Chapter 11:
Create Your Own Objects

Think Java:
How to Think Like a Computer Scientist

5.1.2

by Allen B. Downey

Our usual text takes a fairly non-standard departure in this chapter. Instead, please refer to Chapter 5 Sections 5.1 – 5.4 of Eck, "Java Notes", also linked from the class website. This is a longer treatment than what we need, but it's great for those who want to learn more on this topic.

Agenda

- Structured Programming Paradigm
- Object-Oriented Programming Paradigm
- Creating your own Object Classes
- Time Class
  - Constructors, Access Modifiers, get/set, printing
- Date Class, Student Class for reinforcement
- Object methods vs Class methods
- toString, equals, and compareTo methods
- Generating Javadocs

Programming languages and styles

- Many programming ***languages*** and ***styles***

popular

Bash
C
C#
C++ 4.8.1
C++11
Haskell
Java
Java7
Objective-C
Pascal (fpc)
Pascal (gpc)
Perl
PHP
Python
Python 3
Ruby
SQL
VB.NET

others

Ada
Assembler
Assembler
AWK (gawk)
AWK (mawk)
bc
Brainf**k
C++ 4.3.2
C99 strict
CLIPS
Clojure
COBOL
COBOL 85
Common Lisp
D (dmd)
Erlang
F#
Factor
Falcon
Forth
Fortran
Go
Groovy
Icon
Intercal
JavaScript (rhir
JavaScript (spi
Lua
Nemerle
Nice
Nimrod
Node.js
Ocaml
Octave
Oz
PARI/GP
Perl 6
Pike
Prolog (gnu)
Prolog (swi)
R
Scala
Scheme (guile
Smalltalk
Tcl
Text
Unlambda
Whitespace

Structured Programming (1960's – 1980's)

often called "Procedural" Programming

- Problems are broken down into "chunks"
  - each chunk is a method
  - write each method and test
  - string together calls to different methods → solution
  - focus on the "steps" of a problem
  - (hopefully) re-use methods in other solutions
- As software became more complex,
  - developers found it necessary to group related methods into "Library" files.
  - import Libraries when needed

Structured Programming uses static methods

- There were no classes back then.
  - However, in Java, the closest thing to a "Library" would be a class that contains only static methods.
  - Like the Math class in Java

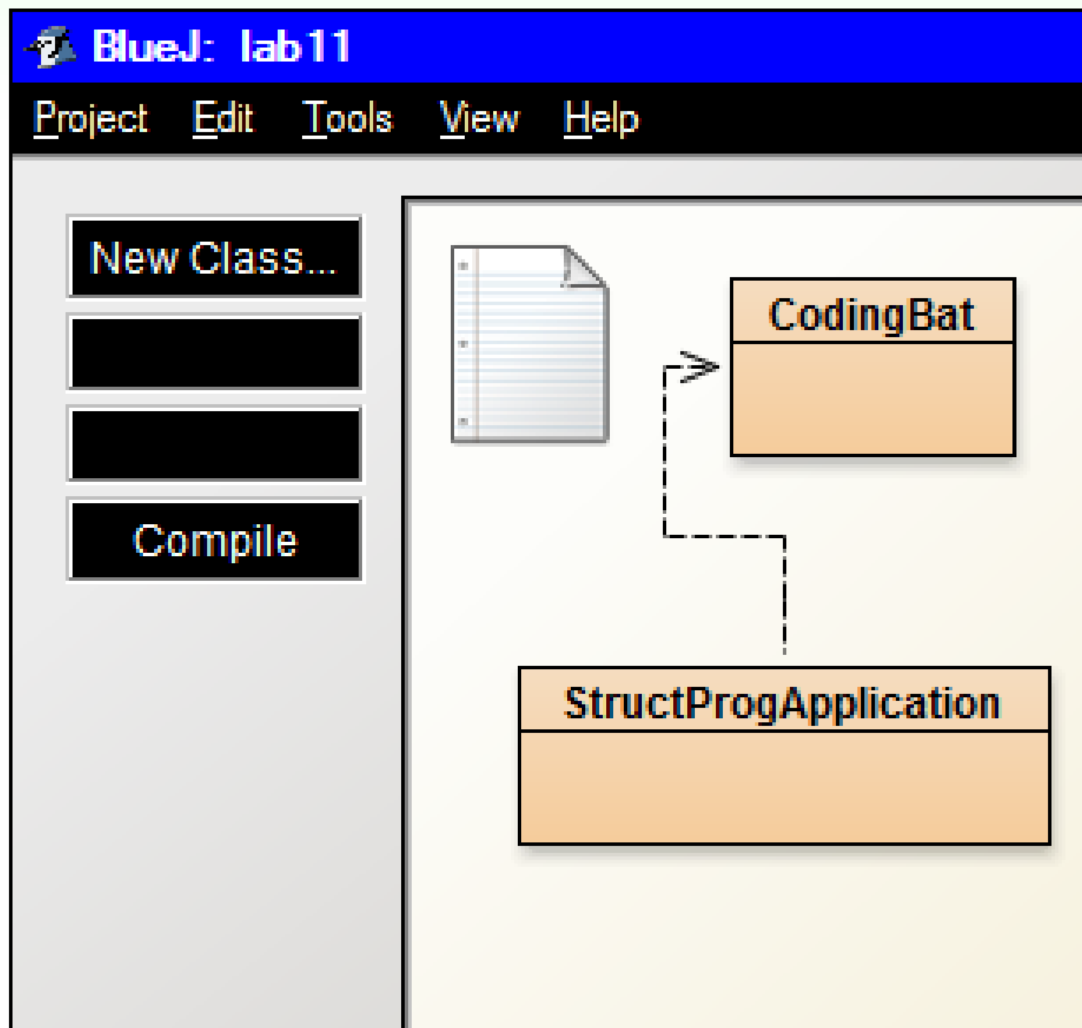
- To use an external "Library" (static) method in Java
  - Name of class, (period), name of method
    - double y = Math.sqrt(248);
    - double z = Math.round(y);

CodingBat Exercises = Structured Programming

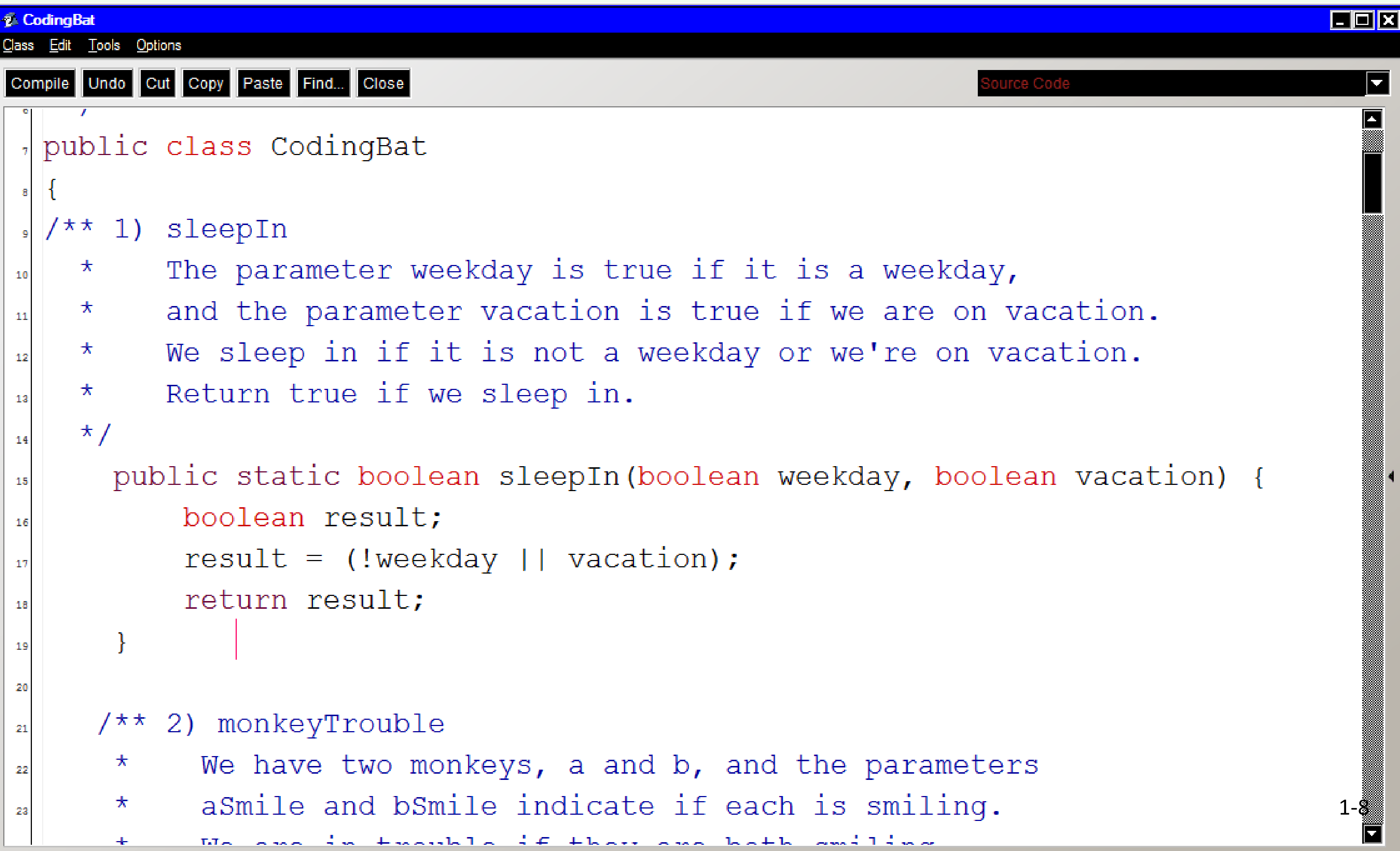
- You wrote and tested a number of methods on CodingBat
  - never got to use them in a real application
  - put all of your static methods in CodingBat class
    - you now have a "library"
    - you can use the methods in other programs.

  - We will demo this now in the Lab11 download
    - Open StructuredProgramming class and complete exercises

Here is the CodingBat "Library" / class with a Structured Programming Application in BlueJ



Here is the CodingBat "Library" with a static method to tell if we can sleepIn



Invoking a static method in an external class
notice: if (***CodingBat***.sleepIn ...)

StructProgApplication

Class Edit Tools Options

Compile Undo Cut Copy Paste Find... Close Source Code

```
11 public class StructProgApplication
12 {
13     public static void main(String [] args){
14         Scanner keyboard = new Scanner(System.in);
15         System.out.println("\f");
16         //************* Demo 1 **********************
17         // We will use the sleepIn method in the CodingBat class
18         //   to tell us if we can sleep in tomorrow
19         System.out.println("Is tomorrow a weekday? Please enter true or false");
20         boolean weekday = keyboard.nextBoolean();
21         System.out.println("Is tomorrow a holiday? Please enter true or false");
22         boolean holiday = keyboard.nextBoolean();
23
24         // notice, to invoke sleepIn, we have to precede it with the
25         if (CodingBat.sleepIn(weekday, holiday))
26             System.out.println("You can sleep in tomorrow!");
27         else
            System.out.println("Better set your alarm!");
```

changed

But…remember Old CodingBat class?

- If the static methods are in the same class as main, you don't need to put the class name in front of them.

```
public class CodingBat
{
    public static void main(String [] args)
    {
        /************** Prob 1 ***********************
        // Write the method sleepIn below main.
        //  test by running this segment
        System.out.println("sleepIn(false, false) =" +sleepIn(false,false) + "
        System.out.println("sleepIn(true, false) =" +sleepIn(true,false) + " sh
        System.out.println("sleepIn(false, true) =" +sleepIn(false,true) + " sh
        System.out.println("sleepIn(true, true) =" +sleepIn(true,true) + " shou
```

Decline of Structured Programming as dominant Paradigm (late '80's)

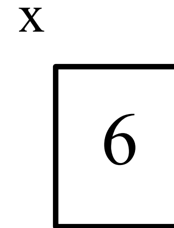
- Eventually, software projects grew even bigger than what the Structured Programming paradigm could deal with
  - Apple introduces Graphical User Interfaces (Mac)
  - Microsoft introduces WIndows
- Humans could not make sense of code involving thousands of methods
  - Coupling -- change one method = change lots more
  - Software maintenance grew increasingly hard
  - Software not so easy to re-use, wasted $$$

Object-Oriented Programming (1990's - ?)

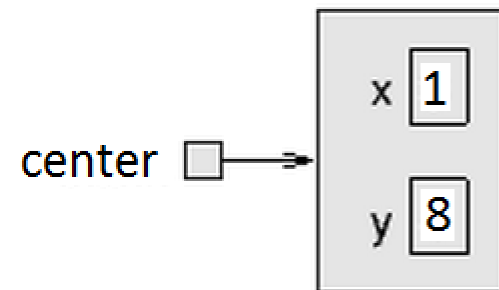
- In time, a new programming paradigm replaced Structured Programming in prominence
  - Software now focused on the "things" in the problem
  - Solutions involve Objects interacting with other Objects
  - Objects are like "smart" variables that contain
    - data (multiple related instance variables)
    - methods that maintain the object's data, interact with others
  - Less coupling ➔ easier to design
    - Code is easier to maintain and re-use
    - saves $$$

Review:
Two Ways of representing information in Java

- Primitive types
  - int x = 6;



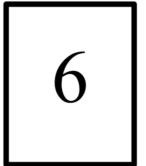
- Object types
  - Point center= new Point(1,8);



Primitive Types

- `int, char, double, boolean`
- hold only one piece of information

```
int x = 6; char c = '@';
```



- allow only certain operations
  - arithmetic, comparison

```
x=x+4;
if(c == '#'){
```

Object types

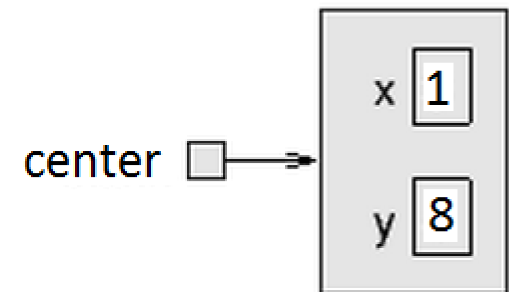
- `Scanner, String, Point, Rectangle`
- composed of multiple pieces of information
- Object methods allow many different operations

```
word.toUpperCase();

center.translate(50,0);

keyboard.nextInt();

shape.addPoint(100,20);
```



You can define your own object types

- If you need an object type Java hasn't defined
  - You can make your own
  - This week we will be defining classes to represent
    - Time objects 11:40:33
    - Date objects 10/31/14
    - Student objects Name: Jasmine Rizzo, units: 50, GPA: 3.9
  - Defining a new **class** also creates a new object **type** with the same name.
  - A class definition is like a template for objects
    - determines what information it holds (instance variables)
    - determines what methods can be performed

Objects and Classes

- Every ***object*** belongs to some object type; that is, *it is an instance of some* ***class***.
- When you invoke **new** to create an object, Java invokes a special method called a **constructor** to initialize the instance variables. You provide one or more constructors as part of the class definition.
- The methods that operate on an object type are defined in the class definition for that type.

Example of a user-defined class

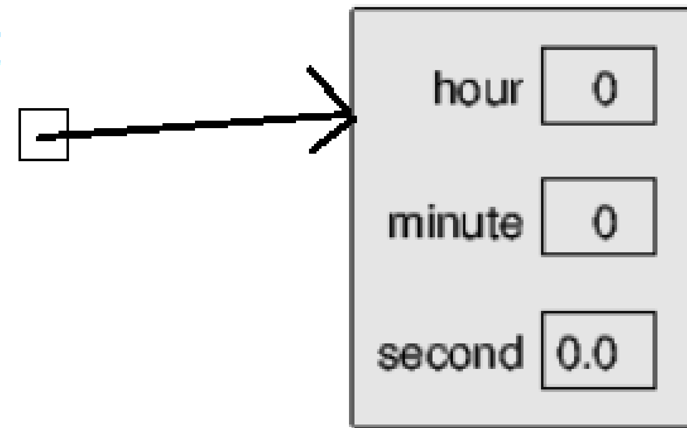
- Declare a Time class:

```
class Time {
     int hour, minute;
     double second;
}
```

instance variables

- Create a Time object variable

```
Time  t = new Time();
```



- What it looks like in memory

- Now run the main method in lab11/TimeTestApp.java

Constructor Method (Default or No-arg)

- Special method to initialize instance variables
- Same name as class in which it's defined

The keyword `static` is removed

No return type

initialize instance variables

```
public Time() {
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
}
```

this means the object we are creating, "this object"

Add the no-arg (aka "default") constructor to the Time class

Explicit Constructor

- Often need a constructor with a parameter list
  - identical names to instance variables
- Just copies the information from the parameters to the instance variables.

```
public Time(int hour, int minute, double second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
}
```

- This is an example of "overloading"
  - now have two methods named Time in Time class

Overloading the Constructor

- Java tells which you mean by your method call:

```
Time t = new Time();
```

→

```
public Time() {
      this.hour = 0;
      this.minute = 0;
      this.second = 0.0;
}
```

```
Time appt = new Time(11,30,0);
```

```
public Time(int hour, int minute, double second) {
      this.hour = hour;
      this.minute = minute;
      this.second = second;
}
```

Add the above constructor to the Time class

Compare: Point Class Constructors

Constructor Summary

`Point()`
Constructs and initializes a point at the origin (0, 0) of the coordinate space.

`Point(int x, int y)`
Constructs and initializes a point at the specified `(x,y)` location in the coordinate space.

`Point(Point p)`
Constructs and initializes a point with the same location as the specified `Point` object.

Why Multiple Constructors?

- Provides flexibility, you can either:
  A. create an object first then fill in the blanks (no-arg)
  B. collect all the info before creating the object.
- Not terribly interesting
  - Writing constructors is a boring, mechanical process.
  - Can write quickly
    - just by looking at list of instance variables.

Our Time Class so far

```
class Time {
    int hour, minute;
    double second;

    public Time() {
        this.hour = 0;
        this.minute = 0;
        this.second = 0.0;
    }

    public Time(int hour, int minute, double second) {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }
```

Our TimeTestApp class main method

```
{
    public static void main(String [] args){
        // Create an object of class Time and store a reference to it
        //  in variable t
        Time t = new Time();
        t.hour = 11;
        t.minute = 8;
        t.second = 3.5;
        System.out.println("t = " + t.hour
                                  + ":" + t.minute
                                  + ":" + t.second);
```

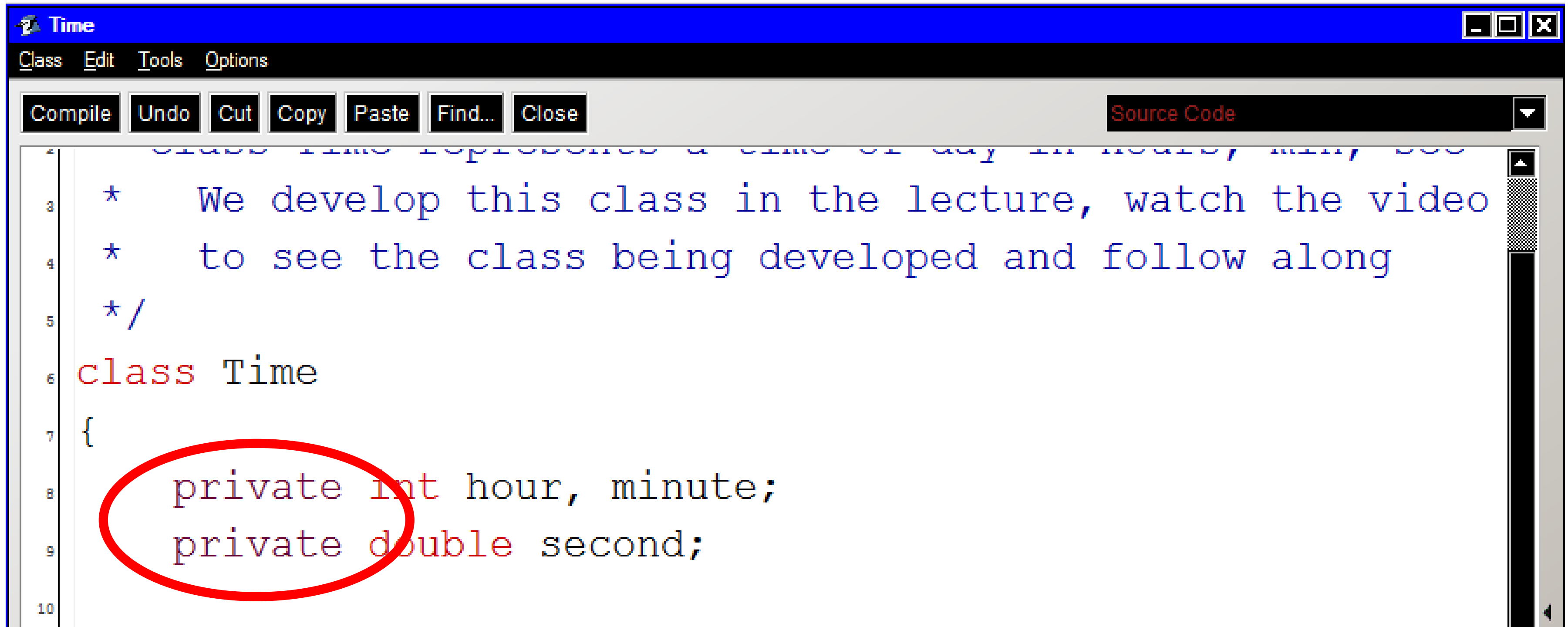
The output of this program is:

```
t = 11:8:3.5
```

Allowing user to set the fields is dangerous!

Access Modifiers Protect data

- An example of Encapsulation
  - "private" means only methods of Time class can change



We now need "Getters and Setters"

to access and modify the object data

- The getHour method
  - returns the hour field of a Time object

```
public int getHour(){
    return hour;
}
```

- The setHour method
  - lets you update the hour field
  - could provide additional safety checking

```
public void setHour(int hour){
    this.hour = hour;
}
```

  - very easy to code..."mindless"

Using the get and set methods

- These methods are invoked using the syntax
  - *object name* (*dot*) *method name* (*parameters*)

```
Time appt = new Time(10,45,00);
System.out.println("Your appointment is at: " + appt.getHour() + ":"
                   + appt.getMinute() + ":" + appt.getSecond());

// you can also set the different fields of a time object directly (for no
appt.setHour(11);
System.out.println("Your new appointment is at: " + appt.getHour() + ":"
                   + appt.getMinute() + ":" + appt.getSecond());
```

Default Printing for User-defined Objects

- By default, Java prints the
  - name of the type (Time)
  - special hexadecimal (base 16) code
- Special code is
  - unique for each object
  - not meaningful in itself
  - can vary from machine to machine, and run to run
  - useful for debugging, in case you want to keep track of individual objects.

```
System.out.println(appt)
```

The output of this program is:

```
Time@80cc7c0
```

Define a print method

```
public void print () {
  System.out.println( hour + ":" +
                      minute + ":" +
                      second);

}
```

```
appt.print();
```

The output of this program is:

```
11:45:0
```

Better, but

Still not perfect

how to get 11:45:00 ?

One way: Revised print method

```
public void print () {
  System.out.print(hour + ":");
  if (minute < 10)
    System.out.print("0" + minute + ":");
  else
    System.out.print(minute + ":");
  if (second < 10)
    System.out.println("0" + second);
  else
    System.out.println(second);
}
```

```
appt.print();
```

The output of this program is:

```
11:45:00
```

Do Lab11 PartA

- Complete the definition for a Date class and test
  - use the Time class to model the pattern
  - follow the instructions in DateTestApp

- Use a Student class
  - add/test more capabilities

Agenda

- Review Object methods vs Class methods
- Using objects of a class in more situations
  - creating objects from info from read from keyboard
  - comparing parts of two objects using get methods
  - writing a static method to process an object

Programming languages and styles

- Programming styles often called ***paradigms***
- Programs we wrote so far are ***procedural*** style
  - emphasis on computational procedures
- Dominant paradigm in modern software is **object oriented** programming
  - emphasis shifts to objects and their behaviors
  - melding of data and methods into one thing: object
- Modern software is often a mix of the two
  - being skilled in both paradigms key to success

Review: Characteristics of OOP

- Objects represent entities in the real world
  - An employee at a company
  - A zombie in a video game
  - A Bug/Rock/Flower in GridWorld
- Majority of methods are ***object methods***
  - like String methods: String s = "hi"; s.length();
- Rather than ***class methods***
  - like Math methods: Math.sqrt(5); Math.random().
- The methods we have written before this week have all been class methods.

Review: Two Principles of OOP

- **Encapsulation**: objects are isolated from each other by limiting the ways they interact, especially by preventing them from accessing instance variables without invoking methods.
- **Inheritance:** Classes are organized in family trees where new classes extend existing classes, adding new methods and replacing others.
  - OOP design principles lead to cost savings over procedural only design

Review: Object methods vs class methods

- Class methods: have keyword **<u>`static`</u>** in header
  - invoked using the Class containing the method
    - Math.sqrt(3) , CodingBat.sleepIn(false, true);
- Object methods: invoked ***on*** an object
  - String s = "hi"; s.length(); s.charAt(1); t.print();
  - any method ***without*** **`static`** is an object method
- Can convert object methods to class methods
  - and vice versa
  - sometimes more natural to use one or the other

Review: print, an object method in Time class

```
class Time{
   int hour, minute; double second;
      ...
      public void print() {
       System.out.println( hour + ":" +
                           minute + ":" +
                           second);
     }
}
```

Inside an object method you can refer to instance variables as if they were local variables

Same print method, adding `this` keyword

- Using **this** to refer to current object:

```
public void print() {
   System.out.println( this.hour + ":" +
                       this.minute + ":" +
                       this.second);
}
```

By using **this**, it makes clear that **hour**, **minute** and **second** are instance variables. Talking about *this* object's hour, minute, second

Invoking the object method `print`

- Here's how it is invoked:

```
Time now = new Time(11,35,40);
now.print();
```

- When you invoke a method on an object
  - it becomes the **current object**
  - also known as `this`
  - Inside `print`, the keyword `this` would refer to the Time object the method was invoked on

printTime, a class method in Time class

- This is a **class** method in the Time class:

```
public static void printTime(Time t) {
  System.out.println(   t.hour + ":" +
                        t.minute + ":" +
                        t.second);
}
```

Notice—1) keyword **static** means class method

2) Time `t` is received as a **parameter**

Invoking the class method `printTime`

- Here's how it is invoked:

```
Time now = new Time(11,35,40);
Time.printTime(now);
```

- When you invoke a class method on an object
  - the object is passed as an argument
  - invoke with the Class name, "dot", method name
- Add printTime to Time class
  - verify in TimeAppTester class

Organizing Class Definitions

- You can define object methods and class methods in same class.
- Common order to keep clear:
  - define instance variables
  - define object methods
  - define class methods

Encapsulation

- To preserve sanity in large projects
  - limit access that other pieces of code have to our particular object.
  - Stop "silly" (or malicious) programmers from changing the values of our instance variables.
  - For example, in another class file someone writes:

```
Time t = new Time(12,25,35);
t.hour = -15;
```

← forbidden when field marked "private"

```
System.out.println("Hour is " + t.hour);
```

Accessing private fields of an object

- Accessor method for Time class, hour field:
  - return the value of an instance variable

```
public int getHour() {
  return hour;
}
```

- Invoking the method:

```
Time t = new Time(10,40,30);
System.out.println("Hour is " + t.getHour());
```

Modifying private fields of an object

- Modifier method for Time class, hour field:
  - change the value of an instance variable

```
public void setHour(int newHour) {
  if (newHour>=0 && newHour < 24)
          hour = newHour;
}
```

- Invoking the method:

```
Time t = new Time(10,40,30);
t.setHour(4);     t.setHour(-15)
```

(ignored)

Summary: our object method toolbox

- Constructors (default and explicit) – to create
- Set and Get methods – to modify
- print method – to see what object holds

- These are a minimum toolbox with which we can do a number of interesting things with objects.

Reading an object from keyboard

- First, ask user to enter data

```
…println("When is appt? Enter hr, min, sec");
                    // user types 9 30 00
```

- Read data into temporary local variables

```
int hr = keyboard.nextInt();
int min = keyboard.nextInt();
double sec = keyboard.nextDouble();
```

- Create object using explicit constructor

```
Time appt = new Time(hr, min, sec);
…print("your appt is at");
appt.print();              // prints 9:30:0
```

We can compare objects

- You can compare different fields of an object

```
if ( appt.getHour() > now.getHour() )
   System.out.println("Your appt has passed!");
```

- Don't use the equals method yet, we'll write later

```
if (appt.equals(now))
   System.out.println("Your appt is now!");
```

We can write additional object methods

```
class Time{
   private int hour, minute;
   private double second;
       ...
    public double totalSeconds() {
        double total =  hour * 3600 +
                        minute * 60 +
                        second;
        return total;
      }
}
```

EXTRA MATERIAL

- The following slides illustrate three other very useful methods you may want to incorporate into your classes to fully round out their capability.
  - toString
  - equals
  - compareTo

- You don't need to learn these for this week. They are optional for now.

The **`toString`** method

- Every object type has a method **`toString`**
  - returns a string representation of the object
- When you **`System.out.print`** an object
  - Java calls the object's toString method automatically
  - Default version just returns the object's Hex address
  - **`System.out.print("The time is " + now);`**
  - prints:

```
The time is Time@80cc7c0
```

Define a **`toString`** method for Time

- Can **override** the default behavior with own def:

```
public String toString() {
        return    hour + ":" +
                  minute + ":" +
                  second;
}
```

- Allows for better output:
  - **`System.out.print("The time is " + now);`**
  - prints: **`The time is 11:35:40`**

You could also invoke **`toString`** explicitly

- Just like any other object method

```
Time now = new Time(12,23,47);
String s = now.toString();
```

The `equals` method

- Two notions of equality:
  - identity (==)
    - two object variables that refer to the same object

```
String s1 = "yes", s2 = "yes";
if (s1 == s2) …
```

false

  - equivalence (`equals` method)
    - two objects that have the same values.

```
if (s1.equals(s2)) …
```

true

- Java provides default equals method
  - same as identity (==)

Defining `equals` method for Time class

```
class Time {
   private int hour, minute;
   private double second;
  ...
   public boolean equals(Time that) {
      return this.hour == that.hour &&
             this.minute == that.minute &&
             this.second == that.second;
   }
}
```

the compareTo method

- Checking for less than/greater than requires writing a compareTo method.
- Remember, compareTo returns
  - negative value if first object < second object
  - 0 if first object .equals second object (same data)
  - positive value if first object > second object
- Here's how you would use it:

```
if ( appt.compareTo(now) > 0 )
   System.out.println("You missed your appt!");
```

Defining `compareTo` method for Time class

```
class Time {
   private int hour, minute;
   private double second;
 ...
   public int compareTo(Time that) {
      if (this.hour < that.hour)
         return -1;
      else if (this.hour == that.hour &&
               this.minute < that.minute
         return -1;
      else if (this.hour == that.hour &&
               this.minute == that.minute &&
               this.second < that.second)
         return -1
         … continue to cover all possible cases
   }
}
```

Alternative `compareTo` method for Time class

```
class Time {
   private int hour, minute;
   private double second;
 ...
   public int compareTo(Time that) {
     double thisTotalSec =   this.totalSeconds();
     double thatTotalSec =   that.totalSeconds();
     if (thisTotalSec < thatTotalSec)
          return -1;
     else if (thisTotalSec > thatTotalSec)
          return 1;
     else
          return 0;
   }
}
```

(the totalSeconds method was defined on slide 51)