西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Object-Oriented Programming

Chapter 4
Objects and Classes

Dr. Helei Cui

28 Mar 2021

Slides partially adapted from lecture notes by Cay Horstmann

Contents

Background

- 1970s: “Structured” or procedural programming.
  - Algorithms + Data Structures = Programs
  - Procedures operate on shared data.
- 1980s: Object-oriented programming.
  - Each object has data and methods.
  - More appropriate for larger problems.
- Java is thoroughly object-oriented.
  - Everything other than a primitive type value is an object.

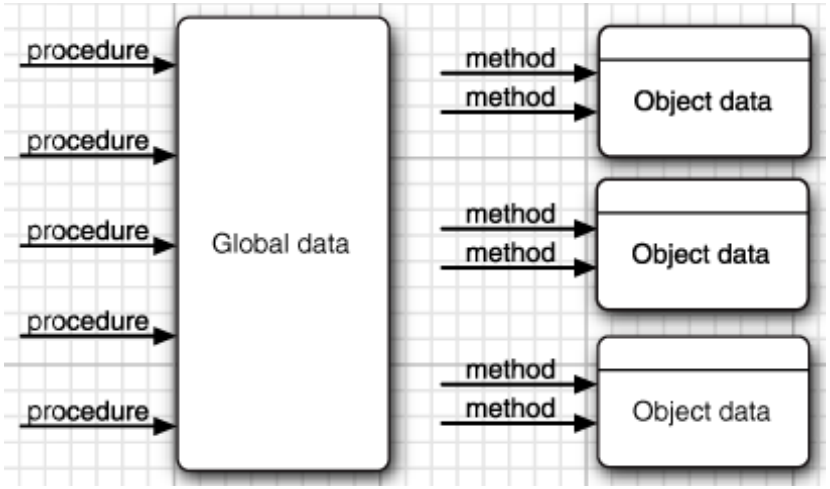


Figure 4.1 Procedural vs. OO programming

Object-oriented vs Procedural

Paradigm	Description	Pros	Cons	Examples
Object-oriented	Treats data fields as *objects* manipulated through predefined methods only	1. Much easier to scale for future needs and development. 2. Good for larger more complex applications. 3. More dynamic and fluid in terms of the architecture and overall design. 4. Maintainable.	1. Can easily become very complicated in terms of design and architecture. 2. Takes much longer to develop initially. 3. More difficult to learn than Procedural.	**Java**, C++, Kotlin, Go, Python, etc.
Procedural	Derived from structured programming, based on the concept of *modular programming* or the *procedure call*	1. Quick to develop and implement. 2. Easy to learn. 3. Simple architecture and overall structure. 4. Good for quick and simple applications.	1. Difficult to scale for future needs. 2. Usually is very flat in terms of design and structure. 3. Not good for larger applications that will likely change over time. 4. Maintaining can be very challenging.	**C**, C++, PHP, Python, etc.

4.1.1 Classes

- **A class is the template from which objects are made.**
  - Describes object data and method behavior.
  - Object = *instance* of class.

https://imagesvc.meredithcorp.io/v3/mm/image?url=https%3A%2F%2Fstatic.onecms.io%2Fwp-content%2Fuploads%2Fsites%2F9%2F2020%2F12%2F03%2Fcookie-cutters-holidays-FT-BLOG1220.jpg

Think of classes as cookie cutters; objects are the cookies themselves.

Encapsulation

- **Encapsulation is simply combining data and behavior in one package and hiding the implementation details from the users of the object.**
  - A.k.a., information hiding.
  - Give an object its "black box" behavior, which is the key to reuse and reliability.

The key to making encapsulation work is to have methods never directly access instance fields in a class other than their own.

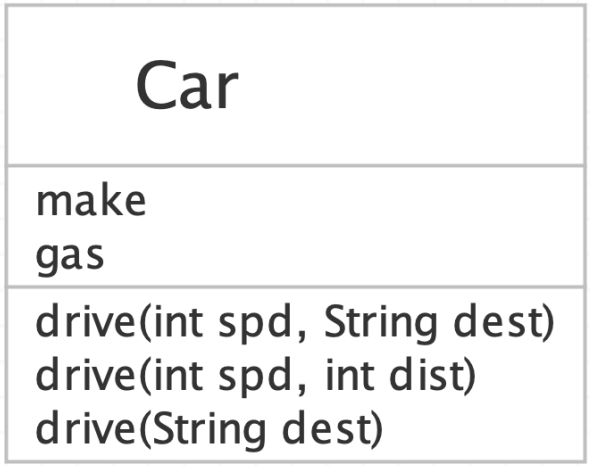
4.1.2 Objects

- **Objects are instances of a class.**
- Three key characteristics:
  - Behavior - *what can you do with this object?*
    - The behavior of an object is defined by the methods that you can call.
  - State - *how does the object react when you invoke those methods?*
    - Each object stores information about what it currently looks like.
    - A change in the state of an object must be a consequence of method calls.
  - Identity - *how is the object distinguished from others that may have the same behavior and state?*
    - Each object has a distinct identity, e.g., two orders that contain the identical items.
    - The individual objects that are instances of a class ALWAYS differ in their identity and USUALLY differ in their state.
- These key characteristics can influence each other.
  - *E.g., if an order is "shipped" or "paid," it may reject a method call that asks it to add or remove items.*

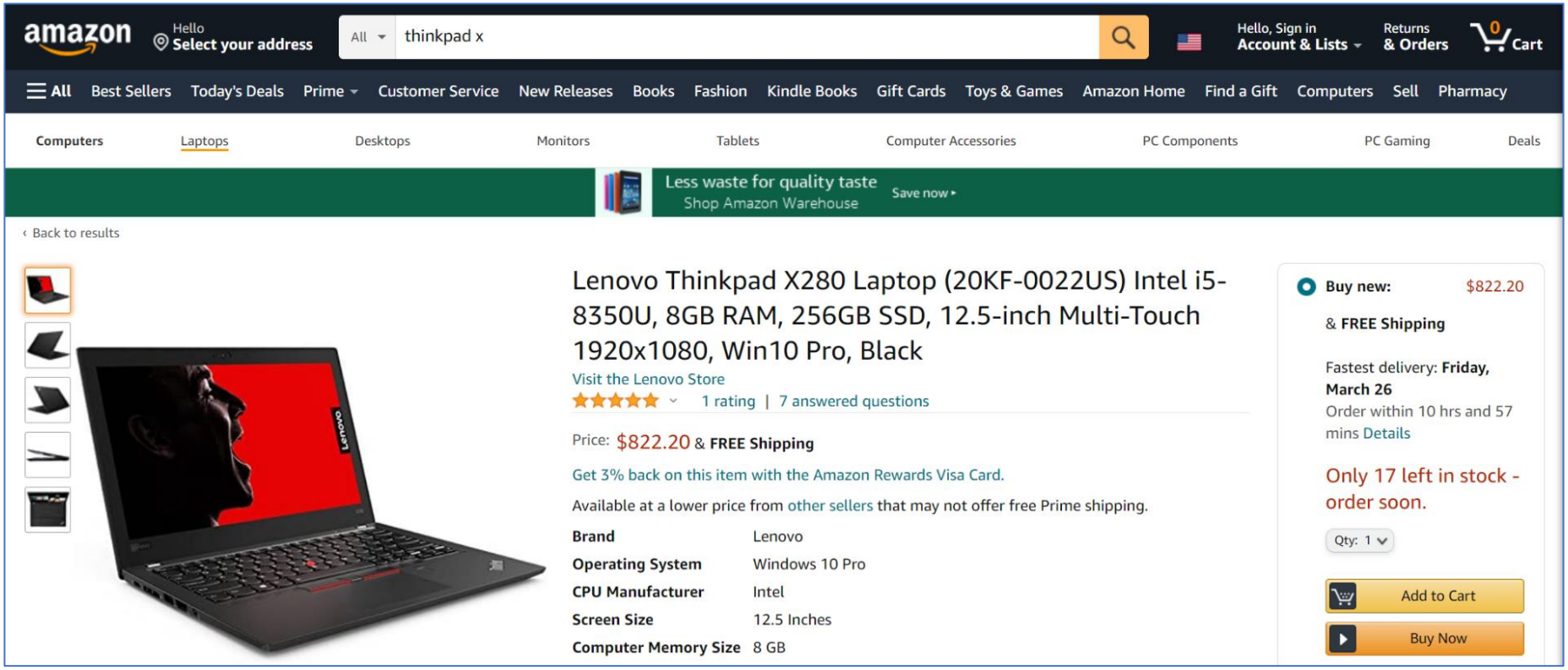
4.1.3 Identifying Classes

- To begin with designing an OO system:
  - **Identify your classes, then add methods to each class.**

- Simple rule:
  - **Nouns ---> classes**
  - **Verbs ---> methods**



4.1.3 Identifying Classes



When building your classes, experience can help you decide which nouns and verbs are the important ones.

Quick question 1

Try to define a student class?

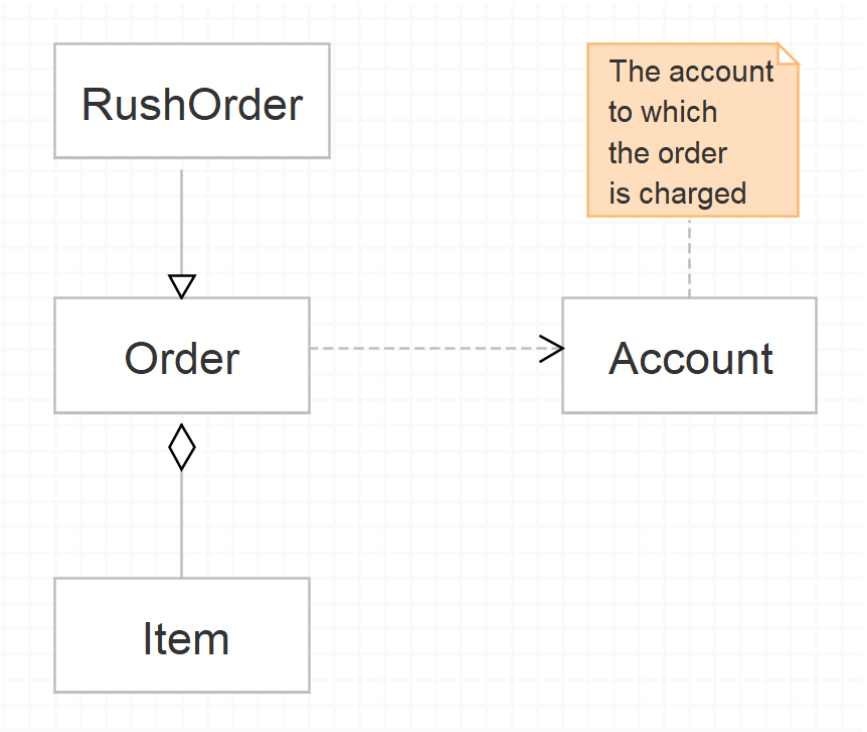
- Class name:
- Attributes:
- Methods:

4.1.4 Relationships between Classes

- Common relationships between classes:
  - Dependence (“uses-a”)
  - Aggregation (“has-a”)
  - Inheritance (“is-a”)

Table 4.1 UML Notation for Class Relationships

Relationship	UML Connector
Inheritance	
Interface implementation	
Dependency	
Aggregation	
Association	
Directed association	



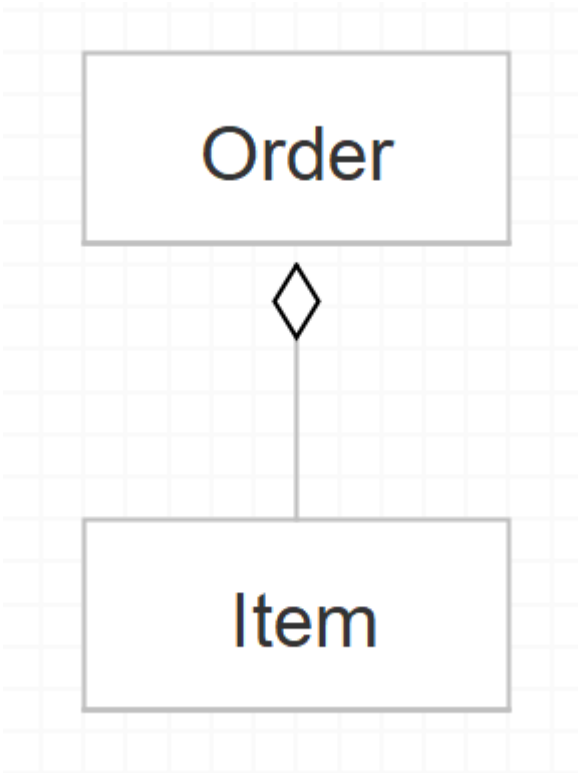
Dependence

- **Dependency depicts how various things within a system are dependent on each other.**
  - Also called “uses-a” relationship.
  - The most obvious and also the most general.
  - E.g., the `Order` class **uses** the `Account` class because `Order` objects need to access `Account` objects to check for credit status.

  - **A class depends on another class if its methods use or manipulate objects of that class.**
- You should try to minimize the number of classes that depend on each other.
  - In software engineering terminology, you want to minimize the coupling between classes.

Aggregation

- **Aggregation is a collection of different things, which describes a part-whole or part-of relationship**
  - Also called “has-a” relationship.
  - Easy to understand as it is concrete.
  - E.g., an `Order` object contains `Item` objects.
  - **Containment means that objects of class A contain objects of class B.**

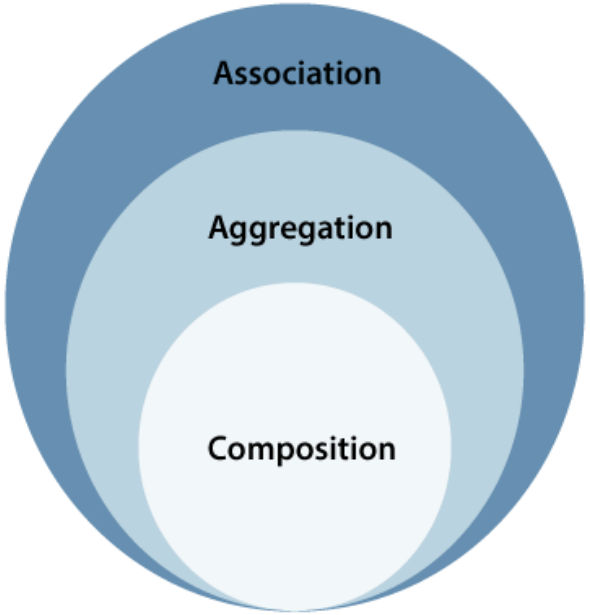


Aggregation vs Association

Association	Aggregation
Association relationship is represented using an arrow.	Aggregation relationship is represented by a straight line with an empty diamond at one end.
In UML, it can exist between two or more classes.	It is a part of the association relationship.
It incorporates one-to-one, one-to-many, many-to-one, and many-to-many association between the classes.	It exhibits a kind of weak relationship.
It can associate one more objects together.	In an aggregation relationship, the associated objects exist independently within the scope of the system.
In this, objects are linked together.	In this, the linked objects are independent of each other.
It may or may not affect the other associated element if one element is deleted.	Deleting one element in the aggregation relationship does not affect other associated elements.
Example: A tutor can associate with multiple students, or one student can associate with multiple teachers.	*Example: A car needs a wheel for its proper functioning, but it may not require the same wheel. It may function with another wheel as well.*

Aggregation vs Association

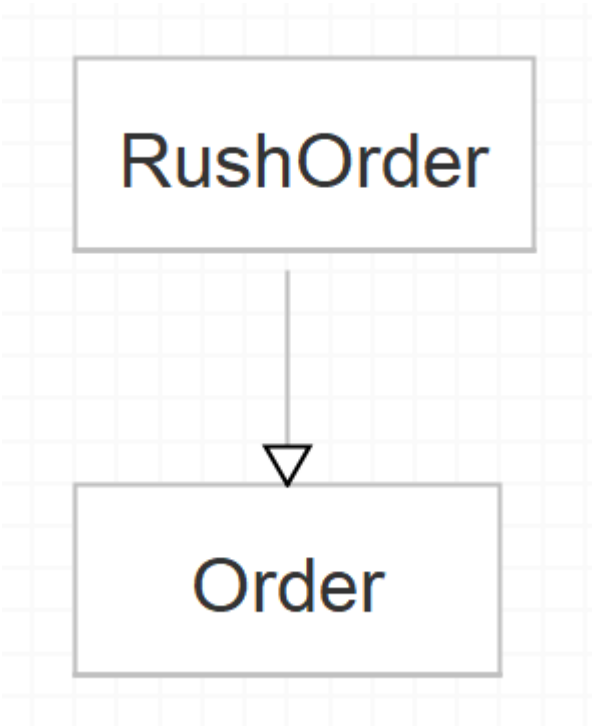
- Some methodologists view the concept of aggregation with disdain and prefer to use a more general “association” relationship.
  - From the point of view of modeling, that is understandable.
- But for programmers, the “has-a” relationship makes a lot of sense.
- We like to use aggregation for another reason as well: The standard notation for associations is less clear.



- For a more detailed comparison, please refer to https://www.javatpoint.com/uml-association-vs-aggregation-vs-composition.

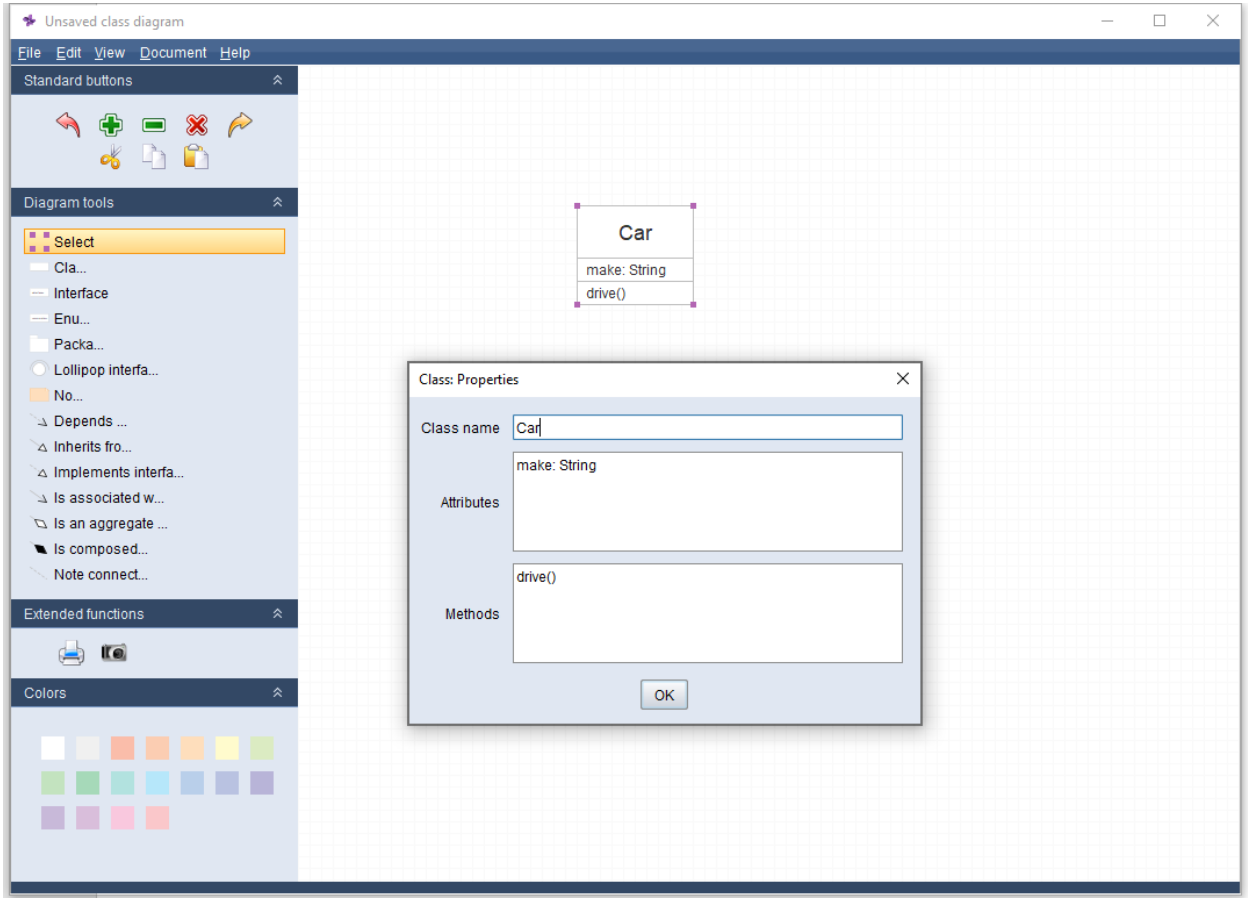
Inheritance

- **Inheritance expresses a relationship between a more special and a more general class.**
  - Also called “is-a” relationship.
  - Expresses a relationship between a more special and a more general class.
  - E.g., a `RushOrder` class inherits from an `Order` class.
  - **In general, if class A extends class B, class A inherits methods from class B but has more capabilities.**



Violet UML Editor

- Search “Violet UML” and try it by yourself.



Contents

Classes we have seen

- Math.sqrt()
- Math.round()
- BigInteger.valueOf()
- BigDecimal.valueOf()
- String.join()
- String.format()
- Arrays.copyOf()
- Arrays.sort()
- Arrays.deepToString()
- System.out.println()
- …

We know how to use them without needing to know how they are implemented. This is encapsulation.

4.2.1 Objects and Object Variables

- To work with objects, you first construct them and specify their initial state. Then you apply methods to the objects.
  - A **constructor** is a special method for constructing and initializing objects.
  - Constructors always have **the same name** as the class name.
- For example, the `Date` class:
  - To construct a `Date` object, combine the constructor with the `new` operator, e.g., "*`new Date();`*".
  - The "new expression" constructs a new object and is initialized to the current date and time.

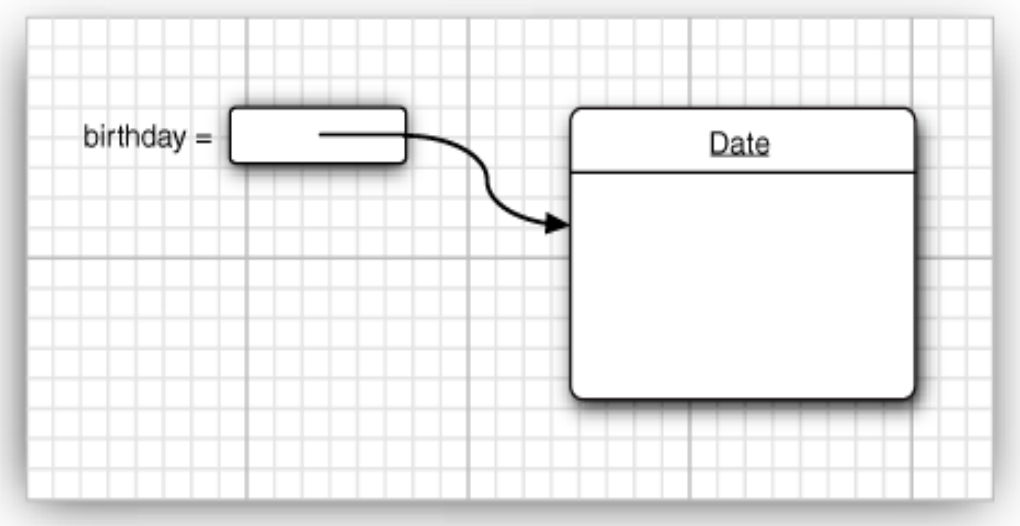
```
System.out.println(new Date());   // pass the object to a method
String s = new Date().toString(); // yield a string of the date
```

In this way, the constructed object can only be used once.

Object Variable

- If you want to keep using a constructed object, you could store the object in a variable.

```
Date birthday = new Date(); // “birthday” is the variable name
```



It shows the object variable `birthday` that refers to the newly constructed object.

```
String s = birthday.toString(); // Now, you can use its methods.
```

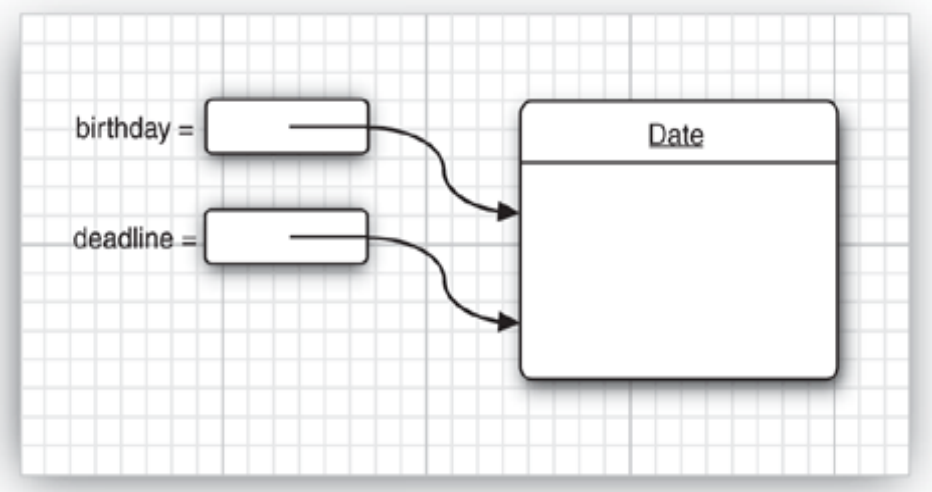
Object vs Object Variable

```
Date deadline; // deadline doesn't refer to any object
String s = deadline.toString(); // not yet initialized
```

- The first line defines a Date object variable, but not yet initialized (i.e., does not refer to an object).
- To initialize, two choices: 1) using new operator; 2) refer to an existing object.

```
deadline = new Date();
deadline = birthday;
```

Now both variables refer to the same object.



Reference

- **An object variable doesn't actually contain an object. It only refers to an object.**
  - In Java, the value of any object variable is a reference to an object that is stored elsewhere.
  - The return value of the `new` operator is also a reference.
  - You can also explicitly set an object variable to `null` to indicate that it currently refers to no object.

```
Date deadline = new Date();
deadline = null;
//. . .
if (deadline != null)
    System.out.println(deadline);
```

4.2.2 The LocalDate Class of the Java Library

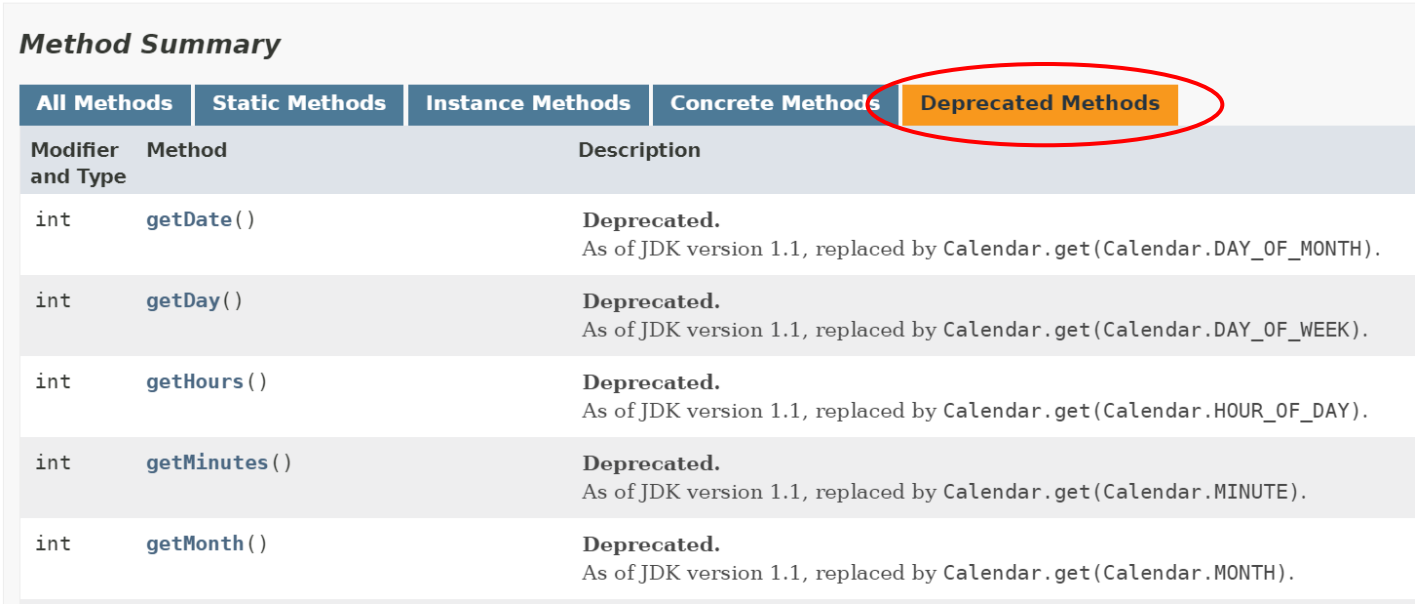
- A Date is a point in time, measured in UTC.
- A LocalDate is a date (day, month, year) in a particular location.
  - Use ***factory methods*** to create instances:

```
LocalDate rightNow = LocalDate.now();
LocalDate newYearEve = LocalDate.of(1999, 12, 31);
```

  - Some useful LocalDate methods:

```
LocalDate aThousandDaysLater = newYearsEve.plusDays(1000);
Year = aThousandDaysLater.getYear();            //2002
Month = aThousandDaysLater.getMonthValue();     //09
Day = aThousandDaysLater.getDayOfMonth();       //26
```

Deprecated Methods



Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method	Description
int	getDate()	**Deprecated.** As of JDK version 1.1, replaced by Calendar.get(Calendar.DAY_OF_MONTH).
int	getDay()	**Deprecated.** As of JDK version 1.1, replaced by Calendar.get(Calendar.DAY_OF_WEEK).
int	getHours()	**Deprecated.** As of JDK version 1.1, replaced by Calendar.get(Calendar.HOUR_OF_DAY).
int	getMinutes()	**Deprecated.** As of JDK version 1.1, replaced by Calendar.get(Calendar.MINUTE).
int	getMonth()	**Deprecated.** As of JDK version 1.1, replaced by Calendar.get(Calendar.MONTH).

- A method is *deprecated* when a library designer realizes that the method should have never been introduced in the first place.
  - The library designers realized that it makes more sense to supply separate classes to deal with calendars.
  - When an earlier set of calendar classes was introduced in Java 1.1, the above `Date` methods were tagged as deprecated.
  - **You can still use them but will get compiler warnings.**
- It is better to stay away from using deprecated methods because they may be removed in a future version of the library.

4.2.3 Mutator and Accessor Methods

- **Mutator** methods will change the state of an object.
- **Accessor** methods access objects without modifying them.

```
GregorianCalendar someDay = new GregorianCalendar(1999, 11, 31);
someDay.add(Calendar.DAY_OF_MONTH, 1000); // Mutator method
year = someDay.get(Calendar.YEAR);        // 2002
month = someDay.get(Calendar.MONTH) + 1;  // 09
day = someDay.get(Calendar.DAY_OF_MONTH); // 26
```

Accessor method

What's the difference between the *GregorianCalendar.add* method and the *LocalDate.plusDays* method?

Practice 1

- Write a Java program to display a calendar for the current month. In addition, use an asterisk (*) to mark the current day.

```
Mon Tue Wed Thu Fri Sat Sun
                        1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19  20  21  22
 23  24  25  26* 27  28  29
 30
```

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/time/LocalDate.html

Contents

4.3.1 An Employee Class

- The simplest form for a class definition in Java:

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

Simplified Version

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

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

    // methods
    public String getName() {
        return name;
    }
    // ... The completed program is shown in Listing 4.2.
}
```

Key Points in Listing 4.2

- Construct an `Employee` array and fill it with three objects:

```
Employee[] staff = new Employee[3];
staff[0] = new Employee("Carl Cracker", . . .);
staff[1] = new Employee("Harry Hacker", . . .);
staff[2] = new Employee("Tony Tester", . . .);
```

- Use the *`raiseSalary`* method to raise each employee's salary by 5%:

```
for (Employee e : staff)
    e.raiseSalary(5);
```

- Print out information about each employee, by calling the accessor ("getter") methods:

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

Key Points in Listing 4.2

- The example program consists of two classes:
  - The `Employee` class;
  - The `EmployeeTest` class with the public access specifier, also contains the ***`main`*** method.
  - The name of the source file is *`EmployeeTest.java`* for matching the name of the public class.
  - You can only have one public class in a source file, but you can have any number of nonpublic classes.
- When you compile this source code, the compiler creates two class files in the directory:
  - *`EmployeeTest.class`* and *`Employee.class`*.
  - Start the program by calling *`java EmployeeTest`*.

```
D:\oop\ch04>dir
 Volume in drive D has no label.
 Volume Serial Number is D30C-135E

 Directory of D:\oop\ch04

03/25/2021  01:33 PM    <DIR>          .
03/25/2021  01:33 PM    <DIR>          ..
03/14/2021  05:47 PM             1,393 EmployeeTest.java
               1 File(s)          1,393 bytes
               2 Dir(s)  716,819,533,824 bytes free

D:\oop\ch04>javac EmployeeTest.java

D:\oop\ch04>dir
 Volume in drive D has no label.
 Volume Serial Number is D30C-135E

 Directory of D:\oop\ch04

03/25/2021  01:34 PM    <DIR>          .
03/25/2021  01:34 PM    <DIR>          ..
03/25/2021  01:34 PM               776 Employee.class
03/25/2021  01:34 PM             1,486 EmployeeTest.class
03/14/2021  05:47 PM             1,393 EmployeeTest.java
               3 File(s)          3,655 bytes
               2 Dir(s)  716,819,521,536 bytes free
```

4.3.2 Use of Multiple Source Files

- Many programmers prefer to put each class into its own source file.
  - *Employee* class ---> *Employee.java*
  - *EmployeeTest* class ---> *EmployeeTest.java*
- You have two choices for compiling the program:
  - You can invoke the Java compiler with a wildcard.

```
javac Employee*.java
```

  - You can simply type

```
javac EmployeeTest.java
```

  - When the Java compiler sees the *Employee* class being used inside *EmployeeTest.java*, it will look for a file named *Employee.class*.

4.3.3 Dissecting the Employee Class

- The keyword ***public*** means that any method in any class can call the method.

```
public Employee(String n, double s, int year, int month, int day)
public String getName()
public double getSalary()
public LocalDate getHireDay()
public void raiseSalary(double byPercent)
```

- The keyword ***private*** ensures that the only methods that can access these instance fields are the methods of the *Employee* class itself.

```
private String name;       // reference to String object
private double salary;
private LocalDate hireDay; // reference to LocalDate object
```

4.3.4 First Steps with Constructors

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

- Constructor runs when you create objects of the `Employee` class:
  - Have the same name as the class.
  - Give the instance fields the initial state.
- Create an instance as follows:

```
new Employee("James Bond", 100000, 1950, 1, 1)
james.Employee("James Bond", 250000, 1950, 1, 1) // ERROR
```

A constructor can only be called in conjunction with the new operator. You can't apply a constructor to an existing object to reset the instance fields.

Keep in Mind

- A constructor has the **same name** as the class.
- A class can have **more than one** constructor.
- A constructor can take **zero, one, or more** parameters.
- A constructor has **no return value**.
- A constructor is always called with the **new** operator.
- ***Do not introduce local variables with the same names as the instance fields.***

```
public Employee(String n, double s, . . .) {
    String name = n;   // ERROR
    double salary = s; // ERROR
    . . .
}
```

4.3.5 Declaring Local Variables with var

- As of Java 10, you can declare local variables with the *var* keyword instead of specifying their type.

```
Employee harry = new Employee("A Hacker", 50000, 1989, 10, 1);
var harry = new Employee("A Hacker", 50000, 1989, 10, 1); // It's OK
```

- This is nice as the type name `Employee` is not required to provide twice.
- But for numeric types, it's better to use their types.
  - It's hard to see the difference between 0 and 0L.

The var keyword can only be used with local variables inside methods. You must always declare the types of parameters and fields.

4.3.6 Working with null References

- Be very careful with *null* values.

```
LocalDate birthday = null;
String s = birthday.toString(); // NullPointerException
```

  - This is a serious error, similar to an "index out of bounds" exception.
    - If your program does not "catch" an exception, it is terminated.
    - Normally, programs don't catch these kinds of exceptions but rely on you not to cause them in the first place.

You should be clear about which fields can be null, e.g., the *name* or *hireDay* field cannot be null.

The “Permissive” Approach

- To turn a null argument into an appropriate non-null value:

```
if (n == null) {
    name = "unknown";
} else {
    name = n;
}
```

- As of Java 9, there is a convenience method:

```
public Employee(String n, double s, int year, int month, int day) {
    name = Objects.requireNonNullElse(n, "unknown");
    . . .
}
```

The "Tough Love" Approach

- To reject a null argument:

```
public Employee(String n, double s, int year, int month, int day) {
    Objects.requireNonNull(n, "The name cannot be null");
    name = n;
    . . .
}
```

  - If someone constructs an `Employee` object with a `null` name, then a `NullPointerException` occurs.
- Two advantages:
  - The exception report has a description of the problem.
  - The exception report pinpoints the location of the problem. Otherwise, a `NullPointerException` would have occurred elsewhere, with no easy way of tracing it back to the faulty constructor argument.

4.3.7 Implicit and Explicit Parameters

- Methods operate on objects and access their instance fields.

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

- Calling *number007.raiseSalary(5)* will execute:

```
double raise = number007.salary * 5 /100;
number007.salary += raise;
```

- The method has two parameters:
  - number007 ---> implicit parameter
  - byPercent ---> explicit parameter

The explicit parameters are explicitly listed in the method declaration, e.g., *double byPercent*. The implicit parameter does not appear in the method declaration.

Keyword this

- The keyword `this` can refer to the implicit parameter in every method.

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

- **This is a better choice as it clearly distinguishes between instance fields and local variables.**

4.3.8 Benefits of Encapsulation

- Note the **private field** and **public method**:

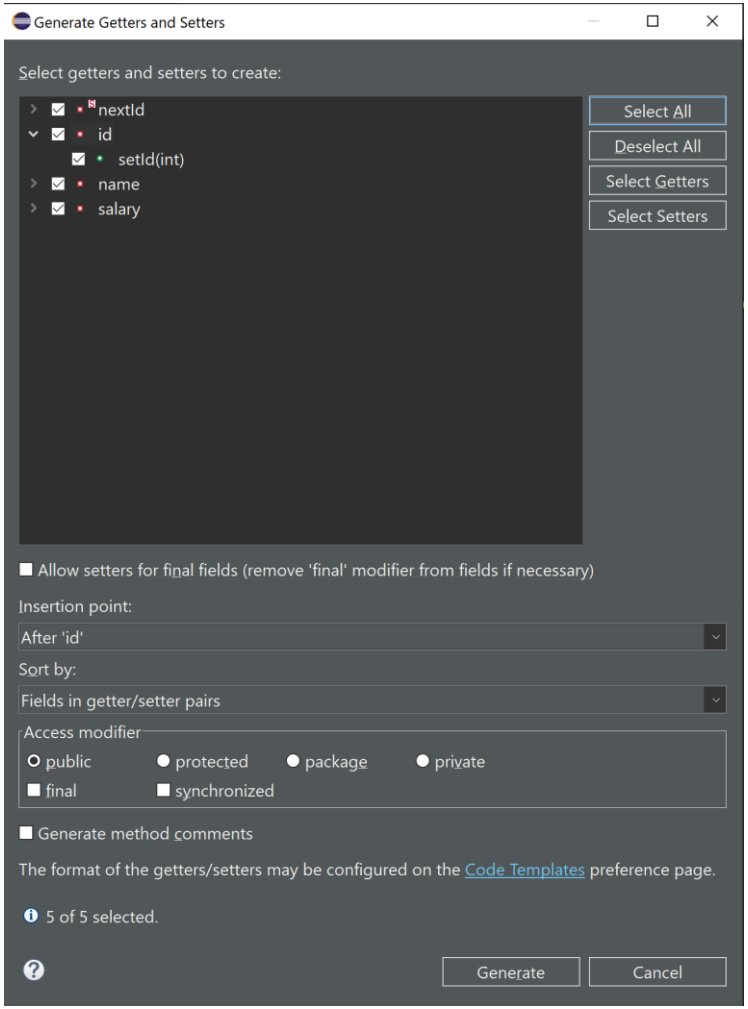
```
private String name;          // instance field
public String getName() {     // accessor method
    return name;
}
```

- Benefit 1: The field is “read-only”.
- Benefit 2: The internal implementation can be changed without affecting any code other than the methods of the class.

```
private String firstName;
private String lastName;
public string getName() {
    return firstName + " " + lastName;
}
```

Three Items

- If you want to get and set the value of an instance field, you need to supply three items:
  - A private data field;
  - A public field accessor method; and
  - A public field mutator method.



4.3.9 Class-Based Access Privileges

- **A method can access the private data of all objects of its class.**

```
class Employee {
    . . .
    public boolean equals(Employee other) {
        return name.equals(other.name);
    }
}
```

- A typical call is

```
if (harry.equals(boss)) . . .
```

- This method accesses the private fields of *harry* and *boss*.
  - A method of the `Employee` class is permitted to access the private fields of any object of type `Employee`.

4.3.10 Private Methods

- While most methods are public, private methods can be useful in some cases.
  - E.g., some helper methods should not be part of the public interface and be best implemented as private.
- **To implement a private method in Java, simply change the `public` keyword to `private`.**
  - If the method is private, the designers of the class can be assured that it is never used elsewhere, so they can simply drop it.
  - If a method is public, you cannot simply drop it because other code might rely on it.

4.3.11 Final Instance Fields

- A field defined as `final` must be initialized when the object is constricted.
  - The field may not be modified again.

```
private final String name;
```

- The `final` modifier is particularly useful for fields whose type is primitive or an immutable class (e.g., `String`).
- For mutable class, the `final` keyword merely means that the object reference stored in the object variable will never again refer to a different object.
  - But the object can be mutated!

```
private final StringBuilder evaluations; // might be confusing
...
evaluations = new Stringbuilder(); // initialized in the constructor
...
evaluations.append("Gold star!\n"); // the object can be mutated
```

Contents

4.4.1 Static Fields

- The `static` fields are associated with the class, rather than with any object.
  - Every instance of the class shares a class variable, which is in one fixed location in memory.

```
class Employee {
     private static int nextId = 1; // nextId is shared among all instances
     private int id;                // every instance has its own id field
     . . .
}
```

- Even if there are no `Employee` objects, the static field `nextId` is present.
  - It belongs to the class, not to any individual object.

4.4.1 Static Fields

- You can use it to assign a unique id for each `Employee` object.

```
public void setId() {
    id = nextId;
    nextId++;
}
```

  - Suppose you set the employee identification number for `harry`:

```
harry.setId(); // harry.id = Employee.nextId; Employee.nextId++;
```

Can you use a static field to count the number of Employee objects?

4.4.2 Static Constants

- A "static+final" field is a class shared constant:

```
public class Math {
    public static final double PI = 3.14159265358979323846;
}
```

  - If the keyword static had been omitted, then PI would have been an instance field of the Math class.

```
public class System {
    public static final PrintStream out = . . .;
} // another static constant in the System class
```

  - Since out has been declared as final, you cannot reassign another print stream to it:

```
System.out = new PrintStream(. . .); // ERROR--out is final
```

4.4.3 Static Methods

- Static methods do not operate on objects.
  - E.g., `Math.pow(a, b)` computes a^b without using a `Math` object.
  - It has no implicit parameter, i.e., no `this`.
- A static method can access a static field:

```
public static int getNextId() {
    return nextId; // returns static field
}
```

- To call this method, you supply the class name:

```
int n = Employee.getNextId();
```

The `main` method is `static` because no objects have been constructed when the program started.

4.4.3 Static Methods

- Use static methods in two situations:
  1. When a method doesn't need to access the object state because all needed parameters are supplied as explicit parameters, *e.g.,* `Math.pow()`.
  2. When a method only needs to access static fields of the class, *e.g.,* `Employee.getNextId()`.

4.4.4 Factory Methods

- Classes such as `LocalDate` and `NumberFormat` use ***static factory*** methods that construct objects.

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // prints $0.10
System.out.println(percentFormatter.format(x));  // prints 10%
```

- *Why doesn't the* `NumberFormat` *class use constructors instead?*
  - **You can't give names to constructors.**
    - The constructor's name is always the same as the class name.
    - But we want two different names to get the currency instance and the percent instance.
  - **When you use a constructor, you can't vary the type of the constructed object.**
    - The factory methods return objects of the class `DecimalFormat`, a subclass that inherits from `NumberFormat`.

4.4.5 The main Method

- The **main** method is a static method.
  - It does not operate on any objects.
    - When a program starts, there aren't any objects yet.
    - The static main method executes and constructs the objects that the program needs.

```
public class Application {
    public static void main(String[] args) {
        // construct objects here
        . . .
    }
}
```

4.4.5 The main Method

```
class Employee {
    public Employee(String n, double s, int year, int month, int day) {
        name = n;
        salary = s;
        hireDay = LocalDate.of(year, month, day);
    }
    public static void main(String[] args) {                    // unit test
        var e = new Employee("Romeo", 50000, 2003, 3, 31);
        e.raiseSalary(10);
        System.out.println(e.getName() + " " + e.getSalary());
    }
}
```

- **Every class can have a main method.** That is a handy trick for unit testing of classes.
  - If you want to test the `Employee` class in isolation, simply execute ***java Employee***.
  - If the `Employee` class is a part of a larger application, you start the application with ***java Application***, and the main method of the `Employee` class is never executed.

Contents

Call by Value & Call by Reference

- **Call by value**: the method gets just the value that the caller provides.
- **Call by reference**: the method gets the location of the variable that the caller provides.
- A method can modify the value stored in a variable passed by reference but not in one passed by value.
- Java always uses ***call by value***.
  - The method gets a copy of all parameter values.
  - In particular, the method cannot modify the contents of any parameter variables passed to it.

```
double percent = 10;
harry.raiseSalary(percent); // the value of percent is still 10
```

An Example

```
public static void tripleValue(double x) {
    x = 3 * x;
}

double percent = 10;
tripleValue(percent); // still doesn't work
```

- The `percent` is not changed:
  1. `x` is initialized with a copy of the value of percent (that is, 10).
  2. `x` is tripled - it is now 30. But `percent` is still 10.
  3. Finally, the parameter variable `x` is no longer in use.

It is impossible for a method to change a primitive type parameter.

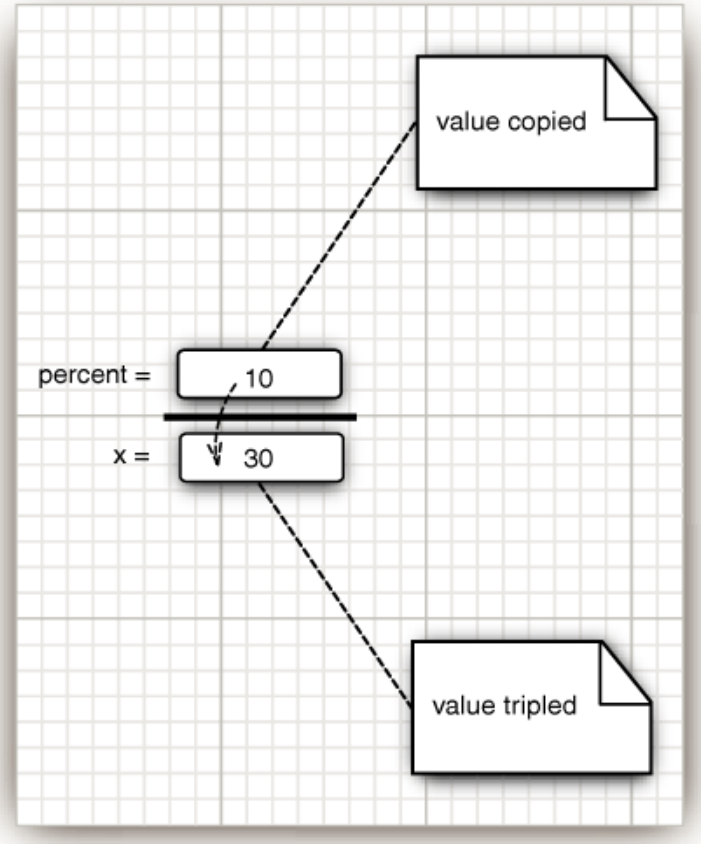


Figure 4.6 Modifying a numeric parameter has no lasting effect

Object References

```
public static void tripleSalary(Employee x) {
    x.raiseSalary(200);
}

harry = new Employee(. . .);
tripleSalary(harry);          // works
```

1. x is initialized with a copy of the value of harry - that is, an object reference.
2. The raiseSalary method is applied to that object reference. The Employee object to which both x and harry refer gets its salary raised by 200 percent.
3. The method ends, and the parameter variable x is no longer in use. Of course, the object variable harry continues to refer to the object whose salary was tripled.

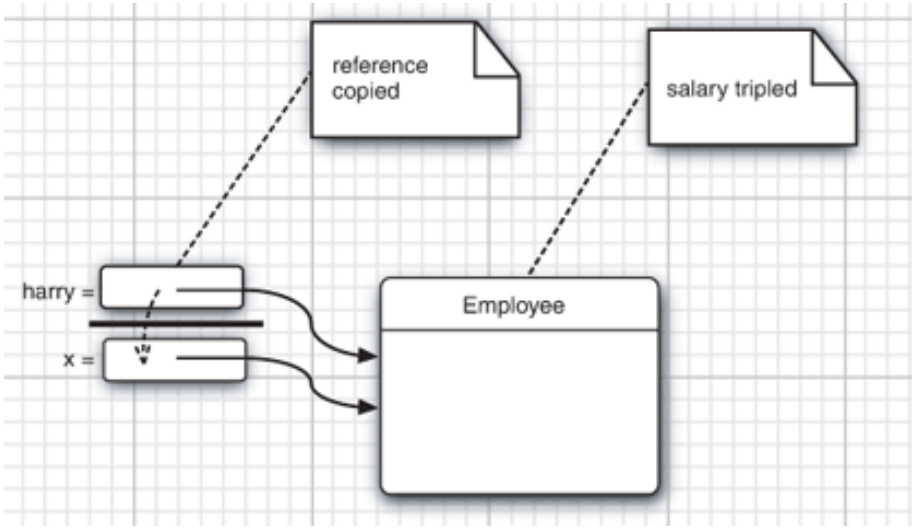


Figure 4.7 Modifying an object parameter has a lasting effect.

Quick question 2

Does Java use call-by-reference for objects?

A. True
B. False

Object References are Passed by Value

- A method tries to swap two Employee objects:

```
public static void swap(Employee x,
Employee y) {
    Employee temp = x;
    x = y;
    y = temp;
}
```

- If Java used call by reference for objects this method would work:

```
var a = new Employee("Alice", . . .);
var b = new Employee("Bob", . . .);
swap(alice, bob);
```

1. The x and y parameters of the swap method are initialized with *copies of these references*.
2. The method then proceeds to swap these copies.
3. When the method ends, x and y are abandoned.

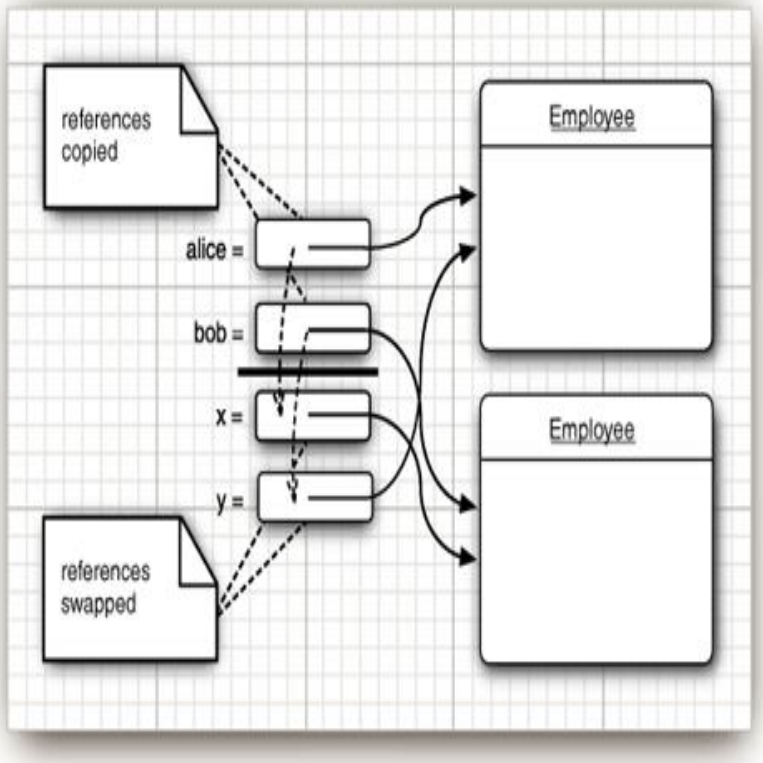


Figure 4.8 Swapping object parameters has no lasting effect.

A Short Summary

- A method cannot modify a parameter of a primitive type (that is, numbers or boolean values).
- A method can change the state of an object parameter.
- A method cannot make an object parameter refer to a new object.

Contents

4.6.1 Overloading

- Some classes have more than one constructors.

```
var messages = new StringBuilder();
var todoList = new StringBuilder("To do:\n"); // with initial string
```

- **Overloading allows different methods to have the same name, but different parameters.**
  - Can differ by the number of input parameters or type of input parameters or both.
  - Overloading is related to compile-time (or static) polymorphism.
  - Java allows you to overload any method - not just constructor methods.

Method Signature

- To completely describe a method, you need to specify its **name together with its parameter types**, called the ***signature* of the method**.
  - E.g., the `String` class has four public methods called `indexOf`. They have signatures:
    - `indexOf(int)`
    - `indexOf(int, int)`
    - `indexOf(String)`
    - `indexOf(String, int)`

The return type is not part of the method signature. That means, you can't have two methods with the same name and parameter type but different return types.

4.6.2 Default Field Initialization

- If you don't set a field explicitly in a constructor, it is automatically set to a default value:
  - numbers ---> 0
  - boolean values---> false
  - object reference ---> null
- It is a poor programming practice to rely on the defaults.
  - E.g., suppose you don't initialize some of the fields in a constructor of the `Employee` class. By default, the `salary` would be initialized with `0` and the `name` and `hireDay` fields would be initialized with `null`.

```
LocalDate h = harry.getHireDay();
int year = h.getYear(); //throws exception if h is null
```

4.6.3 The Constructor with No Arguments

```
public Employee() {
    name = "";
    salary = 0;
    hireDay = LocalDate.now();
}
```

- A constructor with no arguments is allowed.
  - It creates an object with its field set to default.
  - *If you write a class with no constructors whatsoever, then a no-argument constructor is provided for you.*
  - *If a class supplies at least one constructor but does not supply a no-argument constructor, it is illegal to construct objects without supplying arguments.*

You get a free no-argument constructor only when your class has no other constructors.

4.6.4 Explicit Field Initialization

- Regardless of the constructor call, every instance field is better to set to something meaningful.
  - **Assign a value to any field in the class definition;**

```
class Employee() {
    private String name = ""; // carried out before the constructor executes
    ...
}
```

  - **Initialize a field by a method call.**

```
class Employee() {
    private static int nextId;
    private int id = assignId(); // initialized with a method call
    . . .
    private static int assignId() {
        int r = nextId;
        nextId++;
        return r;
    }
    . . .
}
```

4.6.5 Parameter Names

1. Use single-letter parameter names:

```
public Employee(String n, double s) {
    name = n;
    salary = s;
} // You need to read the code to tell what the n and s parameters mean.
```

2. Prefix each parameter with an "a":

```
public Employee(String aName, double aSalary) {
    name = aName;
    salary = aSalary;
}
```

3. Shadow instance fields with the same name:

```
public Employee(String name, double salary) {
    this.name = name;
    this.salary = salary;
}
```

4.6.6 Calling Another Constructor

```
public Employee(double s) {
    this("Employee #" + nextId, s); // calls Employee(String, double)
    nextId++;
}
```

- The first statement of a constructor has the form `this(. . .)`, then the constructor calls another constructor of the same class.
  - *E.g., when you call `new Employee(60000)`, the `Employee(double)` constructor calls the `Employee(String, double)` constructor.*

Using the `this` keyword in this manner is useful - you only need to write common construction code once.

4.6.7 Initialization Blocks

- Three ways to initialize a data field:
  - By setting a value in a constructor;
  - By assigning a value in the declaration;
  - By using ***initialization blocks;***
    - Class declarations can contain arbitrary blocks of code.
    - These blocks are executed whenever an object of that class is constructed.

```
class Employee {
    private static int nextId;
    private int id;
    private String name;
    private double salary;
    // initialization block, runs first before the body of the constructor
    {
        id = nextId;
        nextId++;
    }
    public Employee(String n, double s) {
        name = n;
        salary = s;
    }
}
```

Initialization Blocks is not Common

- Using initialization blocks is never necessary and is not common.
  - **It is usually more straightforward to place the initialization code inside a constructor.**
- It is legal to set fields in initialization blocks even if they are only defined later in the class.
- However, to avoid circular definitions, it is not legal to read from fields that are only initialized later.

What happens when a constructor is called?

1. If the first line of the constructor calls a second constructor, then the second constructor executes with the provided arguments.
2. Otherwise,
   a. All data fields are initialized to their default values (0, false, or null).
   b. All field initializers and initialization blocks are executed, in the order in which they occur in the class declaration.
3. The body of constructor is executed.

Initialize the Static Fields

- To initialize a static field, two choices:
  1. Supply an initial value:

```
private static int nextId = 1;
```

  2. Use a static initialization block:

```
//static initialization block
static
{
    var generator = new Random();
    nextId = generator.nextInt(10000);
}
```

Static initialization occurs when the class is first loaded.

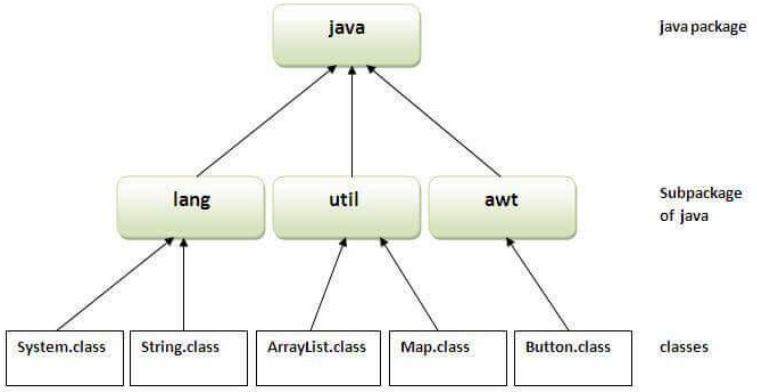
4.6.8 Object Destruction and the `finalize` Method

- Some OOP languages, notably C++, have explicit destructor methods for any cleanup code that may be needed when an object is no longer used.
  - The most common activity in a destructor is reclaiming the memory set aside for objects.
- Java does not support destructors as it does automatic garbage collection.
  - Manual memory reclamation is not needed.
  - Some objects utilize a resource other than memory, such as a file. Remember to supply a `close` method.

Contents

4.7 Packages

- A java **package** is a group of similar types of classes, interfaces and sub-packages.
  - Can be categorized in two form: *built-in package* and *user-defined package*.
  - There are many built-in packages, lang, util, awt, etc.
- Advantages:
  1. Java package is used to categorize the classes and interfaces so that they can be easily maintained.
  2. Java package provides access protection.
  3. Java package removes naming collision.



https://static.javatpoint.com/images/package.JPG

4.7.1 Package Names

- The main reason for using packages is **to guarantee the uniqueness of class names**.
  - Two classes with the same name can be put in different packages, *e.g., java.util.Date ≠ java.sql.Date*.
- To absolutely guarantee a unique package name, use an *Internet domain name written in reverse*.
  - Subpackages can be used for different projects.
  - *E.g., com.horstmann.corejava.Employee.*

From the point of view of the compiler, there is absolutely no relationship between nested packages. For example, the packages java.util and java.util.jar have nothing to do with each other.

4.7.2 Class Importation

- **A class can use all classes from its own package and all public classes from other packages.**
- To access the public classes, you have two methods:
  - Use the fully qualified name:

```
java.time.LocalDate today = java.time.LocalDate.now();
```

  - Use the `import` statement:
    - You can import all classes in a package:

```
import java.time.*;
...
LocalDate today = LocalDate.now(); // no need to provide package prefix
```

    - You can import a specific class inside a package:

```
import java.time.LocalDate;
```

Importing classes explicitly can help readers know exactly which classes you use.

A Potential Error

```
import java.util.*;
import java.sql.*;
. . .
Date today;  // Error - java.util.Date or java.sql.Date?
```

- **The compiler cannot figure out which `Date` class you want as both the packages have a `Date` class.**
- To solve this, simply adding a specific `import` statement:

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

- ***What if you really need both `Date` classes?***
  - *Use the full package name with every class name.*

```
var deadline = new java.util.Date();
var today = new java.sql.Date(. . .);
```

4.7.3 Static Imports

- You can also import **static** methods and fields:

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

  - Now you can refer to `System.out` and `System.exit` without the class name:

```
out.println("Goodbye, World!"); // i.e., System.out
exit(0);                        // i.e., System.exit
```

  - You can import a specific method or field:

```
import static java.lang.System.out;
```

    - Not that clear for `System.out`.
    - Better for mathematical functions:

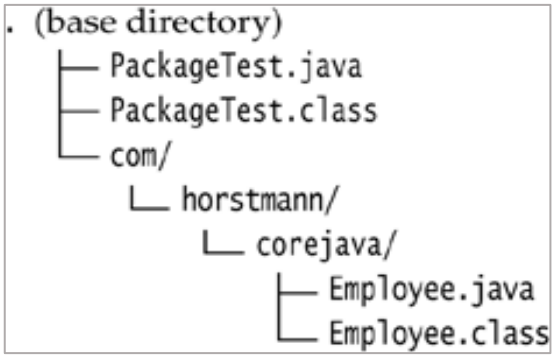
```
import static java.lang.Math.*;
. . .
r = sqrt(pow(x, 2) + pow(y, 2));
```

4.7.4 Addition of a Class into a Package

- To place classes inside a package, put the name of the package at the top of your source file:

```
package com.horstmann.corejava;
public class Employee {
        . . .
}
```

- Place the source file into a subdirectory that matches the package name.
  - *E.g., all source files in the com.horstmann.corejava package should be in a subdirectory.*



4.7.5 Package Access

- Access modifiers:
  - `public` - can be used by any class;
  - `private` - can be used only by the class that defines them;
- If you don't specify either public or private, the feature (that is, the class, method, or variable) can be accessed by all methods in the same package.
  - For classes, this is a reasonable default.
  - For variables, this could be dangerous!

```
public class Window extends Container {
       String warningString;
       . . .
}
```

  - In Java ≤ 1.1, I could add my own class like this:

```
package java.awt;
. . .
Window.warningString = "Trust me!";
```

From Java 1.2, the class loader explicitly disallows loading of user-defined classes whose package name starts with "java.".

4.7.6 The Class Path

- **The class path is the collection of all locations that can contain class files.**
- A JAR file contains multiple class files and subdirectories in a compressed format.
  - ZIP format
- Directories are base directories, containing package directories (such as `com/horstmann/corejava`).
  - Class path elements are separated by `:` (Unix) or `;` (Windows).
  - Can include current directory as `.`
  - Starting with Java 6, you can specify a wildcard for a JAR file directory, e.g., `c:\archives\*`

4.7.7 Setting the Class Path

- Pass to javac or java with *–classpath* option:

```
java -classpath
/home/user/classdir: . :/home/user/archives/archive.jar MyProg
```

```
java -classpath c:\classdir;.;c:\archives\archive.jar MyProg
```

- Or set *CLASSPATH* environment variable:

```
export
CLASSPATH=/home/user/classdir: . :/home/user/archives/archive.jar
```

- With the Windows shell, use

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

It might be a bad idea to set the CLASSPATH environment variable permanently.

Contents

4.8.1 Creating JAR files

- A Java Archive (JAR) file can contain both class files and other file types (e.g., images).
- Use the `jar` tool to make JAR files.
  - In the default JDK installation, it's in the *jdk/bin* directory.
  - **The most common command to make a new JAR file, i.e.,** *jar cvf jarFileName file1 file2 . . .*

```
jar cvf CalculatorClasses.jar *.class icon.gif
```

  - *c - Creates a new or empty archive and adds files to it.*
  - *v - Generates verbose output.*
  - *f - Specifies the JAR file name as the second command-line argument.*
- The jar command has the following format:

```
jar options file1 file2 . . .
```

  - *Please refer to Table 4.2 to see all the options for the jar program.*

4.8.2 The Manifest

- Each JAR file contains a manifest file (MANIFEST.MF) that describes special features of the archive.

```
Manifest-Version: 1.0
lines describing this archive

Name: Woozle.class
lines describing this file
Name: com/mycompany/mypkg/
lines describing this package
```

- To edit the manifest, place the lines that you want to add to the manifest into a text file.

```
jar cfm jarFileName manifestFileName . . .

jar cfm MyArchive.jar manifest.mf com/mycompany/mypkg/*.class
```

- To update the manifest of an existing JAR file, place the additions into a text file.

```
jar ufm MyArchive.jar manifest-additions.mf
```

4.8.3 Executable JAR Files

- Use the ***e*** option of the jar command to specify the entry point of your program.

```
jar cvfe MyProgram.jar com.mycompany.mypkg.MainAppClass files to add
```

- Alternatively, specify the main class of your program in the manifest, by adding the following statement:

```
Main-Class: com.mycompany.mypkg.MainAppClass
```

- Users can simply start the program as:

```
java -jar MyProgram.jar
```

  - On Windows, the Java runtime installer creates a file association for the *".jar"* extension that launches the file with the *javaw –jar* command.
  - On Mac OS X, the operating system recognizes the *".jar"* file extension and executes the Java program when you double-click a JAR file.

4.8.4 Multi-Release JAR Files

- Java 9 introduces multi-release JARs that can contain class files for different Java releases.
- To add versioned class files, use the *--release* flag:

```
jar uf MyProgram.jar --release 9 Application.class
```

- To build a multi-release JAR file from scratch, use the *-C* option and switch to a different class file directory for each version:

```
jar cf MyProgram.jar -C bin/8 . --release 9 –C bin/9Application.class
```

- When compiling for different releases, use the *--release* flag and the *-d* flag to specify the output directory:

```
javac -d bin/8 --release 8 . . .
```

- The main purpose of multi-release JARs is to enable a particular version of your program or library to work with multiple JDK releases.

4.8.5 A Note about Command-Line Options

- Starting with Java 9, multiletter option names are preceded by double dashes, with single-letter shortcuts for common options.

```
ls --human-readable
```

```
ls -h
```

- Single-letter options without arguments can be grouped together:

```
jar -cvf MyProgram.jar -e mypackage.MyProgram */*.class
```

- If you want to be thoroughly modern, you can safely use the long options of the `jar` command:

```
jar --create --verbose --file jarFileName file1 file2 . . .
```

- Single-letter options also work if you don't group them:

```
jar -c -v -f jarFileName file1 file2 . . .
```

Contents

4.9.1 Comment Insertion

- The `Javadoc` extracts information for the following items:
  - Modules
  - Packages
  - Public classes and interfaces
  - Public and protected fields
  - Public and protected constructors and methods
- Each comment is placed immediately above the features it describes.
  - A comment starts with a `/**` and ends with a `*/`.
- Each `/** . . . */` documentation comment contains *free-form text* followed by tags.
  - A tag starts with an `@`, such as `@since` or `@param`.
  - In the free-form text, you use HTML modifiers such as `<strong>. . . </strong>` for strong emphasis, `<img . . ./>` to include an image, etc.
  - If you want to type code without worrying about escaping < characters inside the code, try to use `{@code . . . }`.

4.9.2 Class Comments

- The class comment must be placed ***after*** **any import statements, directly *before* the class definition.**
  - Here is an example of a class comment:

```
/**
 * A {@code Card} object represents a playing card, such as "Queen of
 * Hearts". A card has a suit (Diamond, Heart, Spade or Club) and a
 * value (1 = Ace, 2 . . . 10, 11 = Jack, 12 = Queen, 13 = King)
 */
public class Card {
   . . .
}
```

  - There is no need to add an * in front of every line. But your IDE may supply the asterisks automatically.

```
/**
   A {@code Card} object represents a playing card, such as "Queen of
   Hearts". A card has a suit (Diamond, Heart, Spade or Club) and a
   value (1 = Ace, 2 . . . 10, 11 = Jack, 12 = Queen, 13 = King)
*/
```

4.9.3 Method Comments

- Each method comment must immediately precede the method that it describes:
  - @param *variable description*
  - @return *description*
  - @throws *class description*

```
/**
 * Raises the salary of an employee.
 * @param byPercent the percentage by which to raise the salary
 * @return the amount of the raise
 */
public double raiseSalary(double byPercent) {
  double raise = salary * byPercent / 100;
  salary += raise;
  return raise;
}
```

4.9.4 Field Comments

- You only need to document `public` fields.
  - Generally, it means static constants.

```
/**
 * The "Hearts" card suit
 */
public static final int HEARTS = 1;
```

4.9.5 General Comments

- **@since *text***
  - the *text* can be any description of the version that introduced this feature, *e.g., @since 1.7.1*.
- **@author *name***
  - this makes an author entry. You can have multiple @author tags, one for each author.
- **@version *text***
  - the *text* can be any description of the current version.
- **@see *reference***
  - adds a hyperlink in the "see also" section. It can be used with both classes and methods. The *reference* can be one of the following:
    - *package.class#feature label*
    - *label*
    - *"text"*

4.9.6 Package Comments

- To generate package comments, you need to add a separate file in each package directory. You can have two choices:
  1. Supply a Java file named **package-info.java**. The file must contain an initial Javadoc comment, delimited with /** and */, followed by a **package** statement. It should contain no further code or comments.
  2. Supply an HTML file named **package.html**. All text between the tags **<body>. . .</body>** is extracted.

4.9.7 Comment Extraction

1. Change to the directory that contains the source files you want to document.
   1. If you have nested packages to document, such as **com.horstmann.corejava**, you must be working in the directory that contains the subdirectory **com**. (This is the directory that contains the **overview.html** file, if you supplied one.)
2. Run the command `javadoc -d docDirectory nameOfPackage` for a single package or run `javadoc -d docDirectory nameOfPackage1 nameOfPackage2 ...` to document multiple packages.
   1. If your files are in the unnamed package, run instead `javadoc -d docDirectory *.java`

Contents

Class Design Hints - 1

- **Always keep data private.**
  - This is first and foremost; doing anything else violates encapsulation.
  - You may need to write an accessor or mutator method occasionally, but you are still better off keeping the instance fields private.
  - Bitter experience shows that the data representation may change, but how this data are used will change much less frequently.

When data are kept private, changes in their representation will not affect the users of the class, and bugs are easier to detect.

Class Design Hints - 2

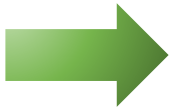
- **Always initialize data.**
  - Java won't initialize local variables for you, but it will initialize instance fields of objects.
  - Don't rely on the defaults, but initialize all variables explicitly, either by supplying a default or by setting defaults in all constructors.

```
private int id;
private String name = ""; // instance field initialization
private double salary;

// static initialization block
static
{
   var generator = new Random();
   // set nextId to a random number between 0 and 9999
   nextId = generator.nextInt(10000);
}

// object initialization block
{
   id = nextId;
   nextId++;
}

// three overloaded constructors
public Employee(String n, double s)
{
   name = n;
   salary = s;
}
```

Class Design Hints - 3

- **Don't use too many basic types in a class.**
  - The idea is to replace multiple *related* uses of basic types with other classes.
    - Keep your classes easier to understand and to change.
    - For example, replace the following instance fields in a `Customer` class with a new class called `Address`. This way, you can easily cope with changes to addresses, such as the need to deal with international addresses.

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



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

Class Design Hints - 4

- **Not all fields need individual field accessors and mutators.**
  - An employee's salary - both "get" and "set" are needed
  - The hiring date - only "get" is needed

Objects have instance fields that you don't want others to get or set, such as an array of state abbreviations in an Address class.

Class Design Hints - 5

- **Break up classes that have too many responsibilities.**
  - E.g., the CardDeck class is a bad design, which can be separated into two new classes.

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

Better to introduce a Card class that represents an individual card.

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

}
```

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

Class Design Hints - 6

- **Make the names of your classes and methods reflect their responsibilities.**
  - A good convention is that a class name should be:
    - a noun, *e.g., Order*
    - a noun preceded by an adjective, *e.g., RushOrder*
    - a gerund (an "-ing" word), *e.g., BillingAddress*
  - As for methods, follow the standard convention:
    - accessor methods begin with a lowercase get, *e.g., getSalary*
    - mutator methods use a lowercase set, *e.g., setSalary*

Class Design Hints - 7

- **Prefer immutable classes.**
  - The `LocalDate` class, and other classes from the `java.time` package, are immutable - no method can modify the state of an object. Instead of mutating objects, methods such as `plusDays` return new objects with the modified state.
    - **When classes are immutable, it is safe to share their objects among multiple threads.**
    - Better for classes that represent values, such as a string or a point in time.
  - **Not all classes should be immutable.**
    - It would be strange to have the `raiseSalary` method return a new `Employee` object when an employee gets a raise.

Recap

- 4.1 Introduction to Object-Oriented Programming
- 4.2 Using Predefined Classes
- 4.3 Defining Your Own Classes
- 4.4 Static Fields and Methods
- 4.5 Method Parameters
- 4.6 Object Construction
- 4.7 Packages
- 4.8 JAR Files
- 4.9 Documentation Comments
- 4.10 Class Design Hints