode() method?

- a. boolean      b. int      c. String

# Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. Inheritance in Java makes it possible for a subclass to enjoy (and re-use) the features of a superclass.
2. All classes in Java – even user defined classes – automatically inherit from the class Object
3. ***Transcendental Consciousness*** is the field of pure awareness, beyond the active thinking level, that is the birthright and essential nature of everyone. Everyone “inherits” from pure consciousness
4. ***Wholeness moving within itself***: In Unity Consciousness, there is an even deeper realization: The only data and behavior that exist in the universe is that which is “inherited from” pure consciousness – everything in that state is seen as the play of one’s own consciousness.

