Classes and Objects: A Deeper Look

Java™ How to Program, 11th Edition

Instructor: Zhuozhao Li

static Versus Instance Variables

- Recall that every object of a class has its own copy of all the instance variables of the class.
  - Instance variables represent concepts that are unique per instance, e.g., name in class Student.

- In certain cases, only one copy of a particular variable should be shared by all objects of a class (e.g., a counter that keeps track of every object created for memory management).
  - A static field—called a class variable—is used in such cases.

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 firstName;

    private String lastName;

    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

- static class members are available as soon as the class is loaded into memory at execution time (objects may not exist yet)
- A class's public static members can be accessed through a reference to any object of the class, or by qualifying the member name with the class name and a dot (.), e.g., Math.PI

```
public class Employee { ...
  public static int count; // number of employees created

  public static void main(String[] args) {
    Employee e = new Employee();
    System.out.printf("# employees = %d", e.count); // not encouraged
    System.out.printf("# employees = %d", Employee.count); // good practice
  }
}
```

static Class Members

- A class’s private static members can be accessed by client code only through methods of the class

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

    public static int getCount() {  return count;  }

    public static void main(String[] args) {
        System.out.printf("# employees = %d", Employee.getCount());
    }
}
```

`static` Class Members

- A `static` method cannot access non-static class members (e.g., instance variables), because a static method can be called even when no objects of the class have been instantiated.

- If a `static` variable is not initialized, the compiler assigns it a default value (e.g., 0 for `int`)

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;  }
}
```

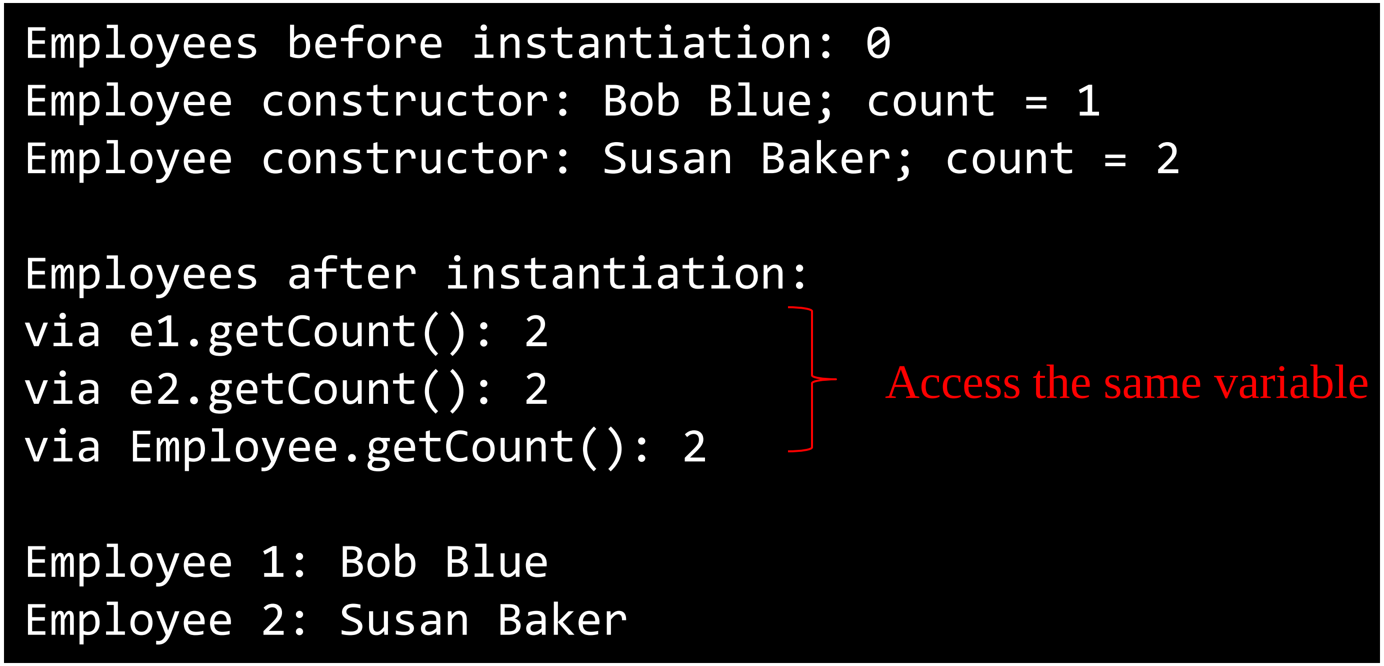
Example

```java
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 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` Import

- Normal import declarations import classes from packages, allowing them to be used without package qualification
- A `static` import declaration enables you to import the `static` members of a class (or interface) so you can access them via their unqualified names, i.e., without including class name and a dot (`.`)
  - `Math.sqrt(4.0)` → `sqrt(4.0)`

`static` Import

- Import a particular `static` member (single `static` import)
  - `import static` *`packageName.ClassName.staticMemberName`*`;`
  - The member could be a field or a method
- Import all `static` members of a class (`static` import on demand)
  - `import static` *`packageName.ClassName.`*`*;`
  - * is a wildcard (通配符), meaning “matching all”

Example

```
import static java.lang.Math.*;

public class StaticImportTest {
    public static void main(String[] args) {
        System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
        System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
        System.out.printf("E = %f%n", E);
        System.out.printf("PI = %f%n", PI);
    }
}
```

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

```java
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.

```java
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

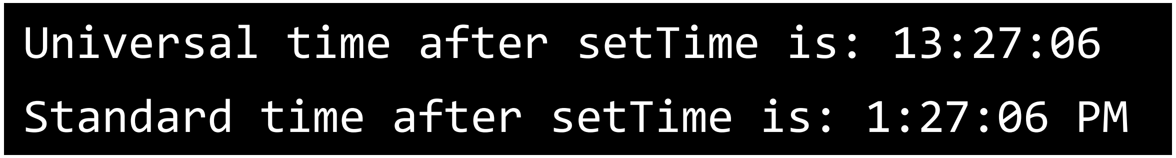
```
public class Time1Test {
    public static void main(String[] args) {
        Time1 time = new Time1(); // invoke default constructor
        System.out.print("The initial universal time is: ");
        System.out.println(time.toUniversalString());
        System.out.print("The initial standard time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

```java
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
```

Handling Invalid Values

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

```java
// 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
}
```

Is there a better 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 consistent 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 were `false`, would attempt to set the time again.

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

- In Exception Handling, we'll learn techniques that enable methods to indicate when invalid values are received.

Data Hiding (Information Hiding)

- The instance variables `hour`, `minute` and `second` are each declared `private`.
- *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)

- Clients could use the same `public` methods and get the same results without being aware of this.

```
// 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() {...}
```

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` a random value).

this Reference

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

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 a 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
        hour
}
```

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.

Quiz

`this` reference can be used in a `static` method

A. True

[✓ B.] False

- `this` is used to access instance variables
- A `static` method cannot access any instance variables

Overloaded Constructors

- **Method overloading:** 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.,
- 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 {
  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. Cannot use Time2(…) here, which can only be used with the “new” keyword.

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 (e.g., 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()`).

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;~~
```

```
private int totalElapsedSeconds;
```

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

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.

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

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

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.
  - A `public` *set* method can—and should—carefully scrutinize attempts to modify the variable's value to ensure that the new value is consistent for that data item.

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

final Instance Variables

- The principle of least privilege （最小权限） is fundamental to good software engineering
  - Code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more.
  - Makes your programs more robust by preventing code from accidentally (or maliciously 恶意地) modifying variable values and calling methods that should not be accessible.

`final` Instance Variables

- The keyword `final` specifies that a variable is not modifiable (i.e., constant) and any attempt to modify leads to an error (cannot compile)

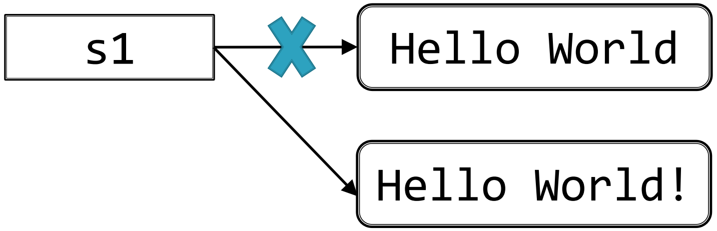
```
private final int INCREMENT;
```

- `final` variables can be initialized when they are declared.
- If they are not, they must be initialized in every constructor of the class.
- Initializing `final` variables in constructors enables each object of the class to have a different value for the constant
- If a `final` variable is not initialized when it is declared or in every constructor, the program will not compile.

Garbage Collection

- Every object uses system resources, such as memory
- We need a disciplined way to give resources back to the system when they’re no longer needed; otherwise, resource leaks may occur.
- The JVM performs automatic garbage collection to reclaim the memory occupied by objects that are no longer used (no references to them).

```java
String s1 = "Hello World";
s1 = s1.concat("!");
```



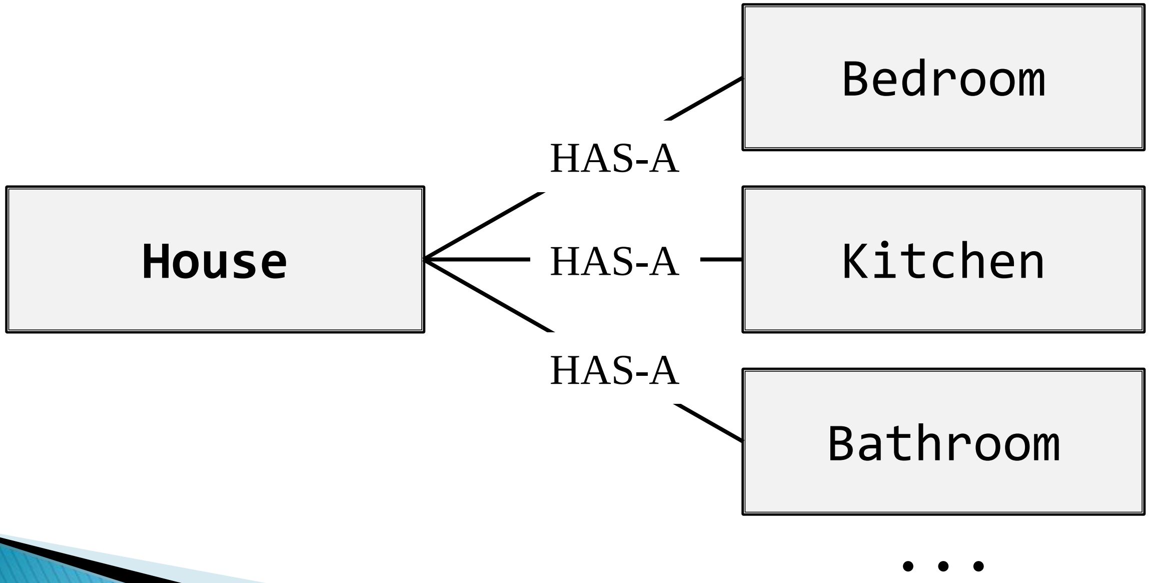
Becomes garbage
if not referenced

Garbage Collection

- With garbage collection, memory leaks （内存泄漏） that are common in other languages like C and C++ (memory is not automatically reclaimed in those languages) are less likely in Java, but some can still happen in subtle ways.

Composition （组成）

- 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)

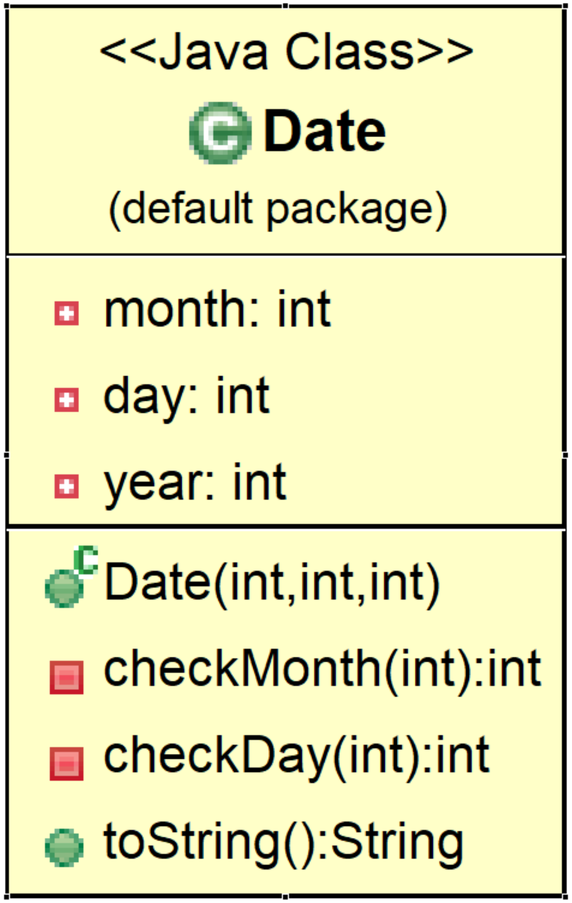
Date of hiring (? type)

… 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?



<<Java Class>>

Date

(default package)

month: int

day: int

year: int

Date(int,int,int)

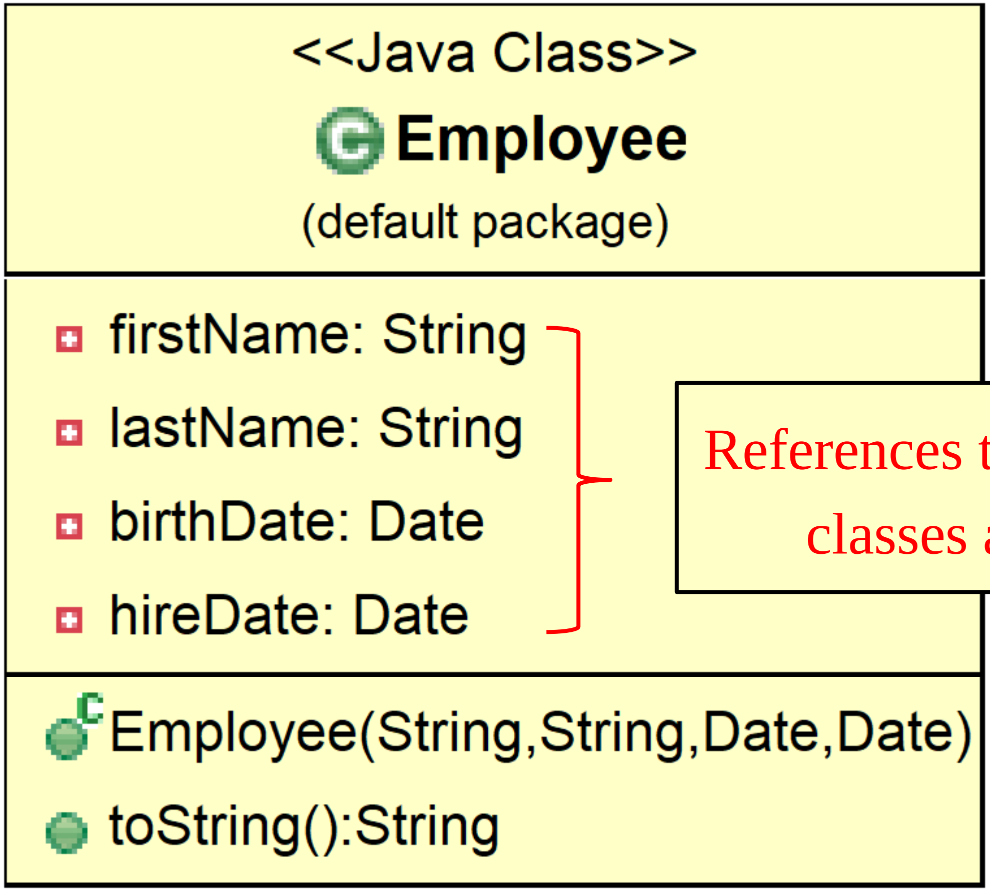
checkMonth(int):int

checkDay(int):int

toString():String

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

Define the Employee class


<<Java Class>>

Employee

(default package)

firstName: String

lastName: String

birthDate: Date

hireDate: Date

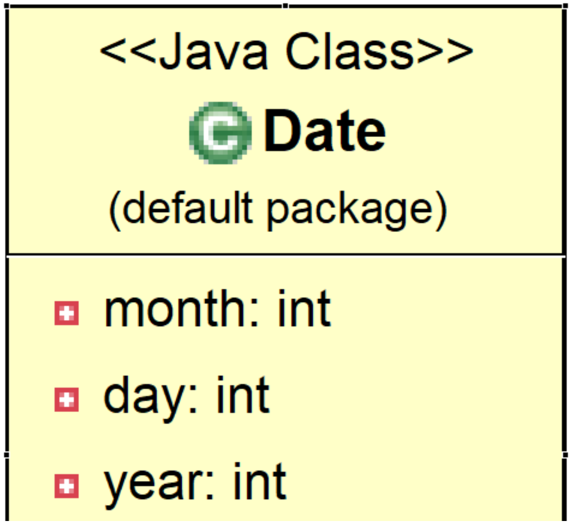
Employee(String,String,Date,Date)

toString():String

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;

}
```



<<Java Class>>

Date

(default package)

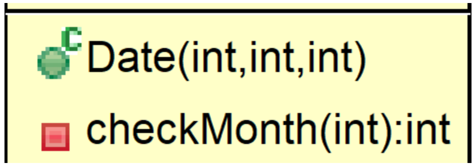
month: int

day: int

year: int

We make the instance variables private for data hiding.

Let’s Look at the Real Code



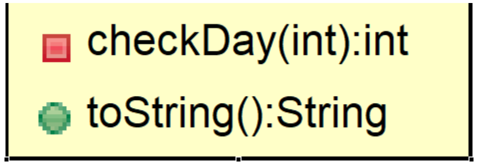
Date(int,int,int)
checkMonth(int):int

```
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



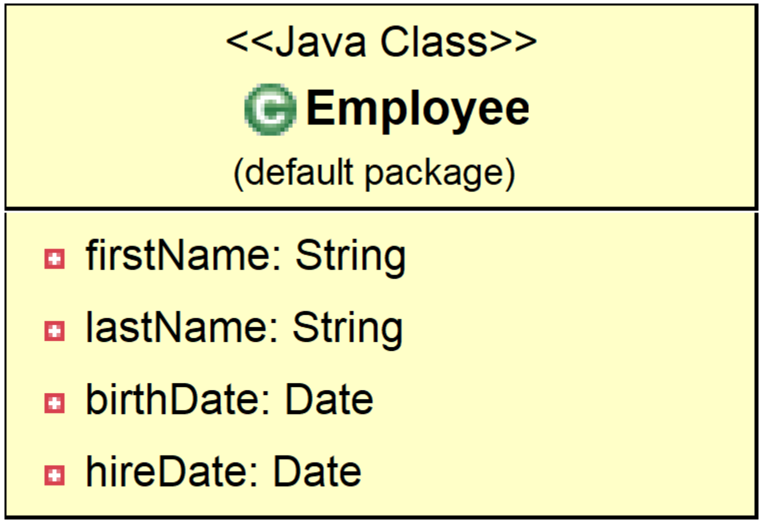
🟥 checkDay(int):int

🟢 toString():String

```
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;

}
```



<<Java Class>>

Employee

(default package)

firstName: String

lastName: String

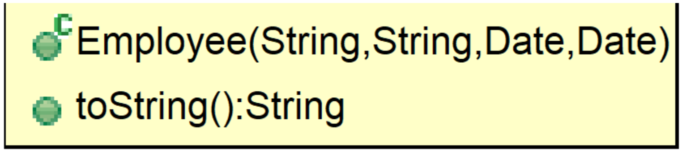
birthDate: Date

hireDate: Date

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

Let’s Look at the Real Code

```java
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);
}
```



Employee(String,String,Date,Date)

toString():String

Let’s Run the Code

```java
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
```

Creating Packages

- Each class in the Java API belongs to a package that contains a group of related classes.
- Packages help programmers manage the complexity of application components.
- Packages facilitate software reuse by enabling programs to import classes from other packages, rather than copying the classes into each program that uses them.
- Packages provide a convention for unique class names, which helps prevent class-name conflicts.

Declaring a reusable class

- **Step 1:** Declare a `public` class
- **Step 2:** Choose a package name and add a `package` declaration to the source file for the reusable class declaration.
  - In each Java source file there can be only one `package` declaration, and it must precede all other declarations and statements.

```
package course.cs102A;

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

Creating Packages (Cont.)

- Placing a `package` declaration at the beginning of a Java source file indicates that the class declared in the file is part of the specified package.
- A Java source file must have the following order:
  - a `package` declaration (if any)
  - `import` declarations (if any)
  - class declarations (you can declare multiple classes in one `.java` file)
- Only one of the class declarations in a particular file can be `public`.
- Other classes in the file are placed in the package and can be used only by the other classes in the package. Non-`public` classes are in a package to support the reusable classes in the package.

Creating Packages (Cont.)

- When a Java file containing a `package` declaration is compiled, the resulting class file is placed in the directory specified by the declaration.

- The class `Time1` should be placed in the directory

```
course

   cs102A
```

```
package course.cs102A;

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

Creating Packages (Cont.)

- `javac` command-line option `-d` causes the compiler to create appropriate directories based on the class's `package` declaration.
- Example command: `javac -d . Time1.java`
  - specifies that the first directory in our package name should be placed in the current directory (`.`)
  - The compiled classes are placed into the directory that is named last in the `package` declaration
  - `Time1.class` will appear in the directory `./course/cs102A/`

Creating Packages (Cont.)

- `package` name is part of the fully qualified name of a class
  - `course.cs102A.Time1`
- We can use the fully qualified name in programs, or `import` the class and use its simple name (e.g., `Time1`).
- If another package contains a class of the same name, the fully qualified class names can be used to distinguish between the classes in the program and prevent a name conflict

Specifying Classpath (Compilation)

- When compiling a class that uses classes from other packages, `javac` must locate the `.class` files for all these classes.
- The compiler uses a special object called a class loader to locate the classes it needs.
  - The class loader begins by searching the standard Java classes that are bundled with the JDK.
  - Then it searches for optional packages.
  - If the class is not found in the standard Java classes or in the extension classes, the class loader searches the classpath, which contains a list of locations in which classes are stored

Specifying Classpath (Compilation)

- By default, the classpath consists only of the current directory

- The classpath can be modified by
  - providing the `–classpath (–cp)` option to the `javac` compiler
  - setting the `CLASSPATH` environment variable (not recommended).

```
javac -classpath .:/home/avh/classes:/usr/local/java/classes Test.java
```

Specifying Classpath (Compilation)

- The classpath consists of a list of directories or archive files, each separated by a directory separator
  - Semicolon (`;`) on Windows, colon (`:`) on UNIX/Linux/Mac OS X
- Archive files are individual files that contain directories of other files, typically in a compressed format
  - Normally end with the `.jar` or `.zip` file-name extensions
- The directories and archive files specified in the classpath contain the classes you wish to make available to the compiler and the JVM

Package Access

- If no access modifier is specified for a class member when it’s declared in a class, it is considered to have package access.

```
public class Time1 {
        int hour;
        int minute;
        int second;
        void setTime(int h, int m, int s) {...}
}
```

No modifier

The variables and method are package-private, visible only to classes of the same package

Access Level Modifiers (So Far)

Modifier	Class	Package	World
public	Y	Y	Y
no modifier	Y	Y	N
private	Y	N	N

Note that this is for controlling access to class members. At the top level, a class can only be declared as `public` or package-private (no explicit modifier)