O Contents Topic 03 - Objects and Classes

I. Classes and Objects – Casual Preview

A. Simple Class Example 1 - Day (See Lab01 Q1)

Add one more method to the **Day** class:

(1) Fill in the blank according to the comment:

```
// advance the current day object by 1 day
public void advance()
{
   if (isEndOfAMonth())
   {
      if (month==12)
      {
         year=_____;
         month=_____;
         day=_____;
      }
      else
      {

      }
   }
   else
   {

   }
}
```

Add this method

```
public class Day
{
  private int year;
  private int month;
  private int day;

  //Constructor
  public Day(int y, int m, int d)
  {
    this.year=y;
    this.month=m;
    this.day=d;
  }

  ...

  // create and return the "next day" of
  // the current day object
  public Day next()
  {
    if (isEndOfAMonth())
      if (month==12)
        return new Day(year+1,1,1);
    else
        return new Day(year,month+1,1);
    else
      return new Day(year,month,day+1);
  }
}
```

```
class ClassName
{
    field1
    field2
    . . .
    constructor1
    constructor2
    . . .
    method1
    method2
    . . .
}
```

Note: some methods return a result, some do not

(2) Both **.advance()** and **.next()** calculate the next day. Complete the code below based on the comments.

Main.java

```
Day d1 = new Day(2014, 1, 28);
System.out.println(d1.toString());  //Show 28 Jan 2014
_____.advance();                    //Advance one day
_____.next();                       //Advance one day
System.out.println(d1.toString());  //Show 30 Jan 2014
```

B. Simple Class Example 2- Employee

- Consider a simplified `Employee` class used in a payroll system:

```
class Employee
{
   // instance fields
   private String name;
   private double salary;
   private Day hireDay;

   // constructor
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = new Day(year,month,day);
   }

   public String getName() {return name;}

   public double getSalary() {return salary;}

   public Day getHireDay() {return hireDay;}

   public void raiseSalary(double percent)
   {
      double raise = salary * percent /100;
      salary += raise;
   }
}
```

- Alternatively:

Initialization can be done for instance fields.

```
class Employee
{
   // instance fields
   private String name;
   private double salary=0;
   private Day hireDay;

   ..
}
```

- We can even do some method call and computation:

```
private double salary = Math.random()*10000;
```

- Using the `Employee` class in a program:

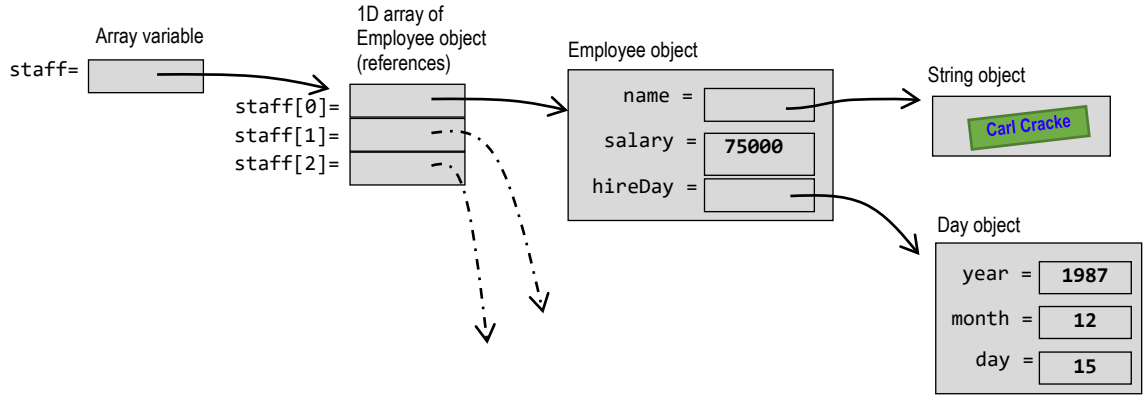
```
public class Main_EmployeeTest
{
   public static void main(String[] args)
   {
      // fill the staff array with three Employee objects
      Employee[] staff = new Employee[3];
      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      // raise everyone's salary by 5%
      for (Employee e : staff)
        e.raiseSalary(5);

      // printing
      for (Employee e : staff)
        System.out.println(
           "name=" + e.getName() +
           ",salary=" + e.getSalary() +
           ",hireDay=" + e.getHireDay());
   }
}
```

Output

```
name=Carl Cracker,salary=78750.0,hireDay=15 Dec 1987
name=Harry Hacker,salary=52500.0,hireDay=1 Oct 1989
name=Tony Tester,salary=42000.0,hireDay=15 Mar 1990
```

C. Array of objects

```
public class Main_EmployeeTest
{
    public static void main(String[] args)
    {
        // fill the staff array with three Employee objects
        Employee[] staff = new Employee[3];
        staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
        staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
        staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
        ..
    }
}
```



D. .toString()

If an object's text representation is needed, Java automatically looks for its .toString() method.

```
public class Main_EmployeeTest
{
  public static void main(String[] args)
  {
    ..
    // printing
    for (Employee e : staff)
      System.out.println(
        "name=" + e.getName() +
        ",salary=" + e.getSalary() +
        ",hireDay=" + e.getHireDay());
  }
}
```

```
class Employee
{
    // instance fields
    private String name;
    private double salary;
    private Day hireDay;
    ..
    public Day getHireDay()
    {
        return hireDay;
    }
    ..
}
```

```
public class Day
{
  private int year;
  private int month;
  private int day;
  ..

  // Return a string for the day (dd MMM yyyy)
  public String toString()
  {
    final String[] MonthNames = {
      "Jan", "Feb", "Mar", "Apr","May", "Jun",
      "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    return day+" "+
      MonthNames[month-1] +
      " "+ year;
  }
  ..
}
```

```
Output:
name=Carl Cracker,salary=78750.0,hireDay=15 Dec 1987
name=Harry Hacker,salary=52500.0,hireDay=1 Oct 1989
name=Tony Tester,salary=42000.0,hireDay=15 Mar 1990
```

Explanation:

- The reference of a `Day` object is returned by `e.getHireDay()`
- However, a string is needed at `",hireDay=" + e.getHireDay());`
- JAVA automatically looks for a `.toString()` method of the `Day` class and invoke it for the `Day` object
- `.toString()` returns a string. Done!

II. Introduction to OOP

OO Programs

- An O-O program is made up of objects

  Each object has
  - a specific functionality exposed to its users
  - A hidden implementation

- Basically, as long as an object satisfies your specifications, you don't care how the functionality is implemented
  - E.g., consider `Scanner` objects: `.nextInt`, `.hasNext`

- Traditional Structured Programming vs OOP
  - Structured Programming: Designing a set of procedures to solve a program

    Usually top-down (starting from `main`())
  - OOP: Puts data first, then looks at the algorithms to operate on the data

    There is no "top".

    Begin by studying a description of the application, and

    identifying classes (often from nouns) and then add methods (often as verbs).

- OOP: More appropriate for larger problems:
  - Because of well-organized code around data
  - E.g., Easier to find bugs which causes faults on a piece of data

Classes

- A class is the template / blueprint from which objects are made.
  - Like cookie cutters from which cookies are made

  - Classes in Java program
  - Provided by the standard Java library
  - Created by ourselves for our application's problem domain

- Terminologies:

  "object" ~ "instance" ~ "object instance"
  - **Instance** of a class : an Object created from a class
    - E.g., **Scanner s; s = new Scanner(..);** // s is an instance of the `Scanner` class
    - E.g., **Day d; d = new Day(2014,1,19);** // d is an instance of the `Day` class
  - **Instance fields**: Data in an object (e.g., `day`, `month`, `year`)
  - **Object state**: The set of values in the instance fields of an object

    Or "how does the object react when its method runs"
  - **Methods**: procedures which operate on the data.
  - **Encapsulation** (sometimes called information hiding)
    - Combine data and behavior in one package, consider as implementation details
    - Implementation details are hidden away from the users of the objects
    - E.g., We use `Scanner` objects, but their implementation details are encapsulated in the `Scanner` class.

III. Objects and Object variables - The Constructors and the new operator

- Work with objects - we often:
  - First: construct them and specify initial state
    (give values for instance fields like `day`, `month`, `year`)
  - Then: apply methods to the objects.

- **Constructors:**
  - We use constructors to construct new instances
  - A constructor is a special method
    - ☆ Purpose: construct and **initialize objects**
    - ☆ Named the same as the **class name**, **no return value**
    - ☆ Always called with **the `new` operator**
      e.g. **`new Day(2014,1,3)`**
    - ☆ A class can have **more than one constructor**

e.g. Add the second constructor to the **Day** class:

```
public Day (int y, int m, int d) {..}
public Day (int y, int nth_dayInYear) {..}
```

Usage of such a constructor:

```
d1=new Day(2014,45); //The 45th day in 2014
System.out.println(d1); //14 Feb 2014
```

- **Examples of Constructors**

```
class Employee
{
   // instance fields
   private String name;
   private double salary;
   private Day hireDay;

   // constructor
   public Employee(String n, double s,
         int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = new Day(year,month,day);
   }
    // ... more methods
}
```

```
public class Day
{
 private int year;
 private int month;
 private int day;

 //Constructor
 public Day(int y, int m, int d)
 {
  this.year=y;
  this.month=m;
  this.day=d;
 }

 //.. more methods
}
```

- **More notes on Constructors**
  - Recall: constructors must be called with the `new` operator

    We **CANNOT** apply it solely for resetting the instance fields like:

    ☒ `birthday.Day(2014,1,27);` // Willing to change the instance fields in birthday?
    "Error: The method Day(int, int, int) is undefined for the type Day"

- **When an object is created, we may:**

1. Pass the object to a method:

   `System.out.println(new Day(2014,1,15)); //` ☆ Note: actually `.toString()` is called.

2. Apply a method to the object just constructed:

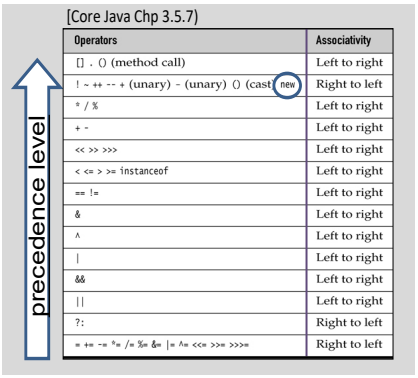
   `System.out.print((new Day(2014,1,15)).next());`

   - Or simply `System.out.print(new Day(2014,1,15).next());`

     Reason: constructor is always called with the `new` operator

   - Which style to write?

     Suggestion: Choose the one you feel comfortable to read.
     But you should be able to understand both styles when you read others' code.



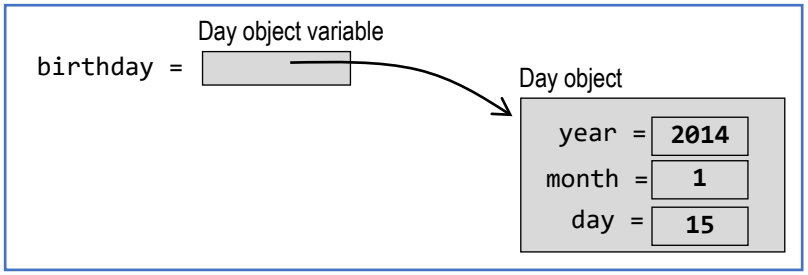
3. Hang on to the object with the use of an object variable:

   `Day birthyday;` //This variable, birthday, doesn't refer to any object yet

   `birthday = new Day(2014,1,15);` //Now, initialize the variable to refer to a new object

   Or, combined as:

   `Day birthday = new Day(2014,1,15);`



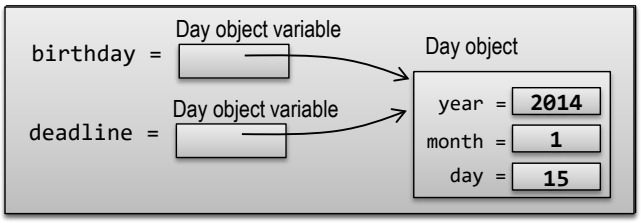
- **An object variable:**

  ☆ Doesn't actually contain an object.

  ☆ The value of any object variable is "a reference to an object that is stored separately".

  ☆ We set a variable to refer to an existing object of the matching type:

```
Day birthday, deadline;
birthday = new Day(2014,1,15);
deadline = birthday;
```



- For convenience, we often **verbally** say *"object"* instead of *"object variable"*.
  But we need to bear in mind the actual picture.

- Note: The `return` value of the `new` operator is also a reference.

- Special value: `null` (means nothing)
  We can set an object variable to `null` to mean that it refers to no object.
  E.g., `Day d1=null; //later checking: if (d1==null) ..`

- Local variables (ie. variables defined in a method) are not automatically initialized.
  To initialize an object variable: we can set it to `null`, or refer to an existing object, or use `new`

```
class Day
{
   private int year;
   private int month;
   private int day;

   public Day(int y, int m, int d)
   {
      this.year=y;
      this.month=m;
      this.day=d;
   }

   public String toString()
   {
      final String[] MonthNames = {
                                   "Jan", "Feb", "Mar", "Apr",
                                   "May", "Jun", "Jul", "Aug",
                                   "Sep", "Oct", "Nov", "Dec"};

      return day+" "+ MonthNames[month-1] + " "+ year;
   }
}
public class Main
{
   private static void testing1()
   {
      Day d1;
      System.out.println(d1);           ① __
      System.out.println(d1.toString()); ② __
   }

   private static void testing2()
   {
      Day d1 = null;
      System.out.println(d1);           ③ __
      System.out.println(d1.toString()); ④ __
   }

   private static void testing3()
   {
      Day d1 = new Day(2014,1,15);
      System.out.println(d1);           ⑤ __
      System.out.println(d1.toString()); ⑥ __
   }

   public static void main(String[] args)
   {
      testing_(); //Change this line to testing1(), testing2(), or testing3() for testing
   }
}
```

An exercise:

Consider this simplified Day class.

Your task: Match ① - ⑥ with the descriptions:

a. ***Error: d1 may not have been initialized***
[This is a compilation problem in Java.
If not fixed, we cannot run the program.]

b. ***Runtime exception: java.lang.NullPointerException***

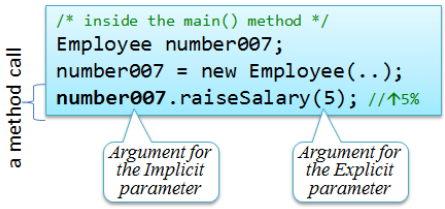
c. ***print: 15 Jan 2014***

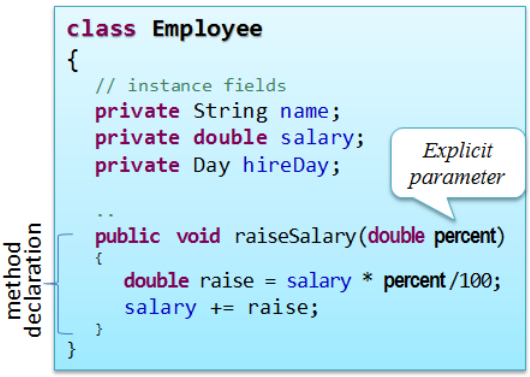
d. ***print: null***

IV. The Implicit Parameter (calling object), and the *this* keyword

- **Implicit and Explicit Parameters**

  Suppose the **raiseSalary** method is called like this:





  We say that the method has 2 parameters:

  - ***Implicit parameter:*** For the object that the method is invoked on. (Here: `number007`) also called "*calling object*"

    "Implicit: not stated directly"

  - ***Explicit parameter:*** The parameter listed in the method declaration. (Here: `percent`)

- **The `this` keyword**

  - In every method, the keyword **`this`** refers to the implicit parameter.

  - E.g. we can rewrite `.raiseSalary`:

    Some programmers prefer this style (clearly distinguishes between instance fields and local variables)

```
class Employee
{
   private String name;
   private double salary;
   private Day hireDay;
   ..
   public void raiseSalary(double percent)
   {
      double raise = this.salary * percent /100;
      this.salary += raise;
   }
}
```

  - When method A invokes method B to handle the implicit parameter (calling object), the implicit parameter is either omitted or specified using the `this` keyword.

```
public class Day
{
   ...
   public boolean isEndOfAMonth()
   {
      ..
   }
   public Day next()
   {
      if (this.isEndOfAMonth())
         ...
      else
         ...
   }
}
```

or

```
public class Day
{
   ...
   public boolean isEndOfAMonth()
   {
      ..
   }
   public Day next()
   {
      if (isEndOfAMonth())
         ...
      else
         ...
   }
}
```

  - We have seen this in the constructor of the Day class.
    More equivalent versions:

```
public Day(int y, int m, int d)
{
   year=y;
   month=m;
   day=d;
}
```

```
public Day(int aYear, int aMonth, int aDay)
{
   year=aYear;
   month=aMonth;
   day=aDay;
}
```

JAVA programmers often use parameter names like these.

```
public Day(int y, int m, int d)
{
   this.year=y;
   this.month=m;
   this.day=d;
}
```

```
public Day(int year, int month, int day)
{
   this.year=year;
   this.month=month;
   this.day=day;
}
```

Note: same spelling

  - We can use this to call another constructor.

    Note: Must be written as the first statement in a constructor

```
public class Day
{
   ...
   public Day(int y, int m, int d) {
      year=y;
      month=m;
      day=d;
   }

   public Day(int y)          {
      this(y,1,1); //first day of the year
   }

   ... more methods
}
```

V. Day – Change of implementation -Using integer data int yyyymmdd;

A new implementation of the `Day` class
- using an integer data as `int yyyymmdd;`

```
public class Day {

   private int yyyymmdd;

   //Constructor
   public Day(int y, int m, int d) {
      yyyymmdd=y*10000+m*100+d;
   }

   // Return a string for the day like dd MMM yyyy
   public String toString() {
      final String[] MonthNames = {
            "Jan", "Feb", "Mar", "Apr",
            "May", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec"};

      return getDay()+" "+ MonthNames[getMonth()-1] + " "+ getYear();
   }
   public int getDay() {return yyyymmdd%100;}
   public int getMonth() {return yyyymmdd%10000/100;}
   public int getYear() {return yyyymmdd/10000;}
   ../Other methods
}
```

Question:
To use this new implementation, do the users need to adjust their code?
Answer:

No.
The way to create and use Day objects (call the public methods) are the same.

VI. Mutator, Accesor Methods getXX, setXX

- **Public methods** are provided for outsiders to act on the data:
  - **Accessor methods (getters)** allow outsiders to obtain the data
    - Often named as `get..`, eg. `getDay()`
    - May not return the values of an instance field literally.
    - E.g., The `Day` class has the instance field `int yyyymmdd;` the asccessor methods are:

```
public int getDay()    {return yyyymmdd%100;}
public int getMonth()  {return yyyymmdd%10000/100;}
public int getYear()   {return yyyymmdd/10000;}
```

  - **Mutator methods (setters)** change the current data of an object
    - Often named as set..., eg. `setDay()`
    - May apply special checking, eg. never make `salary` negative.

- **Note: It is WRONG to think that** "we should add get… and set… for most instance fields!"
  Why? – See the coming explanation later in this topic: *Avoid unnecessary accessor and mutator methods*

VII. public vs private, Encapsulation issue

Public vs Private

- **Public methods**
  Any method in any class can call the method.
  - The followings methods are often public:
    - Constructors
    - Accessor methods
    - Mutator methods
    - Other methods which outsiders might need

- **Public instance fields** 
  For proper practice of encapsulation, we seldom declare instance fields as public.

```
class Employee
{
   // instance fields
   private String name;
   private double salary;
   private Day hireDay;

   // constructor
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = new Day(year,month,day);
   }

   public String getName() {return name;}

   public double getSalary() {return salary;}

   public Day getHireDay() {return hireDay;}

   public void setSalary(double newSalary)
   {
      salary = newSalary;
   }
   public void raiseSalary(double percent)
   {
      double raise = salary * percent /100;
      salary += raise;
   }
}
```

- **Private Instance fields**:
  These instance fields can be accessed only by through methods of the `Employee` class itself.

```
class Employee
{
   // instance fields
   private String name;
   private double salary;
   private Day hireDay;

   ..
}
```

- **Private methods**
  Sometimes implemented, which take the role of *helper methods*.

```
/*Return the number of days in a particular month*/
private static int getMonthTotalDays(int y, int m)
{
   switch(m){
      case 1: case 3: case 5: case 7:
      case 8: case 10: case 12:
            return 31;
      case 4: case 6: case 9: case 11:
            return 30;
      case 2:
            if (isLeapYear(y))
               return 29;
            else
               return 28;
   }
   return 0; //This line never runs. Just for passing the compiler
}
```

- A method can access the private data of all objects of its class, not only the implicit parameter.

```
class X
{
  private int data;
  public X(int d) {data=d*2;}
  public void doSomething(X r)
  {
    X s = new X(8);
    System.out.println(this.data);
    System.out.println(r.data);
    System.out.println(s.data);
  }
}
```

Avoid returning reference from accessor methods

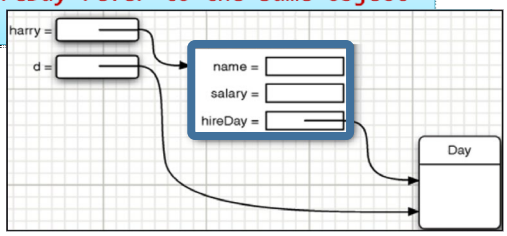
This accessor method breaks encapsulation ☹

* Instance fields should be changed through the methods provided by the class itself.

Now problem happens - The user can write:

```
class Employee {
    private Day hireDay;
    ..
    public Day getHireDay() { return hireDay;}
    ..
}
```

```
Employee harry;
harry = new Employee("harry", 75000, 1987, 12, 15);
Day d = harry.getHireDay(); //d and harry.hireDay refer to the same object
d.advance(); //changes it to 1987-12-16 !!
```



Solution / Rule of thumb:

If you need to return a *multable* data field, you should return a new copy of the Day object. (learn "object cloning" later)

More on mutable , immutable:

mutable: after construction, there are still other way that the object can get changed.
immutable: after construction, there is no way that the object can get changed.

e.g. **Strings are made immutable -**

`String` methods may return new resultant `String` objects,
but `String` methods never amend the content of the original `string` object.

That is, the `String` class does not provide you any method to change the object.
This is known as *immutable*. [http://docs.oracle.com/javase/tutorial/java/data/strings.html]

```
public static void main(String[] args)
{
  String s1, s2;

  s1 = "Everybody gets a good grade";
  s2 = s1.replace("a good grade", "an A+");

  System.out.println(s1); //Output: Everybody gets a good grade
  System.out.println(s2); //Output: Everybody gets an A+
}
```

The **replace()** method of **String**. s1 itself does not change!

Avoid unnecessary accessor and mutator methods

Some beginners think that *"we should add accessor and mutator methods for all instance fields".*

This is BOTH WRONG! ☠

- Getters and setters in this way actually break encapsulation (See next section)
- We should add them only if really needed.
- We can often find replacements:
  Example 1: Inside a Rectangle class we remove `.getX()` and `.getY()`, but add the useful method .draw()
  Example 2: In the Day class, it is bad to provide `setYear(..), setMonth(..), setDay(..)`
  Reason: easily misused, e.g., change 2016-02-29 to 2016-02-**30**
- *"Don't ask for the information you need to do the work; ask the object that has the information to do the work for you. "*
  - http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html

VIII. Benefits of Encapsulation

Recall:

Encapsulation (sometimes called information hiding)

- Simply combining data and behavior in one package, which are considered as implementation details
- Implementation details are hidden away from the users of the objects
- E.g., We can use `Scanner` objects, but their implementation details are encapsulated in the `Scanner` class.

Benefits of Encapsulation

It is a good practice to encapsulate data as **private** instance fields.

1. **Protect the data from corruption by mistake.**
   Outsiders must access the data through the provided public methods
2. **Easier to find the cause of bug**
   Only methods of the class may cause the trouble
3. **Easier to change the implementation**
   - e.g. change from using 3 integers for `year,month,day` to using 1 integer yyyymmdd
   - We only need to change the code in the class for that data type
   - The change is invisible to outsiders, hence not affect users.

IX. The Final keyword - Method constant, Class constant, Final Instance Fields

The `Final` keyword

Can be used to:

① define Method constants (see topic 02)

② define Class constants (see topic 02)

③ declare instance fields as "**Final**" (see below)

Recall [Topic 02]:

① Method constants

```
public class MyApp
{
   public static void main(String[] args)
   {
      final double CM_PER_INCH = 2.54;
      ..
```

② Class constants

```
public class MyApp
{
   public static final double CM_PER_INCH = 2.54;
   public static void main(String[] args)
   {
```

Final Instance Fields

When we declare an instance field as `final`

- It is to be initialized once latest when the object is constructed.
- It cannot be modified again.

③

```
class Employee
{
   private final String name;
   private double salary;
   private final Day hireDay;

   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = new Day(year,month,day);
   }
```

If we remove these statements, we get error: "The blank final field hireDay may not have been initialized"

X. The Static keyword - Class constant (used with final), Static Method, Static Fields

The `Static` keyword

- Used to denote fields and methods that belong to a class (but not to any particular object):

1. **Class constant** (used with `final`)
   - Example 1: `MyApp.CM_PER_INCH`
   - Example 2: `System.out`

2. **Static method** (*class-level method*)
   - Example 1: `Day.isLeapYear(y)`
   - Example 2: `main()`

```
public class MyApp                                  Class constants
{
    public static final double CM_PER_INCH = 2.54;

    public static void main(String[] args)
```

Note: Static methods do not have implicit parameter (ie. no calling object, no `this`)

When to use Static method?

Answer: 1. when we don't need to access the object state because all needed parameters are supplied as explicit parameters (eg. `Day.isLeapYear(y)`)

2. when the method only need to access static fields of the class (eg. `Employee.getNextId()`)

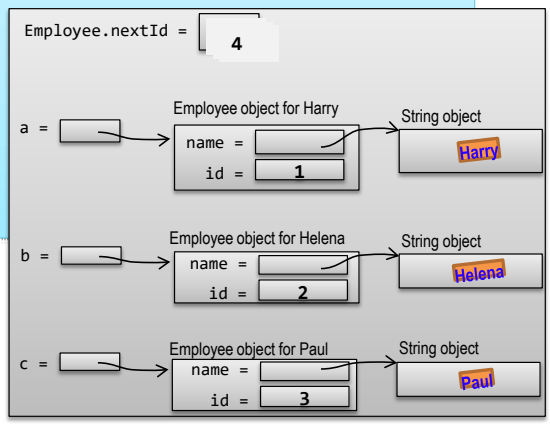
3. **Static Fields**

   A static field is a single field owned by the whole class
   i.e., Not a field per object instance. (taught in next slide)

Static Field Example
`private static int nextId`

```
class Employee
{
   private static int nextId = 1;

   private String name;
   private int id;

   public Employee(String n)      {name = n; id = nextId; nextId++;}
   public static int getNextId()  {return nextId;}
   public String toString()       {return name+" ("+id+") ";}

   public static void main(String[] args)   <- unit test
   {
      Employee a = new Employee("Harry");
      Employee b = new Employee("Helena");
      Employee c = new Employee("Paul");
      System.out.println(a);
      System.out.println(b);
      System.out.println(c);
      System.out.println(
        "The Id of the next new employee will be " +
        Employee.getNextId());
   }
}
```

Note the use of ***nextId*** and update



Output:

```
Harry (1)
Helena (2)
Paul (3)
The Id of the next new employee will be 4
```

☆ If a field is **non-static**, then each object has its own copy of the field. (`name, id`)

☆ If a field is **static**, then there is only one such field per class. (`nextId`)

☆ **Static method** for accessing static fields:
To provide a public method for outsiders to access a static field, we usually make the method static
i.e. class-level method (`getNextId`)

XI. Method Parameters - Java uses "call by value"

Method Parameters

Java uses "call by value":

* Methods receive copy of parameter values
* The original arguments given by the caller do not change
  (Employee a, b in the following example)
* Copy of object reference lets method modify object
  (.salary✓)

```
class Employee
{
   private String name;
   private double salary;

   public Employee(String n, double s)  {name = n; salary=s;}
   public String toString() {return name+" ($"+salary+") ";}

   private static void swapEmployee(Employee e1,Employee e2)
   {
       Employee temp=e1;        //Cannot really swap
       e1=e2;                   // caller's arguments
       e2=temp;
   }

   private static void swapSalary(Employee e1,Employee e2)
   {
       double temp=e1.salary; //OK - swapping the
       e1.salary=e2.salary;   //  instance fields can
       e2.salary=temp;        //  be done.
   }
}
```

See the given example:

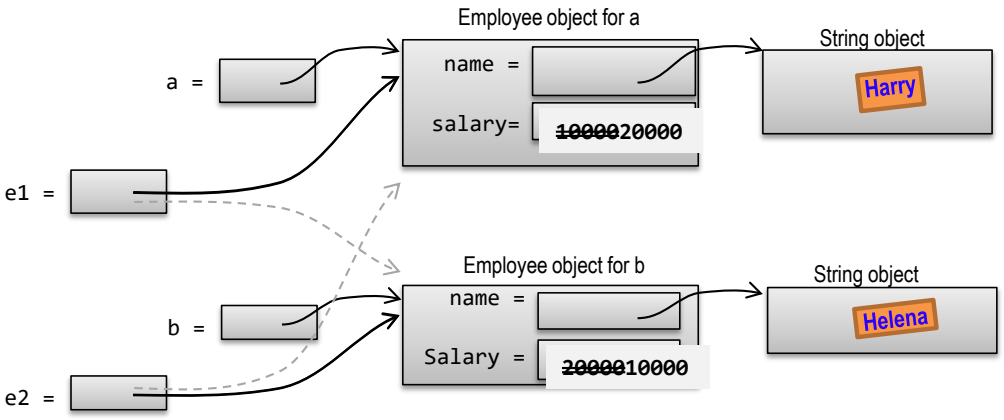
`a` and `b` are object variables, ie. references to objects.

`e1` and `e2` are copies of `a` and `b`.

- Changing `e1, e2` do not affect `a, b`
- But using `e1` and `e2` to refer to the objects and access the salary fields, it does really mean the salaries in the objects pointed by `a` and `b`.

```
public static void main(String[] args)
{
   Employee a = new Employee("Harry",10000);
   Employee b = new Employee("Helena",20000);
   Employee.swapEmployee(a,b); //No change to a and b
   Employee.swapSalary(a,b); //Salary swapped!
   System.out.println(a);
   System.out.println(b);
}
```

Output:
Harry ($20000.0)
Helena ($10000.0)



XII. Default field initialization (not for local variables)

Default field initialization (Note: not for variables!)

- If we don't set a field explicitly in a constructor,
  - * It is set to a default value:
    - Numbers: 0
    - Boolean values: `false`
    - Object references: `null`
- However, local variables are NOT initialized

```
class Employee
{
   private String name;
   private double salary;
   public Employee(String n, double s)  { /* do nothing */}

   public String toString() {return name+" ($"+salary+") ";}

   public static void main(String[] args) // unit test for Employee class
   {
      Employee a = new Employee("Harry",10000);
      System.out.println(a); //show: null ($0.0)  ← no problem!
      String s;
      int x;
      System.out.println(s); // Error: The local variable s may not have been initialized
      System.out.println(x); // Error: The local variable x may not have been initialized
   }
}
```

XIII. Overloading methods, Signature

Overloading methods

- ie. "use the same names for different methods"
- But the parameters must be different
- So that the compiler can determine which to invoke.
  E.g. `Day v1,v2; ...; swap(v1,v2);` //invoke a method called "swap" which accepts 2 Day objects.
- Terminology: Name + parameters = **Signature**

Example 1:

```
void swap(Employee e1, Employee e2) {..}
void swap(Day d1, Day d2) {..} //swap all year,day,month fields
```

Example 2: [Overloaded Constructors]

```
public class Day
{
  ...
 public Day(int y, int m, int d) {year=y;month=m;day=d;}
 public Day(int y) {this(y,1,1);} //first day of the year

 ... more methods
}
```

XIV. Default Constructor - Constructors with zero argument - designed and automatic generated

Default Constructor

- Programmers sometimes create a constructor with no argument, like:

```
public Employee()
{
      name = "";
      salary = 0;
      hireDay = new Day(2014,1,1);
}
```

It is called when we create an object like: `Employee e = new Employee();`

- If we don't provide constructor, then a default constructor is automatically generated.

Like this:

```
public Employee()
{
      name = null;
      salary = 0;
      hireDay = null;
}
```

Recall:
Actually, Fields are already by default null or zero or false. ☺

- If we provide 1 or more constructors, then the above will not be generated for us.

XV. Class Design Hints

Hints for Class Design

1. Always keep data private.
2. Always initialize data
3. Group instance fields as objects when appropriate

```
public class Customer
{
   private String street;
   private String city;
   private String state;
   private String zip;
   ..
}
```

```
public class Customer
{
   private Address address;
   ..
}
```

Create this class

4. Not all fields need individual field accessors and multators
   e.g. once hired, we won't change `hireDay`, therefore `setHireDay(..)` is not needed.
5. Break up classes that have too many responsibilities

```
public class CardDeck // bad design
{
   private int[] value;
   private int[] suit;
   public CardDeck() {...}
   public int getTopValue() {...}
   public int getTopSuit() {...}
}
```

```
public class CardDeck
{
   private Card[] cards;
   public CardDeck() {...}
   public Card getTop() {...
}
```

```
public class Card
{
   private int value;
   private int suit;
   public Card(int v, int s) {...}
   public int getValue() {...}
   public int getSuit() {...}
}
```

6. Proper naming. For classes, use noun, or adjective+noun, or gerund (-ing).
   E.g. `Order`, `RushOrder`, `BillingAddress`

--- end ---