While the Java was designed to be straightforward and easy-to-use, there are a number of language and API features that, when not used correctly, can yield unexpected results. In this lecture we examine some of the features and discuss a number of Java programming “best practices” distilled from Josh Bloch’s *Effective Java Programming* book.

Java Tips and Tricks

- Methods and inheritance

- Numbers

- Exceptions

Copyright ©2003-2004 by David M. Whitlock. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from whitlock@cs.pdx.edu. You the man, Neal!
Java Trick #1

Consider the following program:

class Dog {
    public static void bark() {
        System.out.println("woof");
    }
}
class Collie extends Dog {
    public static void bark() {
    }
}
public class Barks {
    public static void main(String[] args) {
        Dog rover = new Dog();
        Dog lassie = new Collie();
        rover.bark();
        lassie.bark();
    }
}

What does it print?

a) woof

b) woof woof

c) It varies
Java Trick #1

The answer is

b) woof woof

There is no dynamic dispatch on static methods

- The decision as to which method to invoke is made at **compile time**

- Compile-time type of both rover and lassie is Dog

- So, Dog.bark() will be invoked

- Collie.bark() does not override Dog.bark(), it *hides* it (not a good thing to do)

- Be careful not to invoke static methods on instances:
  
  Dog.bark()

  not

  rover.bark()
Consider the following program:

class Indecisive {
    public static void main(String[] args) {
        System.out.println(waffle());
    }

    static boolean waffle() {
        try {
            return true;
        } finally {
            return false;
        }
    }
}

What does it print?

a) true

b) false

c) None of the above
Java Trick #2

The answer is:

b) false

The finally block is always processed after the try

- Be careful of what you do in a finally block
- Don’t do things that effect control flow (return, continue, throw, etc.)
Java Trick #3

Consider the following program:

```java
public class SordidSort {
    public static void main(String args[]) {
        Integer big = new Integer(2000000000);
        Integer small = new Integer(-2000000000);
        Integer zero = new Integer(0);
        Integer[] arr =
            new Integer[] {big, small, zero};
        Arrays.sort(arr, new Comparator() {
            public int compare(Object o1, Object o2) {
                return ((Integer) o2).intValue() -
                        ((Integer) o1).intValue();
            }
        });
        System.out.println(Arrays.asList(arr));
    }
}
```

What does it print?

a) [-2000000000, 0, 2000000000]

b) [2000000000, 0, -2000000000]

c) [-2000000000, 2000000000, 0]

d) It varies
Java Trick #3

The answer is:

d) It varies

The behavior is of the Comparator is undefined

- It uses int subtraction, but the difference of the Integers exceeds 32 bits
- Remember that ints are not integers
- You can get overflow or underflow
- Don’t be too clever

A better Comparator:

```java
public int compare(Object o1, Object o2) {
    int i1 = ((Integer) o1).intValue();
    int i2 = ((Integer) o2).intValue();
    return (i2 < i1 ? -1 : (i2 == i1 ? 0 : 1));
}
```
Tip #1: Consider factory method instead of constructor

Some classes provide a factory method instead of (or in addition to) a constructor:

```java
public static Student getStudent(String name) { ... }
```

Advantages:

- They don’t **have** to create a new object
  - Instances can be pooled or a *canonical* instance can be used
    ```java
    public static Boolean valueOf(boolean b) {
      return b ? Boolean.TRUE : Boolean.FALSE;
    }
    ```
  - May return an instance of a subclass

- Factory methods have names (`getInstance` or `valueOf(int)`, `valueOf(double)`)

Disadvantages:

- Classes without a public or protected constructor cannot be subclassed
- Not easily distinguishable from other static methods
Tip #2: Private constructors enforce singletons

The *singleton* design pattern ensures that only one instance of a class is ever created

- Enforced with a private constructor
- Usually has a factory method

```java
public class Database {
    /** The one and only instance */
    private static Database singleton = new Database();

    /** Factory method */
    public static Database getInstance() {
        return singleton;
    }

    /** Gets a piece of info from the database */
    public Data getData(String key) { ... }
}
```
Tip #3: Minimize the scope of local variables

Older programming languages like C and FORTRAN require that all local variables must be declared before the first statement

- Can be error prone – accidentally reuse a variable where you didn’t intend
- Can clutter up code, especially longer functions

In Java, you can declare a variable anywhere a statement is legal

- Declare a variable where it is first used
- If the variable is only used inside a code block (\{\ldots\}\), then it should be declared in that block
- Always try to initialize a local variable when you declare it
  - If you can’t compute its initial value, maybe you’re not ready to declare it yet
  - Java requires that you must initialize a local variable before you use it
Tip #3: Minimize the scope of local variables

It’s often a good idea to use loop variables

- Use for loops instead of while

Consider the following erroneous code:

```java
Iterator iter = coll.iterator();
while (iter.hasNext()) {
    // do something
}

Iterator iter2 = coll2.iterator();
while (iter.hasNext()) { // type-o!
    // do something
}
```

You can prevent this kind of cut-and-paste error by using a for loop:

```java
for (Iterator iter = coll.iterator();
    iter.hasNext(); ) {
    // do something
}
for (Iterator iter2 = coll2.iterator();
    iter2.hasNext(); ) {
    // do something
}
```
Tip #3: Minimize the scope of local variables

If you had tried to use `iter` in the second `for` loop, the code would not have compiled.

Remember that Java allows you to have multiple statements in the initialized of a for loop:

```java
Object[] objects = ...
for (int i = 0, int size = objects.length;
     i < size; i++) {
    // Process objects[i]
}
```

Minimizing the scope of local variables keeps methods small and focused.
Java Trick #4

Consider the following program:

```java
class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```

What does it print?

a) 0.9  
b) 0.90  
c) It varies  
d) None of the above
Java Trick #4

The answer is:

d) None of the above

The program prints 0.8999999999999999

- floats and doubles do not represent decimal numbers exactly
- ints and longs do represent integral numbers exactly
- If you want an exact representation you can use the java.math.BigDecimal class

Using arbitrary precision

```java
import java.math.BigDecimal;
public class Change2 {
    public static void main(String args[]) {
        System.out.println(
            new BigDecimal("2.00").subtract(
                new BigDecimal("1.10")));
    }
}
```

Note that BigDecimals are String-based
Tip #4: Avoid float and double if exact answers are required

float and double perform binary floating-point arithmetic

- Only provide approximations of the results of arithmetic operations
- Because they are binary, they cannot exactly represent negative powers of ten (like 0.1)
- They shouldn’t be used for representing monetary values

```java
package edu.pdx.cs399J.tips;

public class DoubleTrouble {
    public static void main(String[] args) {
        double increment = 0.10;
        double total = 0.0;
        for (int i = 0; i < 10; i++) {
            System.out.println(total);
            total += increment;
        }
        System.out.println(total
            + (total == 1.0 ? "\nYes" : "\nNo?");
    }
}
```
Tip #4: Avoid float and double if exact answers are required

$ java -cp classes edu.---.DoubleTrouble
0.0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
No?
Tip #4: Avoid float and double if exact answers are required

java.math.BigDecimal provides an decimal value with arbitrary precision

```java
package edu.pdx.cs399J.tips;
import java.math.BigDecimal;

public class BigDecimalDemo {
    public static void main(String[] args) {
        BigDecimal increment = new BigDecimal("0.10");
        BigDecimal total = new BigDecimal("0.0");
        for (int i = 0; i < 10; i++) {
            System.out.println(total);
            total = total.add(increment);
        }
        BigDecimal one = new BigDecimal("1.00");
        System.out.println(total +
                            (total.equals(one) ? "\nYes" : "\nNo?")));
    }
}
```
Tip #4: Avoid float and double if exact answers are required

```
$ java -cp classes edu.---.BigDecimalDemo
0.0
0.10
0.20
0.30
0.40
0.50
0.60
0.70
0.80
0.90
1.00
Yes
```

Using BigDecimals provides arbitrary-precision arithmetic, but

- Not as convenient as using doubles
- Slower than using primitive types

You can also use ints and longs

- Multiple by 100 to get cents of a dollar
- However, it only gives you 18 or 19 digits
Consider the following program:

```java
public class Trivial {
    public static void main(String args[]) {
        System.out.print("H" + "a");
        System.out.print(\'H\' + \'a\');
    }
}
```

What does it print?

a) HaHa

b) Ha

c) None of the above
Java Trick #5

The answer is:

c) None of the above

This program really prints \texttt{Ha169}

- \texttt{'H' + 'a'} evaluates to an \texttt{int} which is then converted to a \texttt{String}

- Remember that in order for string concatenation to occur at least one of the operands has to be a \texttt{String}

- Use \texttt{String.valueOf()} to convert primitive types to a \texttt{String}

- This is one reason why operator overloading is dangerous
Consider the following program:

```java
public class Confusing {
    public Confusing(Object o) {
        System.out.println("Object");
    }
    public Confusing(double[] dArray) {
        System.out.println("double array");
    }
    public static void main(String args[]) {
        new Confusing(null);
    }
}
```

What does it print?

a) Object
b) double array
c) None of the above
Java Trick #6

The answer is:

b) double array

When multiple overloadings apply, the method with most specific argument types wins

• Otherwise, it won’t compile

• If you want to be explicit, cast:

  new Confusing((Object) null);

• Avoid ambiguity when overloading

• If you do have ambiguous overloads, make sure that their behavior is identical
Tip #5: Use String concatenation wisely

In Java the + operator is overloaded to concatenate a String and another piece of data

- A new StringBuffer object is created for every concatenation!
- Using string contention in a loop can be expensive

```java
public String formatResults(double[] results) {
    String s = "";
    for (int i = 0; i < results.length; i++)
        s += results[i] + "\n";
    return s;
}
```

is really:

```java
public String formatResults(double[] results) {
    String s = "";
    for (int i = 0; i < results.length; i++) {
        StringBuffer sb1 = new StringBuffer(s);
        sb1.append(results[i]);
        sb1.append("\n");
        StringBuffer sb2 = new StringBuffer(s);
        sb2.append(sb1.toString());
        s = sb2.toString();
    }
    return s; }
```
Tip #5: Use String concatenation wisely

When you are building a large string in a loop, use StringBuffer explicitly:

```java
public String formatResults(double[] results) {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < results.length; i++) {
        sb.append(results[i]);
        sb.append("\n");
    }
    return sb.toString;
}
```
Tip #6: Only use exceptions for exceptional conditions

Never do this:

```java
public static void main(String[] args) {
    try {
        for (int i = 0; true; i++) {
            String arg = args[i];
            // ...
        }
    } catch (ArrayIndexOutOfBoundsException ex) {
        // All done
    }
}
```

Why is this bad?

- Doesn’t adhere to the standard Java programming idiom:

  ```java
  for (int i = 0; i < args.length; i++) {
      String arg = args[i];
  }
  ```

  - Hard to read

- Inefficient because JVMs don’t optimize exception throwing
Tip #6: Only use exceptions for exceptional conditions

Some guidelines:

- Exceptions should never be used for ordinary control flow

- An API should allow the user to “look before he leaps”
  - If a “state dependent” method may throw exception, there should be a testing method
  - Like hasNext() in Iterator

Don’t do this, either:

```java
try {
    Iterator iter = coll.iterator();
    while (true) {
        Object o = iter.next();
        // ...
    }
}

} catch (NoSuchElementException ex) {
    // All done...
}
```
Tip #7: Checked exceptions vs. run-time exceptions

Checked exceptions should be used for recoverable conditions

- Caller is forced to catch it or rethrow it

Unchecked exceptions should indicate programming errors

- Often thrown when a some “precondition” of invoking a method is violated

- Caller did not conform to the contract (IllegalArgumentException, IllegalStateException)

Only the JVM should throw Errors

- A user’s unchecked exception should be a RuntimeException
Tip #8: Avoid unnecessary use of checked exceptions

Checked exceptions force the caller to deal with exceptional conditions

- Lots of checked exceptions can make an API cumbersome

Unchecked exceptions may be appropriate when:

- If the exceptional condition can be prevented by proper use

- There is no reasonable way to handle the exception
Tip #8: Avoid unnecessary use of checked exceptions

Try breaking the method that may throw the checked exception into a test method and a method that throws an unchecked exception:

```java
if (action.canDoIt())
    action.doIt();
else
    // Handle exceptional condition...
```

Instead of

```java
try {
    action.doIt();
} catch (SomeCheckedException ex) {
    // Handle exception condition...
}
```

A “state-testing method” may not be appropriate if

- State can be changed concurrently by another thread
- The `doIt` method has to re-check a lot of state
Tip #9: Use standard exception types

Consider using these standard exceptions (from java.lang and java.util) in your APIs

- **IllegalArgumentException**: Caller passed a parameter with an invalid value (e.g. negative iteration count)

- **IllegalStateException**: The method cannot perform its work given the receiver object’s state (e.g. object hasn’t been initialized)

- **NullPointerException**: Caller passes a parameter with an invalid null value (distinguishes it from IllegalArgumentException)

- **ConcurrentModificationException**: An object designed for single-threaded access has been concurrently modified

- **UnsupportedOperationException**: The class doesn’t implement one of its methods (rarely used, but good during development)
Tip #10: Document all exceptions thrown by a method

Always document each exception (checked or unchecked) with an @throws tag

- The exceptions a method can throw are part of its contract
- Be specific about the conditions under which an exception can be thrown
- In the throws clause use specific type instead of superclasses

```java
/**
 * Never do this!
 *
 * @throws Throwable Something goes wrong
 */
public void doStuff() throws Throwable {
    
}
```

- Documentating exceptions helps establish the preconditions for a method
- Don’t declare that a method throws an unchecked exception – caller probably does not want to catch it
Tip #11: Strive for failure atomicity

If a method throws an exception, the object should still be in a legal state

- Ideally, the object should be in the same state as it was before the method was invoked

- Failures should be “atomic”:
  - Either the operation succeeded and the change was made
  - OR, the operation failed and no changes were made

- Check the validity of parameters before doing work

- Do work that might fail before modifying the object

- You can also have “recovery code” that restores the state or work with a copy of the old state

- While atomic failures are nice, they are not always possible or desirable
Tip #12: Don’t ignore exceptions

Exceptions are thrown for a reason, don’t ignore them.

```java
try {
    doSomething();
} catch (SomeException ex) { }
```

Even if it’s okay to catch the exception, at least document why you ignored it.

If an exception should never happen, be explicit about it.

```java
try {
    ByteArrayInputStream bais = ... 
    ObjectInputStream ois =
        new ObjectInputStream(bais);
    Object o = ois.readObject();

} catch (IOException ex) {
    // Reading from memory should never cause
    // an IOException
    assert false : "Shouldn’t get here";
}
```
Consider the following program:

class ByteMe {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++) {
            if (b == 0x90)
                System.out.print("Byte me! ");
        }
    }
}

What does it print?

a) (nothing)

b) Byte me!

c) Byte me!  Byte me!
Java Trick #7

The answer is:

a) (nothing)

The constant 0x90 is an int

- 0x90 is 144 which is larger than Byte.MAX_VALUE
- When the comparison is done, b is \textit{promoted} to an int
- You could cast b to a byte
  
  \begin{verbatim}
  if (b == (byte) 0x90)
  \end{verbatim}

  (byte) 0x90 == -112 which is greater than Byte.MIN_VALUE

- You could mask off the high-order bits (note the masking operation promotes b)
  
  \begin{verbatim}
  if ((b & 0xff) == 0x90)
  \end{verbatim}

- Remember that bytes aren’t ints

- Remember that integral constants are \textit{ints} and decimal constants are \textit{doubles}
Tip #13: Obey the contract when overriding equals

Sometime you don’t need to override equals

- Each instance of a class is inherently unique (e.g. Thread)

- It makes no sense to compare instances e.g. Random

- A superclass has implemented equals and the subclass has not added any interesting state

- Class is package-private and you are certain equals will never be invoked
  - Should probably override equals to throw an UnsupportedOperationException
Tip #13: Obey the contract when overriding equals

The `equals` contract provides an *equivalence relation* that is:

- **reflective**: `x.equals(x) == true`

- **symmetric**: `x.equals(y)` implies `y.equals(x)`

- **transitive**: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`

- **consistent**: if `x.equals(y)`, `x` will **always** equal `y`, unless the relevant contents of `x` or `y` changes

- **non-nullity**: if `x` is non-null, then `x.equals(null) == false`
**Tip #13: Obey the contract when overriding `equals`**

A template for a `equals(Object o)` method:

1. If `o == this`, return `true`

2. Use `instanceof` to determine whether or not `o` is of the correct type

3. Cast `o` to the correct type

4. Compare each field of the casted `o` to the fields in `this`
   - If field is an `Object`, recursively invoke `equals` methods
   - If field is a `float` or `double`, compare their bytes using `Float.floatToIntBits()` and `Double.doubleToLongBits()`
   - If field is another primitive type, compare with `==`
   - If field is an array, compare each element of the arrays
   - To handle null fields: `(field == null ? o.field == null : field.equals(o.field))`
Consider the following program:

```java
public class Name {
    private String first, last;
    public Name(String first, String last) {
        this.first = first;
        this.last = last;
    }
    public boolean equals(Object o) {
        if (!(o instanceof Name)) return false;
        Name n = (Name)o;
        return n.first.equals(first) &&
                n.last.equals(last);
    }
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add(new Name("Jeff", "Sinclair");
        System.out.println(s.contains(new Name("Jeff", "Sinclair")));
    }
}
```

What does it print?

a) true
b) false
c) It varies
Java Trick #8

The answer is:

c) It varies

Even though Jeff is in the set, `contains` doesn’t necessarily find him

- Remember that a `HashSet` is backed by a `HashMap`
- `Name` overrides `equals`, but doesn’t override `hashCode`
- It is unlikely that two instances of `Name` will have the same hash code
- Remember to always override `hashCode` when you override `equals`

A good `hashCode` implementation would be:

```java
public int hashCode() {
    return 31 * first.hashCode() + last.hashCode();
}
```
Tip #14: Always override `hashCode` when you override `equals`  

The `hashCode` contract:

- Multiple calls to `hashCode` must return the same value unless the state used for `equals` is modified.

- If two objects are `equals`, their `hashCode` must be the same.

- While not a requirement, distinct hash codes for unequal objects may result in better hashing.
Tip #14: Always override `hashCode` when you override `equals`

While an ideal hashing algorithm can be complex, here's a template `hashCode` method that works pretty well:

1. Build an int `result` starting with small prime number like 17

2. Compute a hash code, `c`, for each field `f` used to compute `equals`
   - boolean: `c = (f ? 0 : 1)`
   - char, byte, short, int: `c = (int) f`
   - long: `c = (int) (f ^ (f >>> 32))`
   - double: Convert to long with `Double.doubleToLongBits(f)`, then above
   - Object: `c = (f == null ? 0 : f.hashCode())`

   • Combine `f`’s hash code into `result`:
     `result = (37 * result) + c`

3. Return `result`
Tip #14: Always override `hashCode` when you override `equals`.

If the computation of the hash code is expensive, you may want to consider caching it in an instance field:

```java
public class CachedHash {
    private int id;
    private double gpa;
    private String name;
    private transient int hashCode = 0;

    public boolean equals(Object o) { ... }

    public int hashCode() {
        if (hashCode == 0) {
            int result = 17;
            int c = (int) id;
            result = 37*result + c;
            long l = Double.doubleToLongBits(gpa);
            c = (int) (l ^ (l >>> 32));
            result = 37*result + c;
            c = (name == null ? 0 : name.hashCode());
            result = 37*result + c;

            hashCode = result;
        }
        return hashCode;
    }
}
```
Consider the following program:

class Base {
    public String name = "Base";
}
class Derived extends Base {
    private String name = "Derived";
}

public class PrivateMatter {
    public static void main(String[] args) {
        System.out.println(new Derived().name);
    }
}

What does it print?

a) Derived
b) Base
c) Won't compile: Can't assign weaker access to name
d) None of the above
Java Trick #9

The answer is:

d) None of the above

The program won’t compile because `PrivateMatter` cannot access the private `name` field

- The private `name` field *hides* the public `name` field
- Remember: you cannot “override” fields like you do methods
- Instead of using a public field, you should use a public accessor method
Java Trick #10

Consider the following program:

```java
public class Truth {
    public static void main(String args[]) {
        new Foo();
    }
}

class Foo {
    static Bar b = new Bar();

    static boolean truth() { return true; }
    static final boolean TRUTH = truth();

    Foo() {
        System.out.println("The truth is: " + TRUTH);
    }
}

class Bar extends Foo { }

This program prints:

The truth is: false
The truth is: true
```
Java Trick #10

But TRUTH is a final field, does its value change?

- Yes, TRUTH is referenced before it is initialized
- When main creates a new Foo, Foo's class is initialized
- Foo's static field b is initialized, a new instance of Bar is created
- Creating a Bar causes Bar's class to be initialized
- When Bar is initialized its superclass, Foo, is initialized
- But, we're already in the process of initializing Foo
- Bar's call to its inherited constructor references the TRUTH field before the Foo class has been completely initialized
- Then after Foo's class has been completely initialized, Foo's constructor is run and the initialized TRUTH is printed
Summary

Java can be tricky!

- Even though Java is a straightforward language, there are some dark corners you can get lost in

- Computers represent numbers in a finite number of bits – be careful of overflow/underflow

- You have to be mindful of the rules when you override methods, invoke static initializers, and the like

- Don’t try to be too clever – you might out-smart yourself!