4. Java Classes and Objects

• Data encapsulation in objects:
  - data (attributes)
  - methods (behaviors)

• Objects hide information:
  - private data/methods
  - public interfaces

• Action-oriented (procedural) vs. object-oriented
Use access control modifiers to hide information from the clients

including:
Constructors
Accessors
Mutators
The Time Class

• Class definition establishes the data unit and desired functions

• A driver exercises the object by instantiating, initializing, and invoking all methods

• The driver/object model can be used to design object units independently during the prototype/implementation phase of a project
public class Time1 extends Object
{
    private int hour;    // 0-23
    private int minute;  // 0-59
    private int second;  // 0-59

    public Time1( )
    {
        setTime( 0, 0, 0 );
    }

    public void setTime( int h, int m, int s )
    {
        hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
        minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
        second = ( ( s >= 0 && s < 60 ) ? s : 0 );
    }

    // Convert to universal time
public String toUniversalString() {
    return String.format( "%02d:%02d:%02d",
                          hour, minute, second );
}

// Convert to standard time
public String toString() {
    return String.format( "%02d:%02d:%02d %s",
                           ( ( hour == 0 || hour == 12 ) ? 12 : hour%12 ),
                           minute, second, ( hour < 12 ? "AM" : "PM" ) );
}
public class Time1Test
{
    public static void main( String args[ ] )
    {
        Time1 time = new Time1();

        System.out.print( "The initial universal time is: " );
        System.out.print( time.toUniversalString( ) );
        System.out.print( "The initial standard time is: " );
        System.out.print( time.toString( ) );
        System.out.print( );

        time.setTime( 13, 27, 6 );
        System.out.print( "universal time after setTime is: " );
        System.out.print( time.toUniversalString( ) );
        System.out.print( "Standard time after setTime is: " );
        System.out.print( time.toString( ) );
        System.out.print( );

        time.setTime( 99, 99, 99 );
        System.out.print( "universal time after setTime is: " );
        System.out.print( time.toUniversalString( ) );
System.out.print( "Standard time after setTime is: " );
System.out.print( time.toString( ) );
System.out.print( );
}
Notes: The Time Class

• Every class defined in Java must extend some other object
  - Default: implicit extension of the "object" class

• Constructors do NOT have return types. Declaring one means the method is not a constructor
Design Issues

- Use the driver to test instances of the class

- Allocate and initialize instance variables in the constructor (and with private helper methods)

- Do not allow "users" of object access to internals of the object; use the interface only

- Group instance variables and various methods together for a logical organization
Class and Block Scope

```java
Class foo {
    int a;

    public int do ( int par ) {
        int temp = par + 20 ;
        a = par ;
    }
}
```

- **Instance variables**
  - global to the object
  - Use them as such

- **Block variables**
  - local to methods/blocks
• **Formal parameters**
  - define the *interface*
Class and Block Scope

- Public, private, protected

- Public:

  - method/instance variable is accessible by name "outside" the class

Time t;
t.hour = 2;  // not permitted since
            // is a private var
Class Access

• Protected

  - access is permitted for members of the same *inheritance hierarchy* and for object within the same *package*

• Protected access

  - an intermediate access level between fully public and private

• Making Testing/debugging easier

  - make all instance variables private and debug via public methods; this localizes testing/debugging to particular object
Packages

- allow portable objects to be "archived" and used by other code, similar to a "library"

- **Steps for making a package:**
  - make an object "public"
  - add "package" statement to object definition
  - compile object source code (compiler will store compiled code according to package statement)
  - import object into other code via "import" statement
Defining Constructors

- characteristics:
  - valid method definition
  - with the same name as the class
  - no return type or return values

- At least one public constructor

- Constructors (and all members) can be overloaded
Constructor Overloading

- Constructors with formal parameters (and variations in parameter list) allow for specific object instantiation

- Initializers:
  - actual parameter list supplied in the `new` command when objects are instantiated
  - must match a defined constructor
Constructors: Proper Use

- Instance variables should be allocated and/or initialized within the constructor.

- Multiple constructors (overloaded) can provide shorthand and natural instantiation calls for code that used the object.

- Default constructor is always a "no argument" (no initializers) constructor. No default constructor is provided if the programmed supplies at least one.
public class Time2 extends Object {
    private int hour; // 0-23
    private int minute; // 0-59
    private int second; // 0-59

    public Time2( ) { setTime( 0, 0, 0 ); } // this(0, 0, 0)

    public Time2( int h ) { setTime( h, 0, 0 ); } // this(h, 0, 0)

    public Time2( int h, int m ) { setTime( h, m, 0 ); }

    public Time2( int h, int m, int s ) { setTime( h, m, s ); }

    public Time2( Time2 time ) {
        setTime( time.hour, time.minute, time.second ); }

    public void setTime( int h, int m, int s ) {
        hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
        minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
second = ( ( s >= 0 && s < 60 ) ? s : 0 );

public int getHour() { return hour; }

public int getMinute() { return minute; }

public int getSecond() { return second; }

// Convert to universal time
public String toUniversalString() {
    return String.format( "%02d:%02d:%02d", 
                           getHour(), getMinute(), getSecond() );
}

// Convert to standard time
public String toString() {
    return String.format( "%02d:%02d:%02d %s", 
                           ( ( getHour() == 0 || getHour() == 12 ) ? 12: getHour()%12 ), 
                           getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
}
public class Time2Test {
    public static void main(String[] args) {
        Time2 t1 = new Time2();
        Time2 t2 = new Time2(2);
        Time2 t3 = new Time2(21, 34);
        Time2 t4 = new Time2(12, 25, 42);
        Time2 t5 = new Time2(27, 74, 99);
        Time2 t6 = new Time2(t4);

        System.out.println("Constructed with: ");
        System.out.println("t1, all arguments defaulted");
        System.out.printf(" %s\n", t1.toUniversalString());
        System.out.printf(" %s\n", t1.toString());

        System.out.println("t2: hour specified; minute and second defaulted");
        System.out.printf(" %s\n", t2.toUniversalString());
        System.out.printf(" %s\n", t2.toString());

        System.out.println("t3: hour and minute specified; second defaulted");
        System.out.printf(" %s\n", t3.toUniversalString());
        System.out.printf(" %s\n", t3.toString());
    }
}
System.out.printf(" %s
", t3.toUniversalString( ) );
System.out.printf(" %s
", t3.toString( ) );

System.out.println("t4: hour, minute and second specified" );
System.out.printf(" %s
", t4.toUniversalString( ) );
System.out.printf(" %s
", t4.toString( ) );

System.out.println("t5: all invalid values specified" );
System.out.printf(" %s
", t5.toUniversalString( ) );
System.out.printf(" %s
", t5.toString( ) );

System.out.println("t6: Time2 object t4 specified" );
System.out.printf(" %s
", t6.toUniversalString( ) );
System.out.printf(" %s
", t6.toString( ) );

} 
}
Accessors and Mutators

• **Accessor** - NO change in object fields
  - return private instance variables to outside caller

• **Mutator** - cause change in object fields
  - allow modification of private instance variables by outside caller

• Often as (public) **get** and **set** methods

• can and should enforce data **consistency** and **error checking**

• Is it true that providing **get** and **set** methods is essentially the same as making instance variables public?
Constants via "final"

- NO change in value
  - similar to the \texttt{const} keyword in C

- Variables declared with \texttt{final}
  - must be initialized in the declaration
  - can be initialized in a constructor, but must be assigned a value in \textit{every} constructor of the class
public class increment {
    ...
    private final int INCREMENT;
    
    public Increment( int inValue) {
        INCREMENT = inValut; }
    ...
}

public class IncrementTest {
    public static void main String args[] ) {
        Increment value = new Increment( 5 );
        
        System.out.println( 
            "Before incrementing: %s
\n\n", value );
        ...
    }
}
Building Composite Objects

- Objects can other objects as instance variables

- Building up complex classes is the goal for good design: reuse of existing classes

- To detect bugs incrementally
  - debugging with test drivers as classes are developed
public class Date extends Object {
    private int mn;    // 1-12
    private int dy;    // 1-31 based on month
    private int yr;    // any year

    public Date( int mn, int dy, int yr ) {
        this.mn = checkMonth( mn );
        this.dy = checkDay( dy );
        this.yr = yr;
    }

    private int checkMonth( int mn ) {
        if ( mn > 0 && mn <= 12 ) return mn;
        else {
            System.out.println(
                "Invalid month (%d) set to 1.", mn );
            return 1;
        }
    }

    public String toString( ) {
        return String.format(
            "%d/%d/%d", mn, dy, yr );
    }
}
public class Employee {
    private String firstName;
    private String lastName;
    private Date birthdate;
    private Date hiredate;

    public Employee( String fName, String lName, Date bDate, Date hDate ) {
        firstName = fName;
        lastName = lName;
        birthDate = bDate;
        hireDate = hDate; }

    public String toString( ) {
        return String.format("%s, %s Hired: %s Birthday: %s",
                           lastName, firstName, hireDate, birthDate); }
}
public class EmployeeTest {
    public static void main( String args[] ){
        Date birth = new Date( 7, 24, 1949 );
        Date hire = new Date( 3, 12, 1988 );
        Employee employee = new Employee(
                "Bod", "Blue", birth, hire );

        System.out.println( employee );
    }
}
References Using "this"

- Handle to yourself
- Reference is to method or instance vars
class SimpleTime {

    private in hour, minute, second;

    public SimpleTime(int hour, int minute, int second) {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    ...
}
Example: Time4

- Can "chain" method calls by returning "this" reference:

```java
// set the hour

public Time4 setHour( int h )
{
    this.hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
    return this;
}
...
Time4 t = new Time4();
String output;

t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
```
Finalizers

• Called at the last moment when an object will not exist

• For housekeeping: garbage collection
  - automatic management of resources by the Java system

• One finalizer definition per class
  - always use the name `finalize`
  - no overloading
  - no parameters, return `void`

• Can explicitly call or can let garbage collector invoke

• Often special "deallocation" can cleanup is done in programmer-defined finalizer
Static Class Members

- Normal instance variables
  - separate copies in each instance of a class

- Static instance variables
  - shared by all instances of a class, i.e., there is only one copy of a static instance variable.
public class Employee extends Object {
    private String firstName;
    private String lastName;
    private static int count;

    public Employee( String fName, String lName )
    {
        firstName = fName;
        lastName = lName;

        count++;
        System.out.println("Employee constructor: %s %s; count = %d\n", firstName, lastName, count);
    }

    ...
}
Notes on Object-Oriented Design

• Design code around class/object/method construct

• Plan for re-use of key data objects via "packages"

• Restrict object access via public/private

• Design a good interface with accessors, mutators and hidden methods/data

• Separate internal implementation of object from it interface