Review

Java Trivia

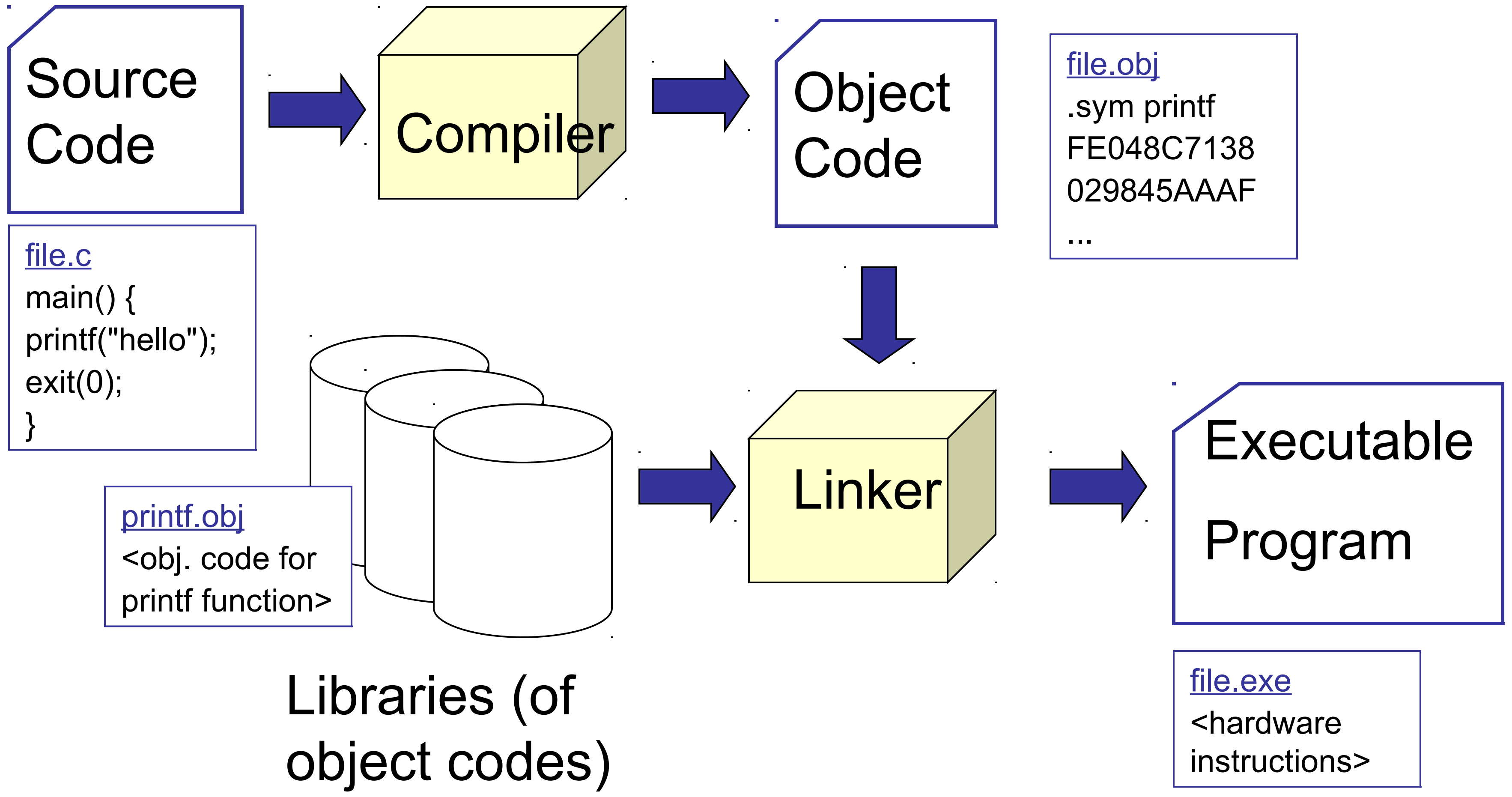
- What is the command to *compile* a Java source file named "Hello.java" ?

```
ubuntu> javac Hello.java
```

- What is the command to *execute* a Java class file named "Hello.class" ?

```
ubuntu> java Hello
```

Compiling a Program in C or C++



file.c
```
main() {
printf("hello");
exit(0);
}
```

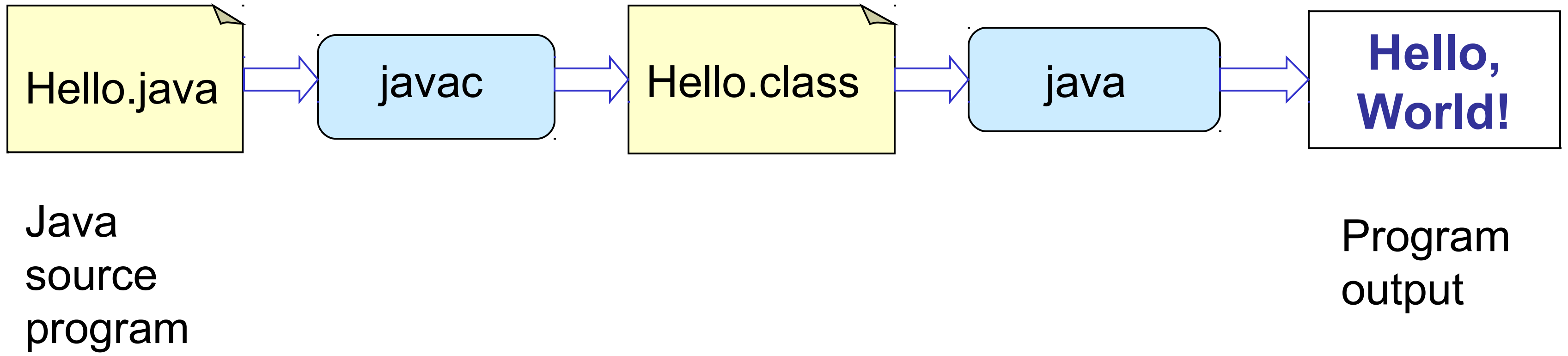
file.obj
```
.sym printf
FE048C7138
029845AAAF
...
```

printf.obj
<obj. code for printf function>

file.exe
<hardware instructions>

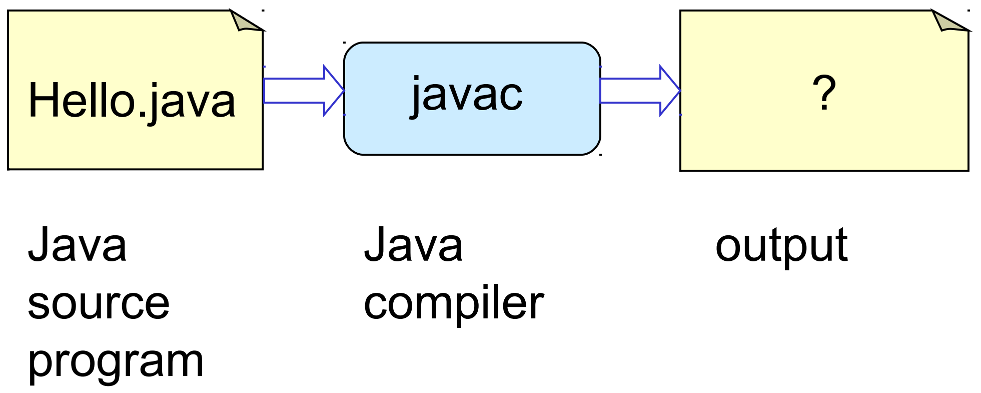
Explain (to your mother) how java works

- What does `javac` do?
- What does `java` do?
- How is Java different from C++ or C#?



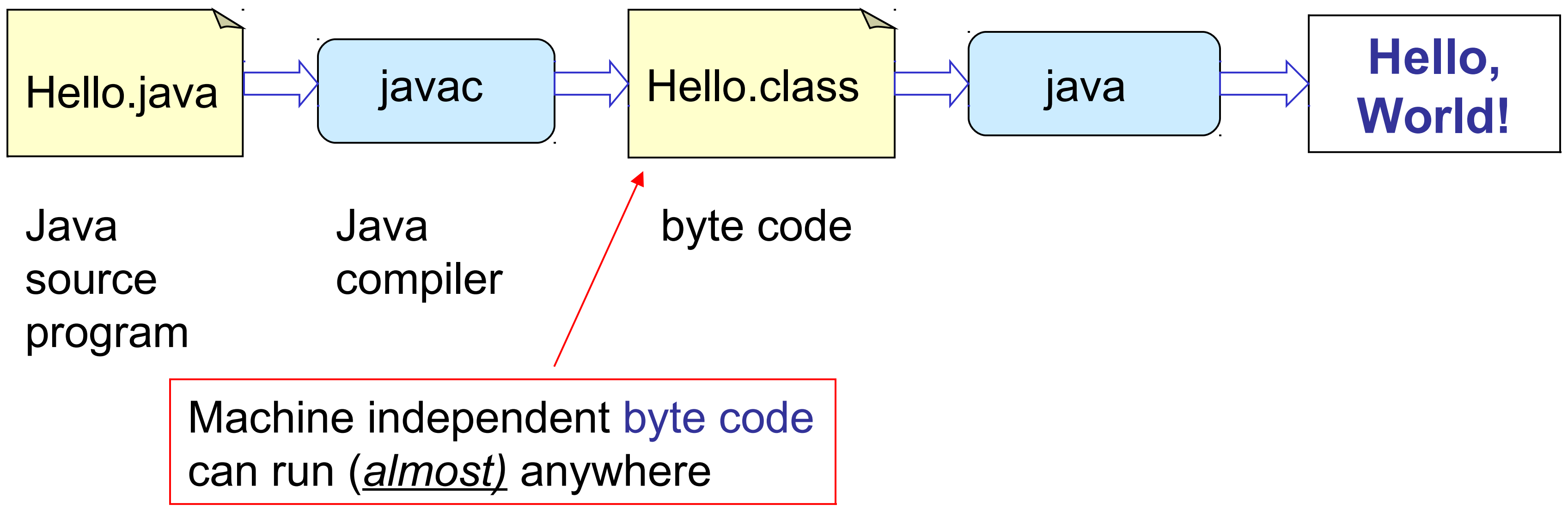
What is the output of javac?

- What is the output of javac?
- What is actually in the file?
- What hardware will it run on?



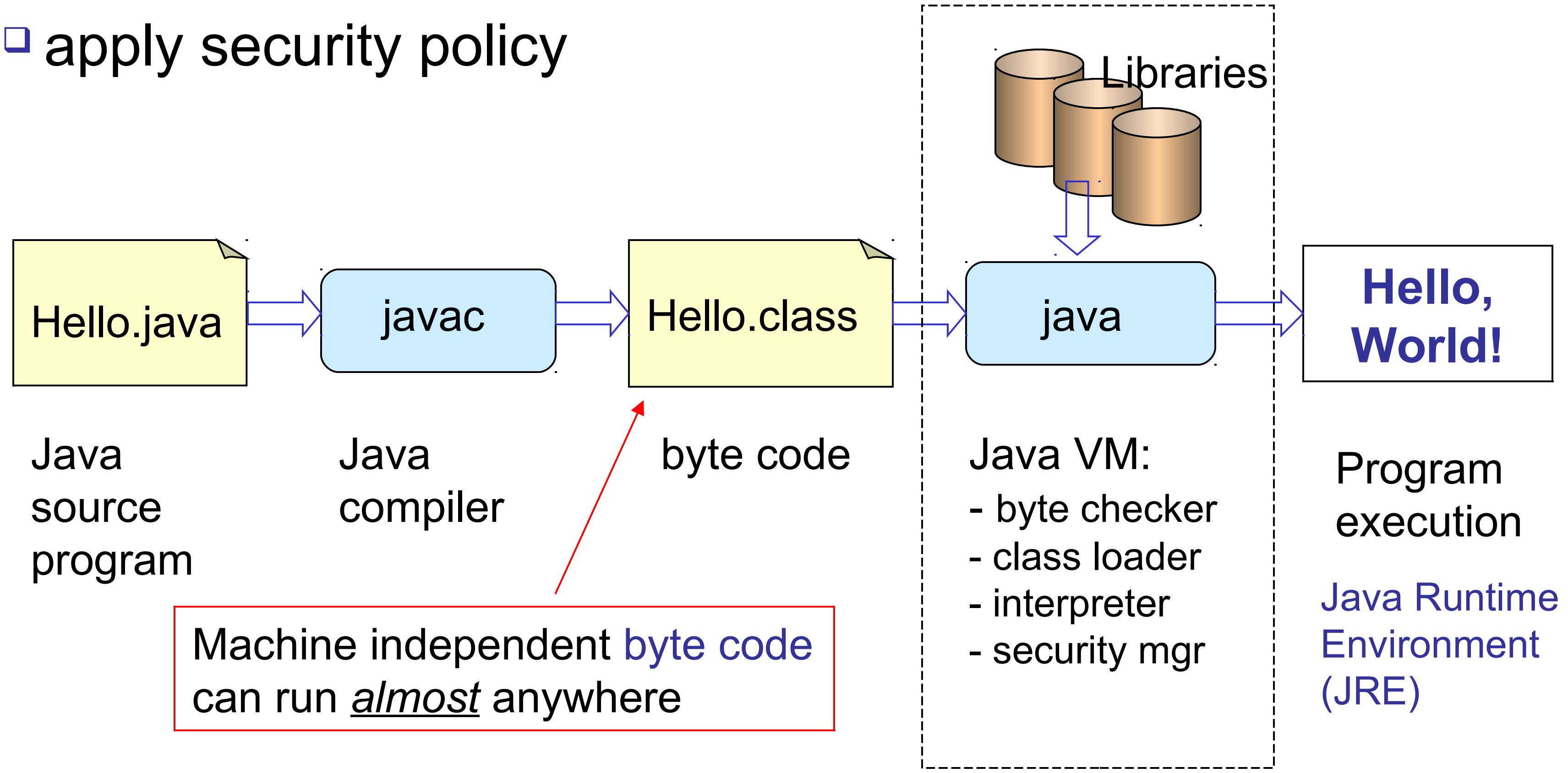
What does `java` command do?

- javac creates *Java byte code* for a virtual machine (a C compiler creates *machine code* for real hardware)
- How can we possibly run this byte code?



java starts the Java Virtual Machine

- verify byte code and version
- resolve classes (classpath) and load them (classloader)
- adapt to real hardware
- apply security policy



Machine independent byte code can run *almost* anywhere

Import

- What does "import java.util.Scanner" do?

1. include the Scanner class in compiled program.
2. include Scanner source code in this program
3. add java.util.Scanner to CLASSPATH
4. add java.util.Scanner to current name space

Import

- The "import" command is processed by (choose one):
  1. Java compiler
  2. Java interpreter (Java VM)
  3. Java program editor or IDE

- "`import X.Y`" is like what statement in C# ?
  1. `using X.Y;`
  2. `namespace X.Y`
  3. `include "X.Y"`

Locations

- What is "**`java.lang`**" ?
- Name some classes in "**`java.lang`**"
- (True/False) You should always "**`import java.lang.*`**" so your program can use the classes in **`java.lang`**.

More import

(True/False)
The command "**`import java.util.*`**" does...

- includes the code for all classes in java.util into your program
- adds all classes in **`java.util`** to the name space
- makes the compiled program larger

Ordering

In a Java program, which of these should come first? second? third?

```
a)    import stuff;
b)    package packagename;
c)    /**
       * Javadoc comment for this class.
       * @author Fatalai Jon
       */
d)    public class MyClass { ... }
```

More import

- Java has 2 Date classes: java.util.Date & java.sql.Date. Which Date class will be used here?

```
import java.util.*;
import java.sql.*;
public class Test {
   Date today = new Date( );
   ...etc...

}
```

Answers:

1. java.util.Date because java.util is imported first.
2. java.sql.Date because java.sql was imported last.
3. implementation dependent (can be either one).
4. neither - compiler will raise an error

No Ambiguity Allowed

- If there are 2 or more classes with the same name in the list of imports, the compiler issues an error.
- No error if you *exactly* specify the class name on the import command.

```
import java.util.*;  }  ambiguous (not clear)
import java.sql.*;   }
import java.util.Date; // specify the Date class
public class Test {
  Date today = new Date( );
  ...etc...

}
```

No import

- How can you use Date without an "import" ?

```
// NO imports
public class Test {
    java.util.Date today = new java.util.Date( );
   ...etc...
}
```

No import (answer)

- Write the complete path.

```
public class Test {
   java.util.Date today = new java.util.Date( );
   ...etc...

}
```

Fully Qualified Class Name is also necessary when loading classes at runtime. For example:

```
Class cl = Class.forName( "java.util.Date" );
     // load the class

Date now = (Date) cl.newInstance( ); // new object
```

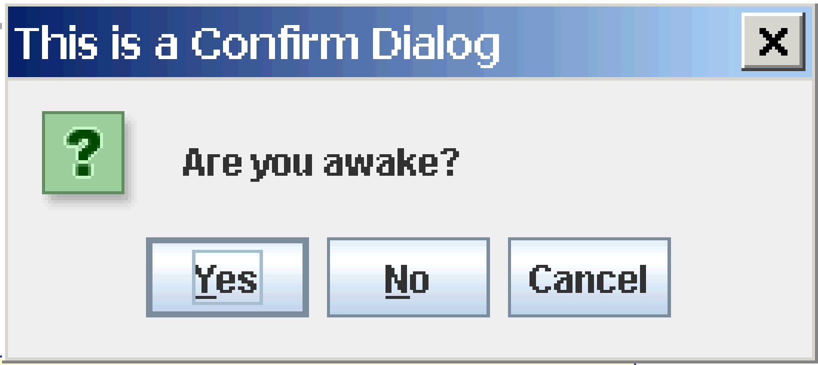
What is "import static" ?

- What does "**import static** ... " mean?

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
import static java.lang.Math.sqrt;

public class Circle {
  double radius;  // radius of circle
  /** get area of circle */
  public double area() {
    return PI * pow(radius,2);
  }
```

import static JOptionPane



```
import static javax.swing.JOptionPane.*;

public class WakeUp {
    ...
    int choice;
    choice = showConfirmDialog( null,
        "Are you awake?","This is a Confirm Dialog",
        YES_NO_CANCEL_OPTION );

    if ( choice == NO_OPTION )
    showMessageDialog( null, "Liar!!");
```

Static block

- What does `static { ... }` mean?

```
public class Point {
   static Map<String,Integer> nums = ???

   static {
      nums = new HashMap<String,Integer>();
      nums.add("one", 1);
      nums.add("two", 2);
   }
  public Point() {
```

Constructors

- What is the purpose of a constructor?

1. create an object
2. allocate memory for a new object
3. initialize attributes or state of a new object

- Can a class have <u>no</u> constructors?
- Can a class have <u>more than one</u> constructor? How?
- Can one constructor call another constructor?
  - If so, how?

Constructors

```
public class Fraction {
     private long numerator;
     private long denominator;
    /** a new Fraction = num/denom */
    public Fraction( long num, long denom ) {
           long gcd = gcd(num, denom);
           this.numerator = num/gcd;
           this.denominator = denom/gcd;
    }
    /** a new Fraction with integer value */
    public Fraction( long numerator ) {
           this( numerator, 1L );
    }
```

Constructors (3)

- What is wrong here?

```
public class Fraction {
   /** construct a new Fraction object */
   public Fraction( long num, long denom ) {
      /* do the real work here */
      ...
   }
   /** constructor makes fraction from a double */
   public Fraction(double x) {
      if ( Double.isNaN(x) ) this( 0L, 0L );
      else if ( Double.isInfinite(x) )
           this( 1L, 0L );
      else ...
   }
```

What will be printed?

```
public class Greeter {
   String name;
   static { // static initialization block
        System.out.println("Static block");
   }

   {    // dynamic initialization block
        System.out.println("Anonymous block");
   }

   public Greeter(String name) {
        System.out.printf("Greeter for "+name);
   }
```

What will be printed? (2)

```
public static void main(String [] args) {

     Greeter john = new Greeter( "John" );

     Greeter nok = new Greeter( "Nok" );

}
```

The Three Noble Truths

- What are the three pillars (key characteristics) of object-oriented programming?

  Encapsulation: an object contains both data and the methods that operate on the data. It may expose some of these to the outside and hide others.
  This design separates the *public interface* from the *implementation*, and enforces data integrity.

  Inheritance: one class can inherit attributes and methods from another class.

  Polymorphism: the operation performed by a named method can depend on context. In particular, it can depend on the type of object it is applied to.

Immutable Objects

- What does it mean for an object to be immutable?
- Does this class define immutable objects?

```
public class Appointment {
   private Date date;
   private String description;
   public Appointment( Date when, String what ) {
      date = when;
      description = what;
   }
   public Date getDate( ) { return date; }
   public String getDescription { return description; }
}
```

No mutator methods!

Not Really Immutable

- You can change the appointment date.
- How? (Two ways)

```
public class Appointment {
   private Date date;
   private String description;
   public Appointment( Date when, String what ) {
      date = when;
      description = what;
   }
   public Date getDate( ) { return date; }
   public String getDescription { return description; }
}
```

Breaking Encapsulation...

```
Date date = new Date(2015-1900, Calendar.MARCH, 18);
Appointment exam =
             new Appointment(date, "OOP Midterm");

// change the exam date to April 1.
date.setMonth( Calendar.APRIL );
date.setDate( 1 );

// I'm still not ready.  Postpone exam.
exam.getDate( ).setMonth( Calendar.JUNE );
```

Encapsulation and Mutability

- A Date object is mutable.
- If you copy a *reference* to a mutable object, it can break encapsulation.
- If you return a *reference* to a mutable attribute, it breaks encapsulation.

```
class Person {
   private String name;
   private Date birthday;
   public Person(String name, Date birthday)
   {    this.birthday = birthday;    // broken
   public Date getBirthday() {
        return birthday;             // broken
   }
```

Really Immutable

For strong encapsulation, do this:

1) Copy or clone references to mutable objects.

2) Don't return a reference to an immutable object.

```
public class Appointment {
   private Date date;
   private String description;
   public Appointment( Date when, String what ) {
      date = (Date) when.clone();
      description = what;
   }
   public Date getDate( ) { return new Date(date); }
   public String getDescription { return description; }
}
```

Lists and Arrays are Mutable

- List and array are mutable.
- Note that sometimes returning a reference is required (Persistence frameworks need to get/set attributes).

```
public class Purse {
    private List<Valuable> items;


    /** Get items in the purse. */
    public List<Valuable> getItems( ) {
        return items;
    }
```

An Immutable List (or Set or Map)

- Collections can "wrap" your List in an Unmodifiable List
- This is example of the *Decorator* Design Pattern.

```
public class Purse {
    private List<Valuable> items;

    /** Get items in the purse. */
    public List<Valuable> getItems( ) {
      return
       Collections.unmodifiableList(items);
    }
```

UnmodifiableList is a View (Wrapper)

- Its a wrapper (decorator) not a copy!
- Prove it!

a) create an Unmodifiable wrapper for a List

b) modify the underlying list

```
> List<String> food = new ArrayList<>( );
> food.add("apple"); food.add("banana");
> List<String> copy =
        Collections.unmodifiableList(food);
> copy.size()
2
> food.add( "cake" );
> copy.get(2)    // returns "cake" !
```

What's the Difference?

- java.util.*Collection*
- java.util.Collections - utility methods for collections

By analogy, what is the purpose of this?

java.util.Arrays

5 Criteria for a Good Class Interface

- (High) Cohesion - all the methods are related to one purpose.

  Example: all methods of a Stack are related to managing a stack.

- Clarity - purpose of the interface should be easy to understand.

  Example:

  push(), pop(), peek(), isFull( )

  Bad Example:

- Q: What about Coupling ?

  A: coupling is a property of the implementation