# Chapter 9
# Classes and Objects: A Deeper Look (I)

TAO Yida

taoyd@sustech.edu.cn

# Outline

- A deeper look at designing classes, controlling access to class members and creating constructors
- Reexamine the use of *set* and *get* methods (setter & getter)
- Composition (组合)—a capability that allows a class to have references to objects of other classes as members (has-a relationship)

# Recall Our Car Example

- Class – a car's engineering drawings (a blueprint)
- Method – designed to perform tasks (e.g., making a car move)
- Object – the car we drive
- Method call – perform the task (e.g., pressing the accelerator pedal)
- Instance variable – to specify the attributes (e.g., the amount of gas)

# A Time Class

```
public class Time1 {
    private int hour; // 0 - 23
    private int minute; // 0 - 59
    private int second; // 0 - 59

    // set a new time value using universal time
    public void setTime(int h, int m, int s) {  // ...
    }

    // convert to String in universal-time format (HH:MM:SS)
    public String toUniversalString() {  // ...
    }

    // convert to String in standard-time format (H:MM:SS AM or PM)
    public String toString() {  // ...
    }
}
```

private instance variables

public instance methods (public services / interfaces the class provides to its clients)

# Method Details

```
public class Time1 {
    // set a new time value using universal time
    public void setTime(int h, int m, int s) {
       hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // validate hour
       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
    }
}
```

# Method Details Cont.

```
public class Time1 {
    // convert to String in universal-time format (HH:MM:SS)
    public String toUniversalString() {
        return String.format("%02d:%02d:%02d", hour, minute, second);
    }
    // convert to String in standard-time format (H:MM:SS AM or PM)
    public String toString() {
        return String.format("%d:%02d:%02d %s",
                (hour == 0 || hour == 12) ? 12 : hour % 12,
                minute, second, hour < 12 ? "AM" : "PM");
    }
}
```

# Default Constructor

- Class `Time1` does not declare a constructor
- It will have a default constructor supplied by the compiler
- `int` instance variables implicitly receive the default value `0`
- Instance variables also can be initialized when they are declared in the class body, using the same initialization syntax as with a local variable

```
public class Time1 {
    private int hour = 10; //default constructor will not initialize hour
    private int minute; //default constructor will initialize minute to 0
    private int second; //default constructor will initialize second to 0
}
```

# Using The `Time` Class

```
public class Time1Test {
    public static void main(String[] args) {
        Time1 time = new Time1(); // invoke default constructor
        System.out.print("The initial universal time is: ");
        System.out.println(time.toUniversalString());
        System.out.print("The initial standard time is: ");
        System.out.println(time.toString());
    }
}
```

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM
```

# Manipulating The Object

```
public class Time1Test {
    public static void main(String[] args) {
        Time1 time = new Time1();

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

    }
}
```

Use object reference to invoke an instance method

```
Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM
```

# Manipulating The Object

```
public class Time1Test {
    public static void main(String[] args) {
        Time1 time = new Time1();

        time.setTime(99, 99, 99);
        System.out.println("After attempting invalid settings: ");
        System.out.print("Universal time: ");
        System.out.println(time.toUniversalString());
        System.out.print("Standard time: ");
        System.out.println(time.toString());
    }
}
```

```
After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

# Manipulating The Object (Analysis)

- A `Time1` object always contains *valid data*
  - The object's data values are always kept in range, even after incorrect values are passed to `setTime`.
- In our example, zero is a *valid* value for `hour`, `minute` and `second`.
- `hour`, `minute` and `second` are all set to zero by default; thus, a `Time1` object contains valid data from the moment it is created.

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

# Valid Value vs. Correct Value

- A valid value for `minute` must be in the range 0 to 59.
- A correct value for `minute` in a particular application would be the actual minute at that time of the day.
  - If the actual time is 17 minutes after the hour and you accidently set the time to 19 minutes after, the 19 is a *valid* value (0 to 59) but not a *correct value*.
  - If you set the time to 17 minutes after the hour, then 17 is a correct value—and a correct value is *always* a valid value.

# Handling Invalid Values

- Our current `setMethod` sets the corresponding instance variables to zeros when receiving invalid values.

```
// set a new time value using universal time
public void setTime(int h, int m, int s) {
    hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // validate hour
    minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
    second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
}
```

While 0 is certainly a valid value, it is unlikely to be correct. Is there an alternative approach?

# Handling Invalid Values

- When receiving invalid values, we could also simply leave the object in its current state, without changing the instance variable.
  - `Time` objects begin in a valid state and `setTime` method rejects any invalid values.
  - Some designers feel this is better than setting instance variables to zeros.

```
// set a new time value using universal time
public void setTime(int h, int m, int s) {
    if(h >= 0 && h < 24) hour = h; // reject invalid values
    if(m >= 0 && m < 60) minute = m;
    if(s >= 0 && s < 60) second = s;
}
```

# Notifying The Client Code

- Approaches discussed so far do not inform the client code of invalid values (no return to callers)

```
// approach 1: setting to zeros
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 );
}
```

```
// approach 2: keeping the last object state
public void setTime(int h, int m, int s) {
    if(h >= 0 && h < 24) hour = h;
    if(m >= 0 && m < 60) minute = m;
    if(s >= 0 && s < 60) second = s;
}
```

# Notifying The Client Code

- `setTime` could return a value such as `true` if all the values are valid and `false` if any of the values are invalid.
  - The caller would check the return value, and if it is `false`, would attempt to set the time again.
  - Problem: Some Java technologies (such as JavaBeans) require that the *set* methods return `void`.

```
public boolean setTime(int h, int m, int s) {...}
```

- Exception Handling is another technique that enables methods to indicate when invalid values are received.

# Data Hiding (Information Hiding)

- The instance variables `hour`, `minute` and `second` are each declared `private`.
- **Principle:** *The actual data representation used within the class is of no concern to the class's clients.*

```
public class Time1 {
    private int hour; // 0 - 23
    private int minute; // 0 - 59
    private int second; // 0 - 59
}
```

# Data Hiding (Information Hiding)

- It is reasonable for `Time1` to represent the time internally as the number of seconds since midnight or the number of minutes and seconds since midnight.
- Clients could use the same `public` methods and get the same results without being aware of this implementation detail.

```
// set a new time value using universal time
public void setTime(int h, int m, int s) {...}

// convert to String in universal-time format (HH:MM:SS)
public String toUniversalString() {...}

// convert to String in standard-time format (H:MM:SS AM or PM)
public String toString() {...}
```

# Data Hiding (Information Hiding)

- No data hiding (all instance variables are public): The implementation of the class is inflexible and unsafe
  - Inflexible
    - e.g., we cannot easily add support for different time zone in the future
    - e.g., we cannot easily change the implementation of instance variables (float to int)
  - Unsafe: e.g., the time can be changed to be negative
- With data hiding: data is hidden behind a well-defined interface of methods
  - We have control over how fields are modified
  - We can also change the underlying implementation without breaking client code

# Controlling Access to Members

- Access modifiers `public` and `private` control access to a class's variables and methods.
  - Later, we will introduce another access modifier `protected`
- `public` methods present to the class's clients a view of the services the class provides (the class's `public` interface).
  - Clients need not be concerned with how the class accomplishes its tasks (i.e., its implementation details).
- `private` class members are not accessible outside the class.

# Accessing Private Members

```
public static void main(String[] args) {
    Time1 time = new Time1();
    time.hour = 7; // compilation error
    time.minute = 15; // compilation error
    time.second = 30; // compilation error
}
```

If this is allowed, objects can easily enter invalid states (clients can give `hour` arbitrary values).

# `this` Reference

- The keyword `this` is a reference variable that refers to the current object in Java.

Why non-static method?

- When a non-`static` method is called on a particular object, the method's body implicitly uses keyword `this` to refer to the object's instance variables and other methods.

```
// set a new time value using universal time
public void setTime(int h, int m, int s) {
    if(h >= 0 && h < 24) hour = h; // compiler's view: this.hour
    if(m >= 0 && m < 60) minute = m; // compiler's view: this.minute
    if(s >= 0 && s < 60) second = s; // compiler's view: this.second
}
```

```
Time timeObj = new Time();
timeObj.setTime(1,2,3);
// this is replaced by timeObj
```

# `this` Reference

- The main use of `this` is to differentiate the formal parameters of methods and the data members of classes.
- If a method contains a local variable (including parameters) with the same name as an instance variable, the local variable ***shadows*** the instance variable in the method's scope.

```
// set a new time value using universal time
public void setTime(int hour, int minute, int second) {
    // if we use hour here, it refer to the local variable
    // not the instance variable
}
```

Shadowing: using variables in overlapping scopes with the same name where the variable in low-level scope overrides the variable of high-level scope.

# this Reference

```
public class Time1 {
    private int hour; // 0 - 23
    private int minute; // 0 - 59
    private int second; // 0 - 59

    // set a new time value using universal time
    public void setTime(int hour, int minute, int second) {
        if(hour >= 0 && hour < 24) this.hour = hour;
        if(minute >= 0 && minute < 60) this.minute = minute;
        if(second >= 0 && second < 60) this.second = second;
    }
}
```

this enables us to explicitly access instance variables shadowed by local variables of the same name.

# this Reference

```
public class Time1 {
    // convert to String in universal-time format (HH:MM:SS)
    public String toUniversalString() {
        return String.format("%02d:%02d:%02d", hour, minute, second);
    }
    public String buildString() {
        return "Universal format: " + this.toUniversalString();
    }
}
```

**Q:** Do we need `this` reference here?

**A:** `this` is not required to call other methods of the same class.

# Overloaded Constructors

- **Method overloading (重载):** methods of the same name can be declared in the same class, as long as they have different sets of parameters
  - Used to create methods that perform same tasks on different types or different numbers of arguments (e.g., println(), methods in String class)
- Similarly, **overloaded constructors** enable objects of a class to be initialized in different ways (constructors are special methods).
- Compiler differentiates overloaded methods/constructors by their *signature* (method name, the type, number, and order of parameters).
  - `max(double, double)` and `max(int, int)`

# Overloaded Constructors (Example)

```
public class Time2 {
  private int hour;
  private int minute;
  private int second;

  public void setTime(int h, int m, int s) {
    setHour(h);
    setMinute(m);
    setSecond(s);
  }

  public void setHour(int h) {
    hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
  }

  public void setMinute(int m) {
    minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
  }

  public void setSecond(int s) {
   second = ( ( s >= 0 && s < 60 ) ? s : 0 );
  }
```

Set methods for manipulating the fields

# Overloaded Constructors (Example)

```
  public int getHour() {
    return hour;
  }
  public int getMinute() {
    return minute;
  }
  public int getSecond() {
    return second;
  }
  public String toUniversalString() {
    return String.format("%02d:%02d:%02d",
      getHour(), getMinute(), getSecond());
  }
  public String toString() {
    return String.format("%d:%02d:%02d %s",
      ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
      getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM") );
  }
}
```

Get methods for retrieving the value of the fields

# Overloaded Constructors (Example)

```
public class Time2 {
  public Time2(int h, int m, int s) {
    setTime(h, m, s);
  }
  public Time2(int h, int m) {
    this(h, m, 0);
  }
  public Time2(int h) {
    this(h, 0, 0);
  }
  public Time2() {
    this(0, 0, 0);
  }
  public Time2(Time2 time) {
    this(time.getHour(), time.getMinute(), time.getSecond());
  }
}
```

Invoke setTime to validate data for object construction

Invoke three-argument constructor, hour and minute values supplied

Using `this` in method-call syntax invokes another constructor of the same class. This helps reuse initialization code.

# Overloaded Constructors (Example)

```
public class Time2 {
  public Time2(int h, int m, int s) {
    setTime(h, m, s);
  }
  public Time2(int h, int m) {
    this(h, m, 0);
  }
  public Time2(int h) {
    this(h, 0, 0);
  }
  public Time2() {
    this(0, 0, 0);
  }
  public Time2(Time2 time) {
    this(time.getHour(), time.getMinute(), time.getSecond());
  }
}
```

Invoke three-argument constructor, hour value supplied

No-argument constructor, invokes three-argument constructor to initialize all values to 0

Another object supplied, invoke three-argument constructor for initialization

# Overloaded Constructors (Example)

```
public class Time2 {
  public Time2(int h, int m, int s) {
    setTime(h, m, s);
  }
  public Time2(int h, int m) {
    this(h, m, 0);
  }
  public Time2(int h) {
    this(h, 0, 0);
  }
  public Time2() {
    this(0, 0, 0);
  }
  public Time2(Time2 time) {
    this(time.getHour(), time.getMinute(), time.getSecond());
  }
}
```

Invoke three-argument constructor, hour value supplied

No-argument constructor, invokes three-argument constructor to initialize all values to 0

We use "`this`" to invoke a constructor here.
Cannot use Time2(…), which can only be used with the "`new`" operator

# Using Overloaded Constructors

```
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(t1.toUniversalString());
        System.out.println(t2.toUniversalString());
        System.out.println(t3.toUniversalString());
        System.out.println(t4.toUniversalString());
        System.out.println(t5.toUniversalString());
        System.out.println(t6.toUniversalString());
    }
}
```

Compiler determines which constructor to call based on the number and types of the arguments

```
00:00:00
02:00:00
21:34:00
12:25:42
00:00:00
12:25:42
```

# More on Constructors

- Every class must have at least one constructor.
- If you do not provide any constructors in a class's declaration, the compiler creates a default constructor that takes no arguments when it's invoked.
- The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values (zero for primitive numeric types, `false` for `boolean` values and `null` for references).
- If your class declares any constructors, the compiler will not create a default constructor.
  - In this case, you must declare a no-argument constructor if default initialization is required (i.e., you want to initialize objects with `new ClassName()`).

# Outline

- A deeper look at designing classes, controlling access to class members and creating constructors

- Reexamine the use of *set* and *get* methods (setter & getter)

- Composition (组合)—a capability that allows a class to have references to objects of other classes as members (has-a relationship)

# Notes on *Set and Get Methods*

- Classes often provide `public` methods to allow clients to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) `private` instance variables.
- *Set* methods are also called mutator methods, because they typically change an object's state by modifying the values of instance variables.
- *Get* methods are also called accessor methods or query methods.

```
private int hour;

public void setHour(int h) { hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); }

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

# Notes on *Set and Get Methods*

- The set and get methods are used in many other methods even when these methods can directly access the class's private data

```
public class Time2 {
  private int hour;
  private int minute;
  private int second;

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

  public String toString() {
    return String.format("%d:%02d:%02d %s",
      ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
      getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM") );
  }
}
```

Why not directly accessing the fields?

# Suppose we directly access fields…

- Someday, if we want to optimize the program by using only one `int` variable (4 bytes of memory) to store the number of seconds elapsed since midnight rather than three `int` variables (12 bytes of memory)

```
public class Time2 {
  private int hour;
  private int minute;
  private int second;

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

  public String toString() {
    return String.format("%d:%02d:%02d %s",
      ( (hour == 0 || hour == 12) ? 12 : hour % 12 ),
      second, second, (hour < 12 ? "AM" : "PM") );
  }
}
```

```
private int totalElapsedSeconds;
```

We need to modify all methods: `getHour, getMinute, getSecond, setHour, setMinute, setSecond, toUniversalString, toString...`

# If We Use *Set* and *Get* Methods

- We only need to modify: `getHour`, `getMinute`, `getSecond`, `setHour`, `setMinute`, `setSecond`
- No need to modify `toUniversalString`, `toString` etc. because they do not access the private data directly.

Designing the class this way reduces the likelihood of programming errors when altering the class's implementation

```
public class Time2 {
public String toUniversalString() {
    return String.format("%02d:%02d:%02d",
      getHour(), getMinute(), getSecond());
  }
  public String toString() {
    return String.format("%d:%02d:%02d %s",
      ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
      getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM") );
  }
}
```

# Code Reuse (Avoid Duplications)

```
public class Time2 {
  public Time2(int h, int m, int s) {
    setTime(h, m, s);
  }
  public Time2(int h, int m) {
    this(h, m, 0);
  }
  public Time2(int h) {
    this(h, 0, 0);
  }
  public Time2() {
    this(0, 0, 0);
  }
  public Time2(Time2 time) {
    this(time.getHour(), time.getMinute(),
         time.getSecond());
  }
}
```

- Similarly, each `Time2` constructor could be written to include a copy of the statements from methods `setHour`, `setMinute` and `setSecond`.
  - Doing so may be slightly more efficient, because the extra constructor call and call to `setTime` are eliminated.
  - However, duplicating statements in multiple methods or constructors makes changing the class's internal data representation more difficult.
  - Having the `Time2` constructors call the three-argument constructor requires any changes to the implementation of time setting to be made only once (by changing `setTime`).

# More on Data Hiding and Integrity

- It seems that providing *set* and *get* capabilities is essentially the same as making the instance variables `public`.
  - A `public` instance variable can be read or written by any method that has a reference to an object that contains that variable.
  - If an instance variable is declared `private`, a `public` *get* method certainly allows other methods to access it, but the *get* method can control how the client can access it (e.g., get only the last 4 digits of a phone number).
  - A `public` *set* method can—and should—carefully scrutinize attempts to modify the variable's value to ensure that the new value is valid for that data item.

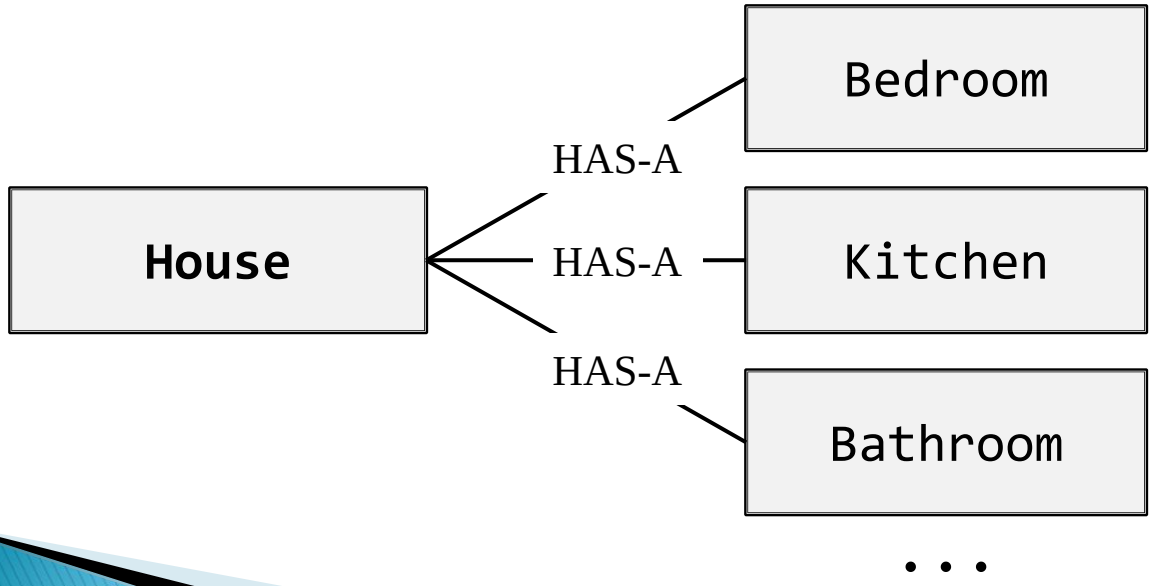
```
public int hour; // this makes coding easier, but...
public int minute;
public int second;
```

# Outline

- A deeper look at designing classes, controlling access to class members and creating constructors
- Reexamine the use of *set* and *get* methods (setter & getter)
- Composition (组合)—an approach to reusing code

# Composition

- A class can have references to objects of other classes as members.
- This is called composition and is sometimes referred to as a has-a relationship.



# Designing an `Employee` Class

- Suppose we are designing an Employee Management System, what information should be included in the `Employee` class?

First name (`String` type)

Last name (`String` type)

Date of birth (? type)

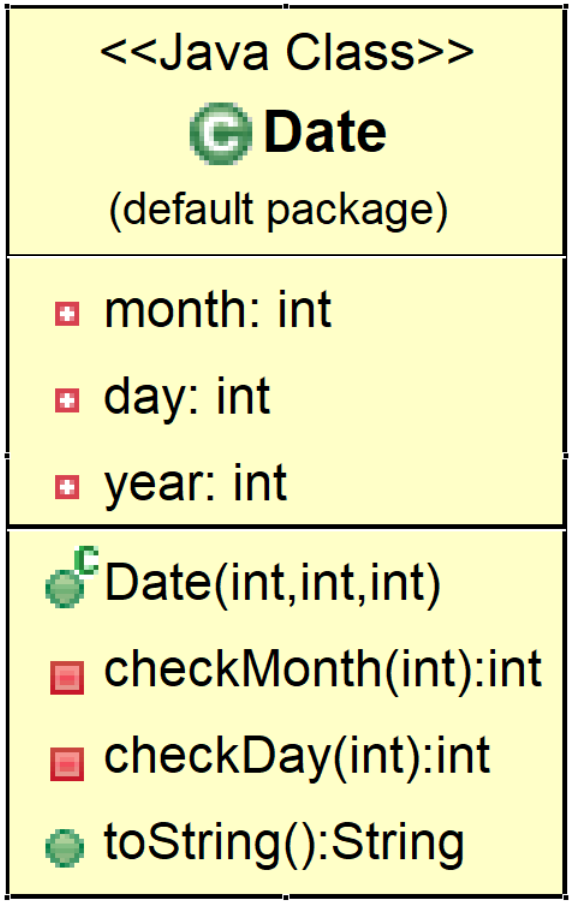
Date of hiring (? type)

… potentially lots of other information

# Let's Define a `Date` Class

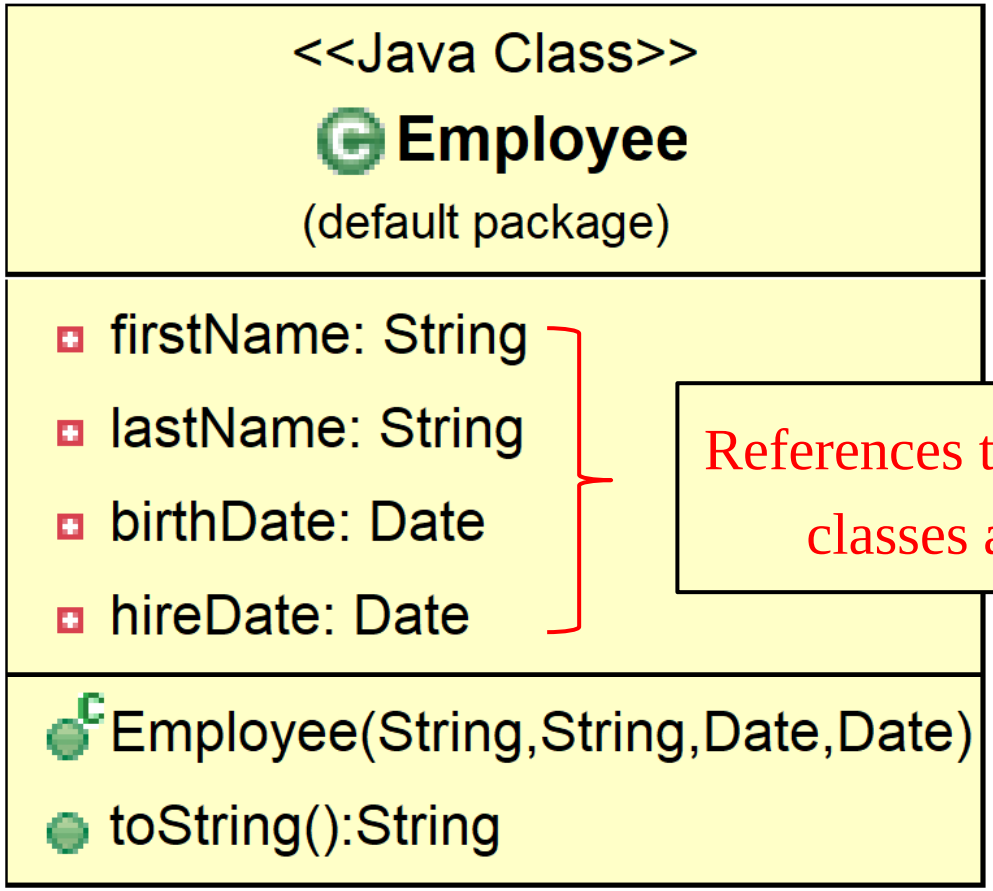
- What kind of information (stored in instance variables) should be included?

- What kind of operations (methods) should be included?




This UML class diagram is automatically generated by Eclipse with a plugin named ObjectAid

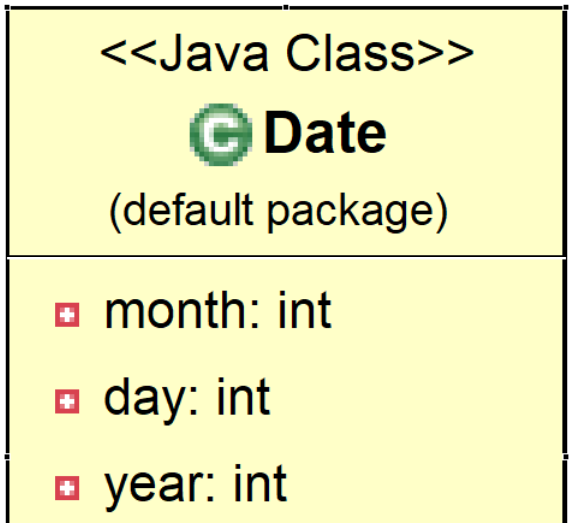
# Define the `Employee` class




References to objects of `String` and `Date` classes as members **(composition)**

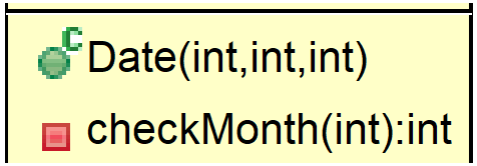
# Let's Look at the Real Code

```
public class Date {

        private int month;

        private int day;

        private int year;

}
```




We make the instance variables private for data hiding.

# Let's Look at the Real Code

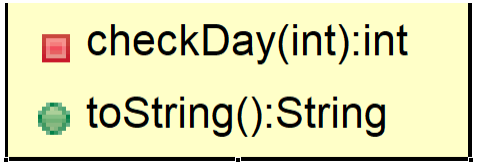



```
public Date(int theMonth, int theDay, int theYear) {
    month = checkMonth(theMonth);
    year = theYear;
    day = checkDay(theDay);
    System.out.printf("Date object constructor for date %s\n", this);
}

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

Constructor performs data validation

Data validation

# Let's Look at the Real Code

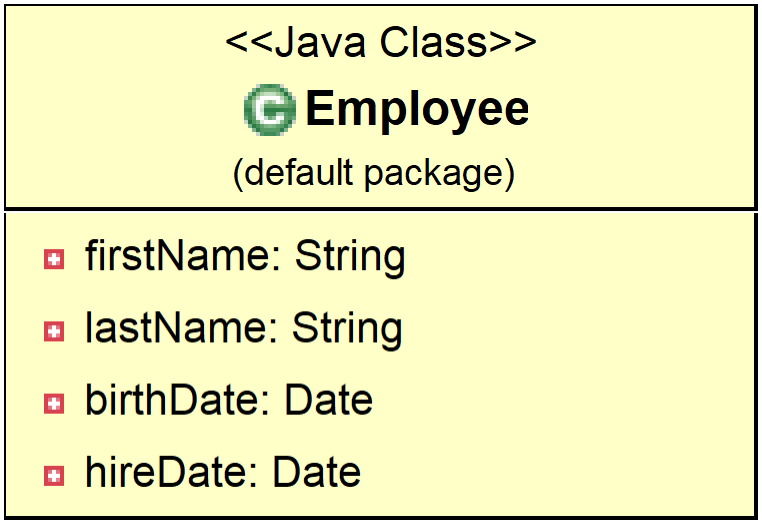



```
private int checkDay(int testDay) { // data validation
    int[] daysPerMonth =
          { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    if(testDay > 0 && testDay <= daysPerMonth[month]) return testDay;
    if(month == 2 && testDay == 29 && (year % 400 == 0 ||
      (year % 4 == 0 && year % 100 != 0)))
        return testDay;
    System.out.printf("Invalid day (%d), set to 1", testDay);
    return 1;
}

public String toString() { // transform object to String representation
    return String.format("%d/%d/%d", month, day, year);
}
```

# Let's Look at the Real Code

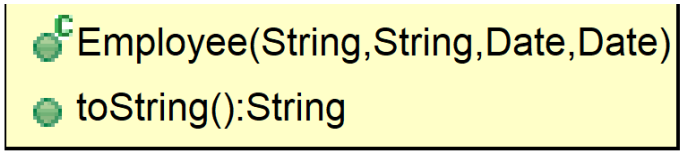
```
public class Employee {

        private String firstName;

        private String lastName;

        private Date birthDate;

        private Date hireDate;

}
```




Again, we make the instance variables private for data hiding.

# Let's Look at the Real Code

```
public Employee(String first, String last, Date dateOfBirth,
                Date dateOfHire) { // constructor
    firstName = first;
    lastName = last;
    birthDate = dateOfBirth;
    hireDate = dateOfHire;
}

public String toString() { // to String representation
    return String.format("%s, %s Hired: %s Birthday: %s",
            lastName, firstName, hireDate, birthDate);
}
```



# Let's Run the Code

```
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("Bob", "Blue", birth, hire);
        System.out.println(employee);
    }
}
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949
```

# Quiz 2

What is the output?

```
class Test {
    int i;
}
class Main {
    public static void main(String args[]) {
        Test t = null;
        System.out.println(t.i);
    }
}
```

- 0
- null
- Compilation error
- Runtime error

The code compiles and could be executed, but throws NPE during runtime

# Quiz 2 extended

```
class Test {
    int i;
    Dog dog;
}

class Quiz2 {
    public static void main(String[] args){
        Test t = new Test();
        System.out.println(t.i);
        System.out.println(t.dog);
    }
}
```