Chapter 8
Classes and Objects: A Deeper Look (I)

TAO Yida

taoyd@sustech.edu.cn

Objectives

- A deeper look at designing class
  - Access control
  - Data validation
  - Data encapsulation / Data hiding
- Composition
- Static members vs. instance members of a class

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
```

Using the `Time` Class

```
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
```

Using the `Time` Class

```
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());
    }
}
```

```
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

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 and you accidently set the time to 19 minutes, 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 – Design 1

- Current `setTime` 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 – Design 2

- 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 (do nothing about) 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;
}
```

Handling Invalid Values

- Designs 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;
}
```

Handling Invalid Values – Design 3

- `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.

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

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;
    }
}
```

The use of `this` is to differentiate the formal parameters of methods and the data members of classes with the same name.

this Reference

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

```
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;
    }
}
```

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

.

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.

Constructors

```
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

Constructors (cont.)

```
  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

```
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

```
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

```
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());
  }
}
```

- 'this()' can only be called in constructors
- Call to 'this()' must be first statement in constructor body

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
```

Controlling Access to Members

- Access modifiers `public` and `private` control access to a class's members, i.e., variables and methods.
- `private` class members can only be accessed from within the class, and are not accessible outside the class.
- `public` class members can be accessed by any other classes.

Accessing Private Members

```
public class Time1 {
   private int hour;
   private int minute;
   private int 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 class TimeTest {

      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

      }

}
```

```
public int hour;
public int minute;
public int second;
```

How about this?

We use `private` to protect data fields, and use `public` setters and getters to control the access to the data

Data Encapsulation (数据封装)

- 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.
- *Setter* methods are also called mutator methods, because they typically change an object's state by modifying the values of instance variables.
- *Getter* 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; }
```

Design I

```
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 ),
      minute, second, (hour < 12 ? "AM" : "PM") );
  }
}
```

Design II

```
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") );
  }
}
```

Both code works.
Which is a better design?

Data Encapsulation

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


}
```

- Someday, we may want to alter the data fields
- We may want to use an `int[]` to store hour, minute, and second
- We may use 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)

Design I

```
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 ),
      minute, second, (hour < 12 ? "AM" : "PM") );
  }
}
```

In such cases, we need to modify all methods that directly access the private data members: getHour, getMinute, getSecond, setHour, setMinute, setSecond, toUniversalString, toString...

Design II

```
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") );
  }
}
```

- 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.

Design Guide

- Declare class data fields private.
  - To prevent data from being tampered with
  - To make the class easy to maintain (hide the implementation details), i.e., we can change the underlying implementation of data fields without breaking client code

- Declare methods public.
  - We have control over how fields are accessed (getters) and modified (setters)
  - Clients can use public methods without worrying about internal details

Objectives

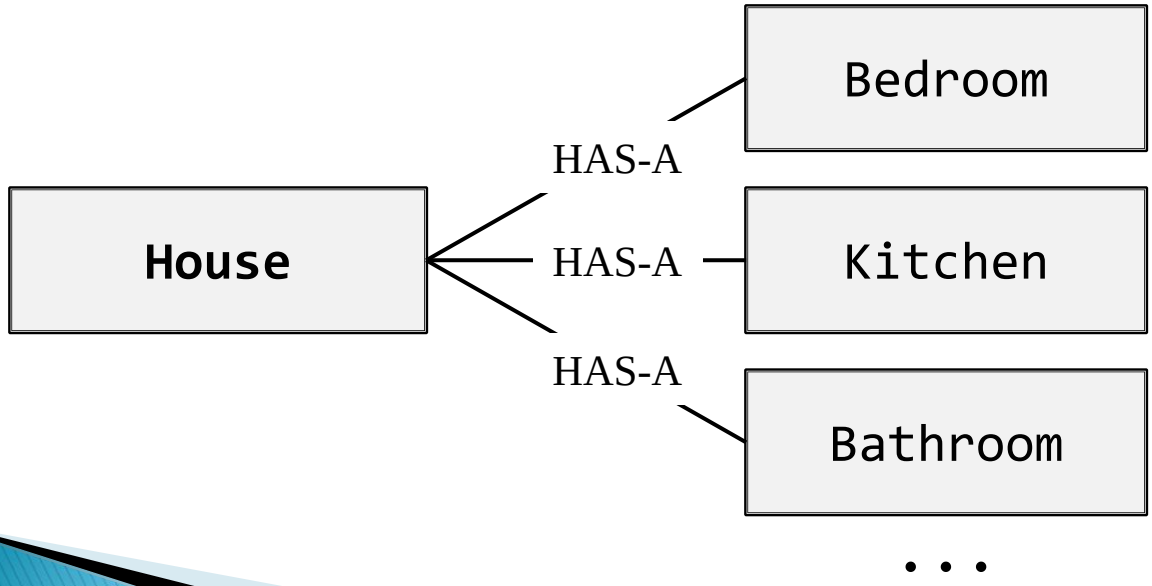
- A deeper look at designing class
  - Access control
  - Data validation
  - Data encapsulation / Data hiding
- Composition
- Static members vs. instance members of a class

Code Reuse

- The essence of code reuse is that you don't need to rewrite the code
- In OOP, code reuse means that new classes (types) are built on the basis of already existing classes (types).
  - Composition (组合): a class contains an object of another class. This approach uses the functionality of the finished code;
  - Inheritance (继承): Later

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)

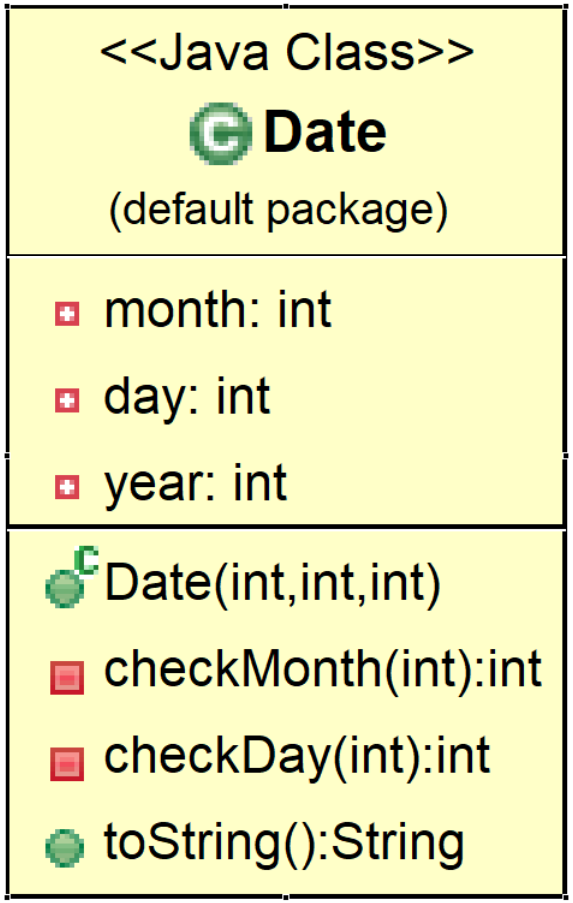
Date of birth (? type)

Date of hiring (? type)

… potentially lots of other information

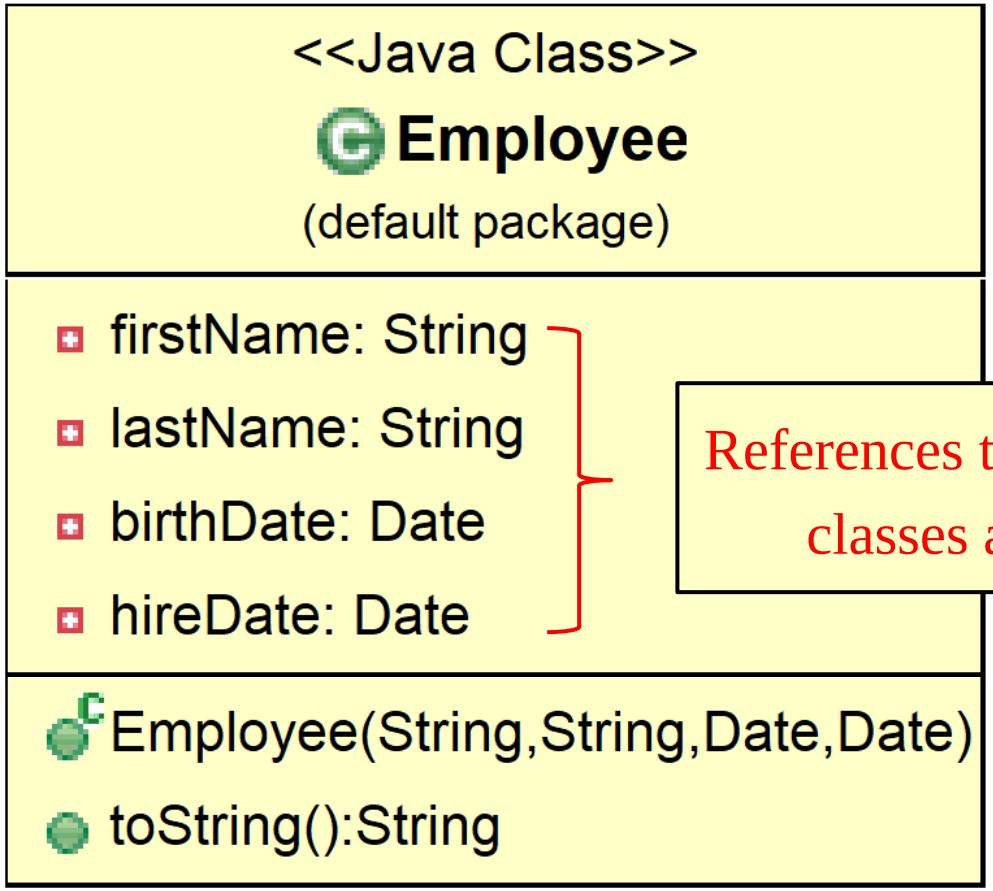
Define the `Date` Class

- 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

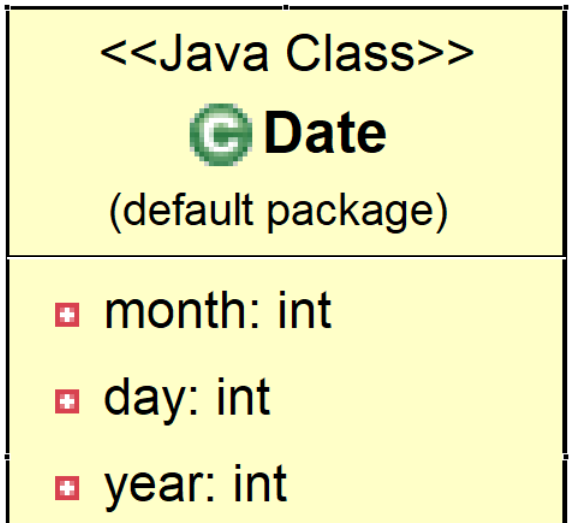
Define the `Employee` class



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

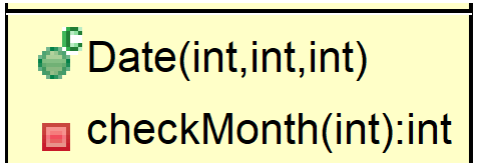
Let's Look at the Real Code

```
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

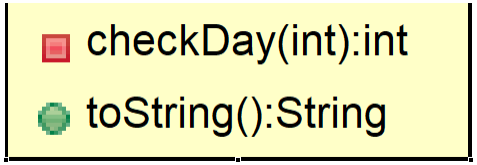



```
public Date(int theMonth, int theDay, int theYear) {
    month = checkMonth(theMonth);
    year = theYear;
    day = checkDay(theDay);
    System.out.printf("Date object constructor for date %s\n", this);
}

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

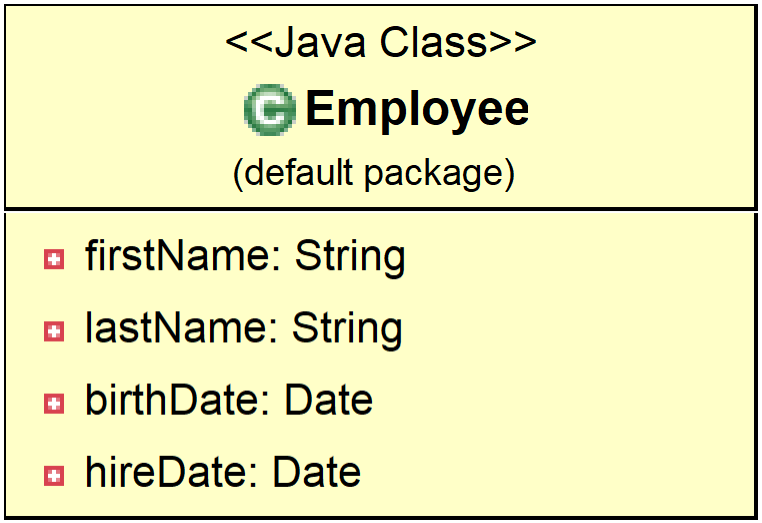
Let's Look at the Real Code




```
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

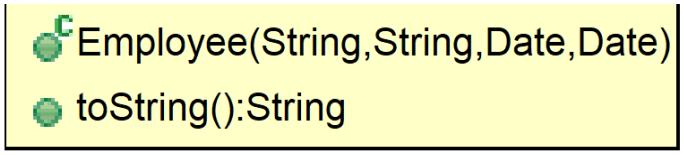
```
public class Employee {

        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
```

Objectives

- A deeper look at designing class
  - Access control
  - Data validation
  - Data encapsulation / Data hiding
- Composition
- Static members vs. instance members of a class

`static` Class Members

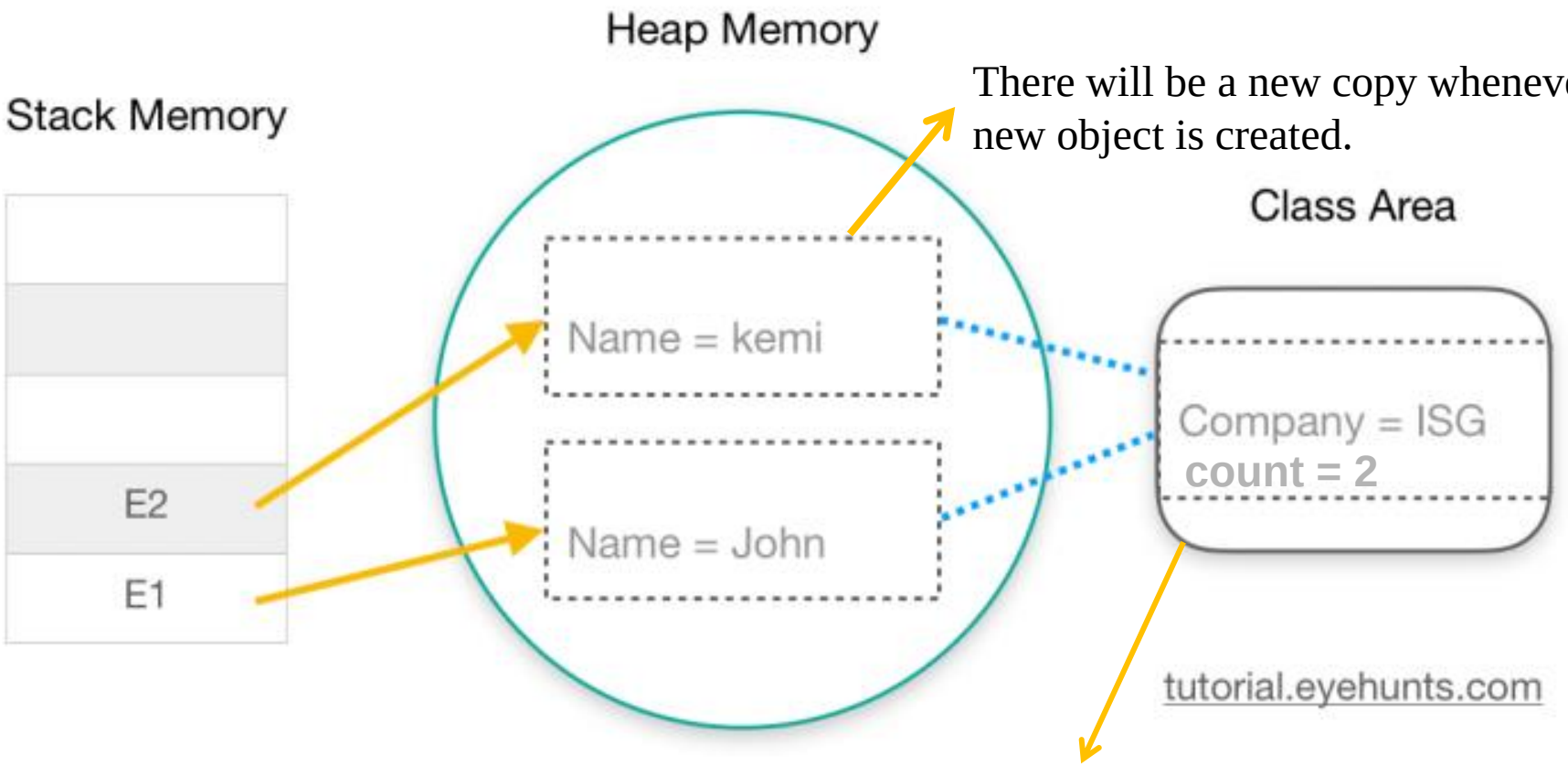
- A `static` variable represents class-wide information. All objects of the class share the same piece of data.

```
public class Employee {

    private String name;

    private static int count; // number of employees created

}
```

There will be a new copy whenever a new object is created.

There is only one copy for each static variable. Make a variable `static` when all objects of the class must use the same copy of the variable.

`static` Class Members



There is only one copy for each static variable. Make a variable `static` when all objects of the class must use the same copy of the variable.

Example

```
public class Employee {
    private String firstName;
    private String lastName;
    private static int count; // number of employees created

    public Employee(String first, String last) {
        firstName = first;
        lastName = last;
        ++count;
        System.out.printf("Employee constructor: %s %s; count = %d\n",
                          firstName, lastName, count);
    }

    public String getFirstName() {  return firstName;  }

    public String getLastName() {  return lastName;  }

    public static int getCount() {  return count;  }
}
```

Example

```
public class EmployeeTest {
  public static void main(String[] args) {
    System.out.printf("Employees before instantiation: %d\n",
                      Employee.getCount());
    Employee e1 = new Employee("Bob", "Blue");
    Employee e2 = new Employee("Susan", "Baker");
    System.out.println("\nEmployees after instantiation:");
    System.out.printf("via e1.getCount(): %d\n", e1.getCount());
    System.out.printf("via e2.getCount(): %d\n", e2.getCount());
    System.out.printf("via Employee.getCount(): %d\n", Employee.getCount());
    System.out.printf("\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
                      e1.getFirstName(), e1.getLastName(),
                      e2.getFirstName(), e2.getLastName());
  }
}
```

The only way to access private static variables at this stage

More choices when there are objects

Example

```
Employees before instantiation: 0
Employee constructor: Bob Blue; count = 1
Employee constructor: Susan Baker; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Bob Blue
Employee 2: Susan Baker
```

Access the same variable

`static` Class Members

- `getCount()` is a `static` method
- `getCount()` accesses a static variable `count`

```
public class Employee {
    private String name;
    private static int count; // number of employees created
    public static int getCount() {  return count;  }
}
```

Can getCount() access the instance variable name?

Can we make getCount() an instance method?

```
public class Employee {
    private static int count;
    private String name;

    public static void m1(){
        count++;
        m3();

        System.out.println(name);
        m4();
    }

    public void m2(){
        count++;
        m3();

        System.out.println(name);
        m4();
    }

    public static void m3(){}

    public void m4(){}
}
```

- `count` is static/class variable
- `name` is instance variable

Static method m1

- can access static variable count and static method m3
- CANNOT access instance variable name and instance method m4

- A `static` method CANNOT access non-static class members (instance variables, instance methods), because a static method can be called even when no objects of the class have been instantiated.
- For the same reason, the `this` reference cannot be used in a `static` method.

```java
public class Employee {
    private static int count;
    private String name;

    public static void m1(){
        count++;
        m3();

        System.out.println(name);
        m4();
    }

    public void m2(){
        count++;
        m3();

        System.out.println(name);
        m4();
    }

    public static void m3(){}

    public void m4(){}
}
```

- `count` is static/class variable
- `name` is instance variable

Instance method m2

- can access static variable count and static method m3
- can access instance variable name and instance method m4


```java
public class Employee {
    private static int count;
    private String name;

    public static int getCount() {
        return count;
    }

    public static void main(String[] args) {

        System.out.println(count);

        System.out.println(name);

        Employee e = new Employee();
        System.out.println(e.name);

        System.out.println(e.count);
        System.out.println(e.getCount());

        System.out.println(Employee.count);
        System.out.println(Employee.getCount());
    }
}
```

main(String[] args) is also a static method

OK.

- `main()` can access private `count` for being inside the same class
- `main()` can access static `count` as it's a static method itself
- `count` is not initialized explicitly, the compiler assigns it a default value (e.g., 0 for `int`)

```java
public class Employee {
    private static int count;
    private String name;

    public static int getCount() {
        return count;
    }

    public static void main(String[] args) {

        System.out.println(count);

        System.out.println(name);

        Employee e = new Employee();
        System.out.println(e.name);

        System.out.println(e.count);
        System.out.println(e.getCount());

        System.out.println(Employee.count);
        System.out.println(Employee.getCount());
    }
}
```

main(String[] args) is also a static method

NO. A static method CANNOT access an instance variable.

We need to create an instance first before accessing name.

```java
public class Employee {
    private static int count;
    private String name;

    public static int getCount() {
        return count;
    }

    public static void main(String[] args) {

        System.out.println(count);

        System.out.println(name);

        Employee e = new Employee();
        System.out.println(e.name);

        System.out.println(e.count);
        System.out.println(e.getCount());

        System.out.println(Employee.count);
        System.out.println(Employee.getCount());
    }
}
```

main(String[] args) is also a static method

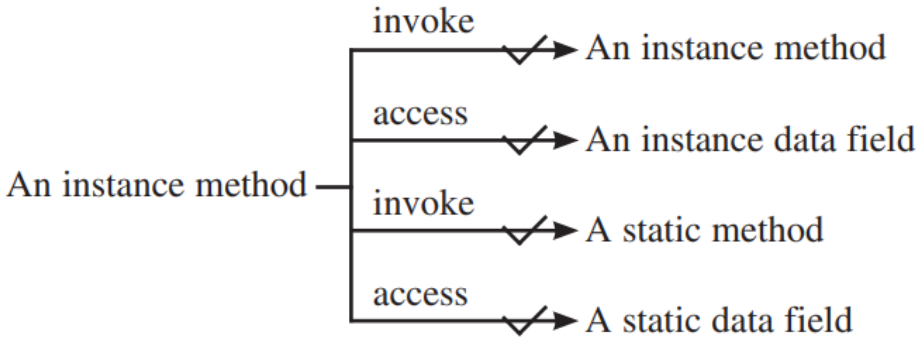
OK. A class's `static` member can be accessed through a reference to any object of the class. But this is NOT recommended.

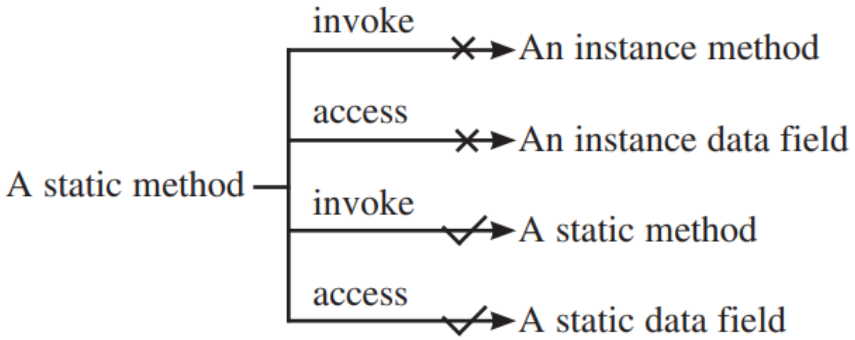

```java
public class Employee {
    private static int count;
    private String name;

    public static int getCount() {
        return count;
    }

    public static void main(String[] args) {

        System.out.println(count);

        System.out.println(name);

        Employee e = new Employee();
        System.out.println(e.name);

        System.out.println(e.count);
        System.out.println(e.getCount());

        System.out.println(Employee.count);
        System.out.println(Employee.getCount());

    }
}
```

main(String[] args) is also a static method

OK and recommended. A class's `static` member can be accessed by qualifying the member name with the class name and a dot (.), e.g., `Math.PI`

`static` Methods vs `instance` Methods





`static` members in Class `Math`

- Fields and methods in class `Math` are `public` and `static`
  - `public` allows you to use these fields anywhere in your own classes
  - `static` makes them accessible via the class name `Math` and a dot (`.`) separator

- Class `Math` declares commonly used mathematical constants
  - `Math.PI` (3.141592653589793)
  - `Math.E` (2.718281828459045) is the base value for natural logarithms

Many more useful static methods in `java.lang.Math` class:

Method	Description	Example

`abs( x )`	absolute value of x	`abs( 23.7 )` is `23.7`
		`abs( 0.0 )` is `0.0`
		`abs( -23.7 )` is `23.7`

`ceil( x )`	rounds x to the smallest integer not less than x	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of x (x in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	exponential method e^x	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`
`floor( x )`	rounds x to the largest integer not greater than x	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`log( x )`	natural logarithm of x (base e)	`log( Math.E )` is `1.0` `log( Math.E * Math.E )` is `2.0`
`max( x, y )`	larger value of x and y	`max( 2.3, 12.7 )` is `12.7` `max( -2.3, -12.7 )` is `-2.3`
`min( x, y )`	smaller value of x and y	`min( 2.3, 12.7 )` is `2.3` `min( -2.3, -12.7 )` is `-12.7`
`pow( x, y )`	x raised to the power y (i.e., x^y)	`pow( 2.0, 7.0 )` is `128.0` `pow( 9.0, 0.5 )` is `3.0`
`sin( x )`	trigonometric sine of x (x in radians)	`sin( 0.0 )` is `0.0`
`sqrt( x )`	square root of x	`sqrt( 900.0 )` is `30.0`
`tan( x )`	trigonometric tangent of x (x in radians)	`tan( 0.0 )` is `0.0`

Static Method `valueOf` in `String`

- Every object in Java has a `toString` method that enables a program to obtain the object's `String` representation.
- Unfortunately, this technique cannot be used with primitive types because they do not have methods.
- Class `String` provides `static` methods (associated with class, no need to create objects for their invocation) that take an argument of any type and convert it to a `String` object.

```
static String    valueOf(boolean b)
static String    valueOf(char c)
static String    valueOf(char[] data)
static String    valueOf(char[] data, int offset, int count)
static String    valueOf(double d)
static String    valueOf(float f)
static String    valueOf(int i)
static String    valueOf(long l)
```

```
boolean booleanValue = true;
char charValue = 'Z';
int intValue = 7;
long longValue = 10000000000L;
float floatValue = 2.5f;
double doubleValue = 33.3333; // no f suffix, double is default
char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
System.out.println(String.valueOf(booleanValue));
System.out.println(String.valueOf(charValue));
System.out.println(String.valueOf(intValue));
System.out.println(String.valueOf(longValue));
System.out.println(String.valueOf(floatValue));
System.out.println(String.valueOf(doubleValue));
System.out.println(String.valueOf(charArray));
```

```
true
Z
7
10000000000
2.5
33.3333
abcdef
```

