西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Object-Oriented Programming

Chapter 4
Objects and Classes

Dr. Helei Cui

28 Mar 2020

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.

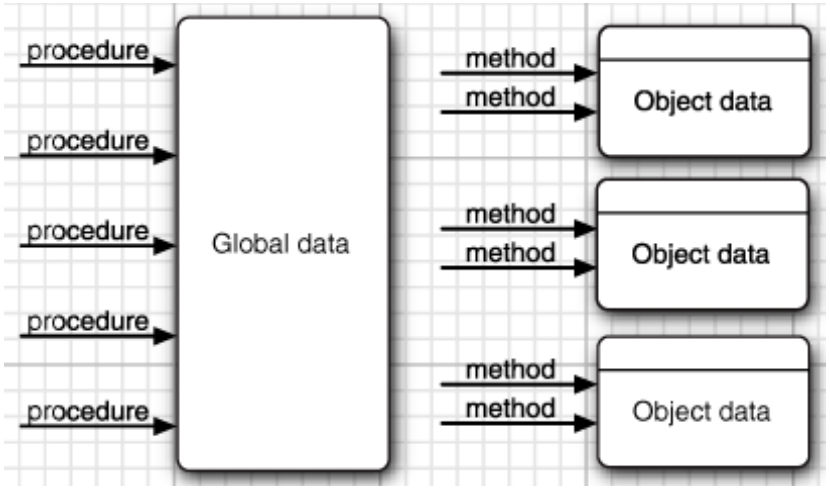


Figure 4.1 Procedural vs. OO programming

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.

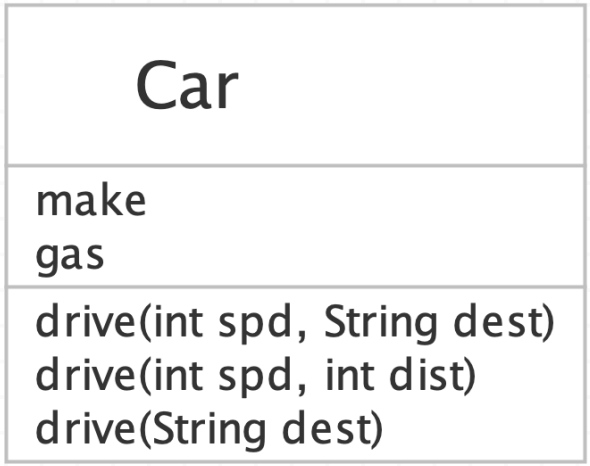
4.1.2 Objects

- **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.*

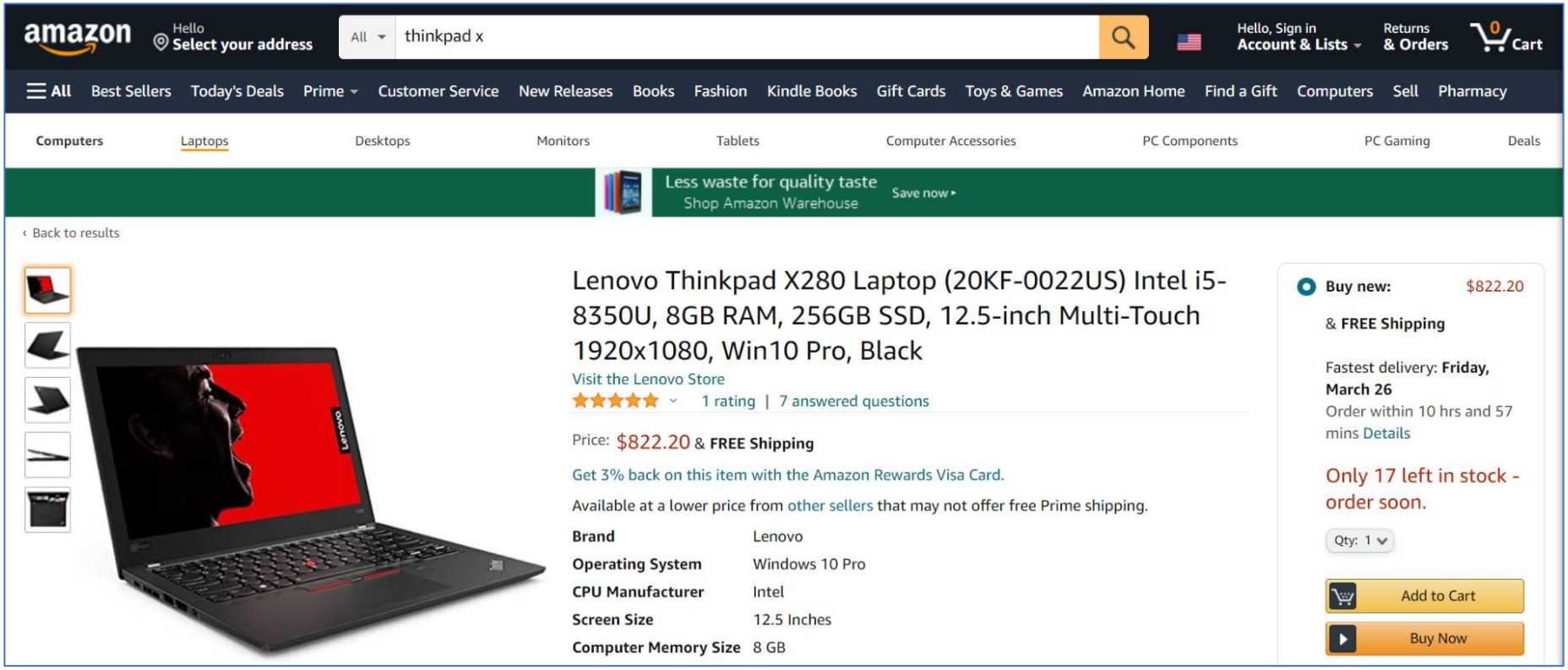
4.1.3 Identifying Classes

- 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?

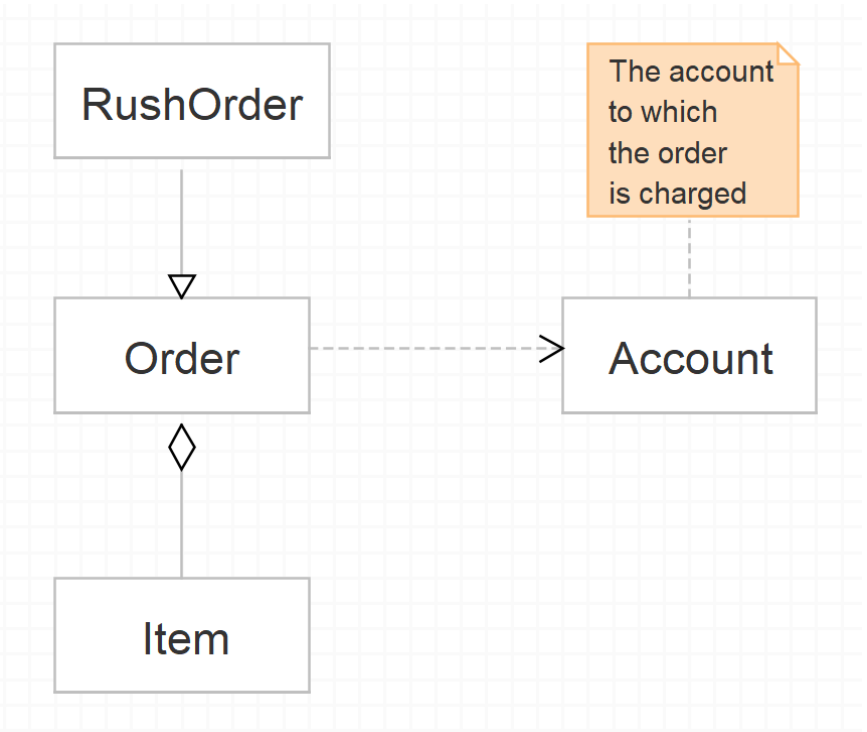
- Class name:
- Attributes:
- Methods:

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	



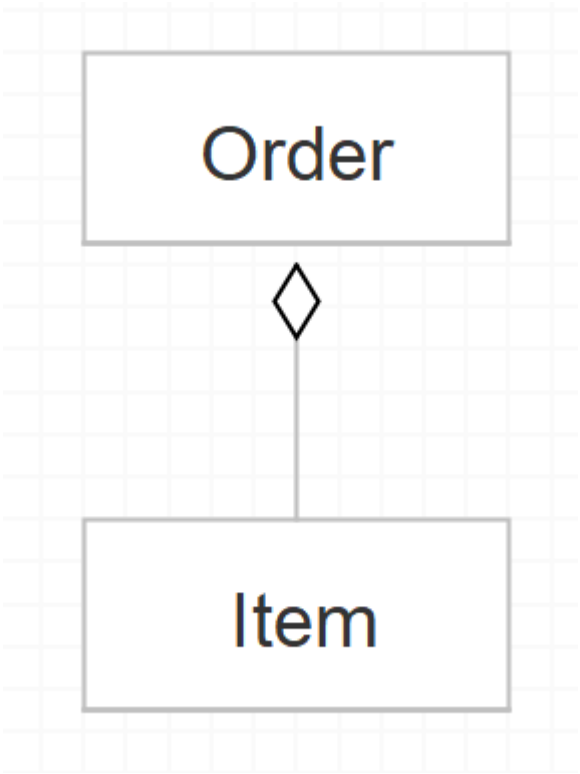
Dependence

- **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.**

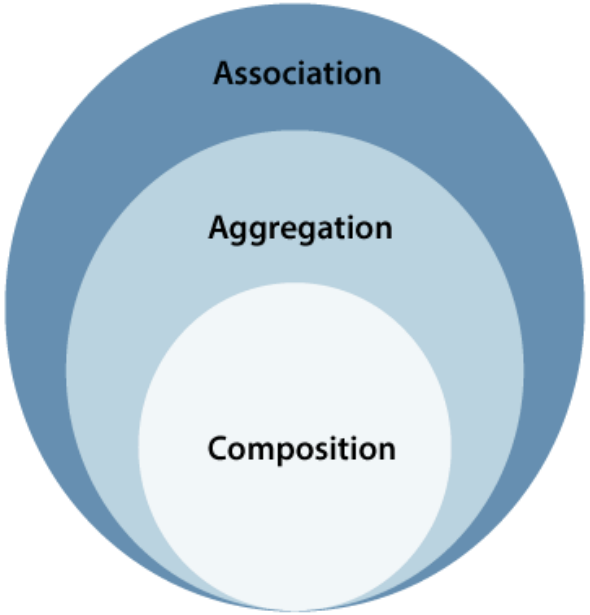


Aggregation vs Association

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

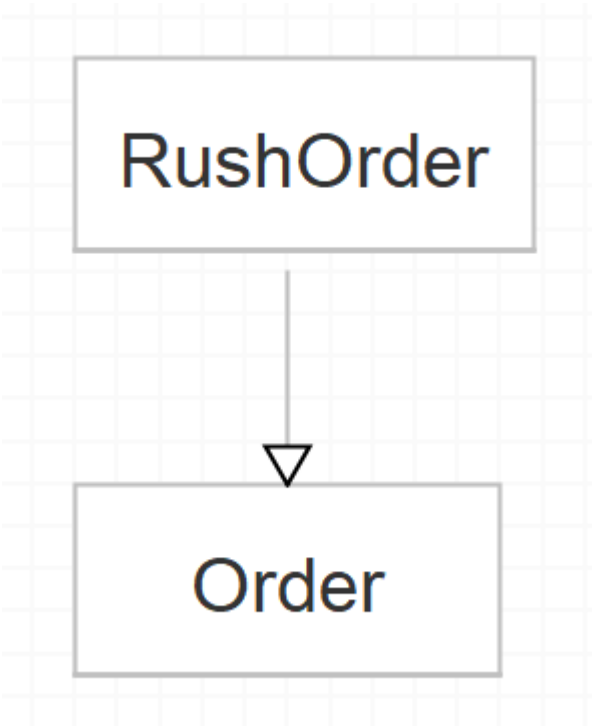
- Some methodologists view the concept of aggregation with disdain and prefer to use a more general “association” relationship.
  - From the point of view of modeling, that is understandable.
- But for programmers, the “has-a” relationship makes a lot of sense.
- We like to use aggregation for another 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.

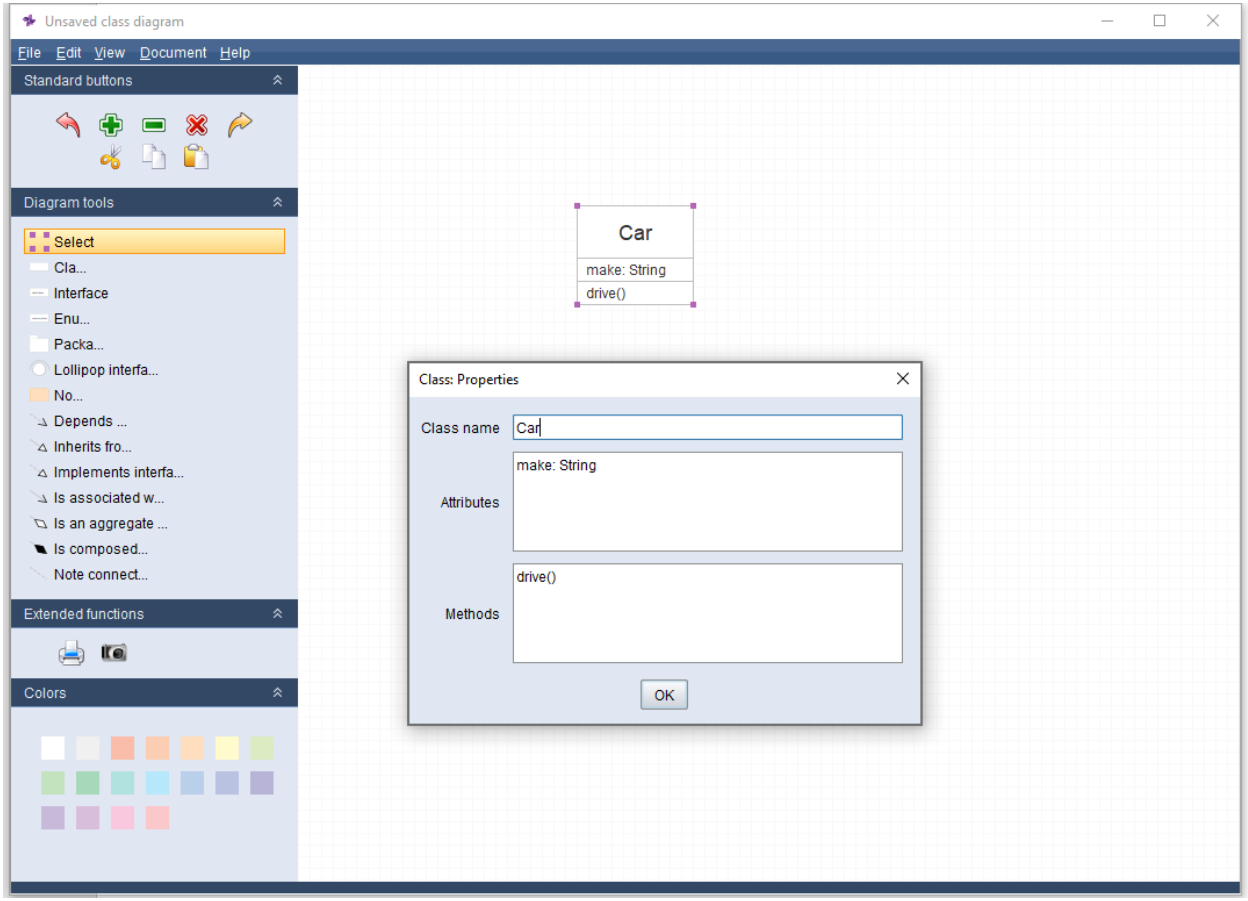
Inheritance

- **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.

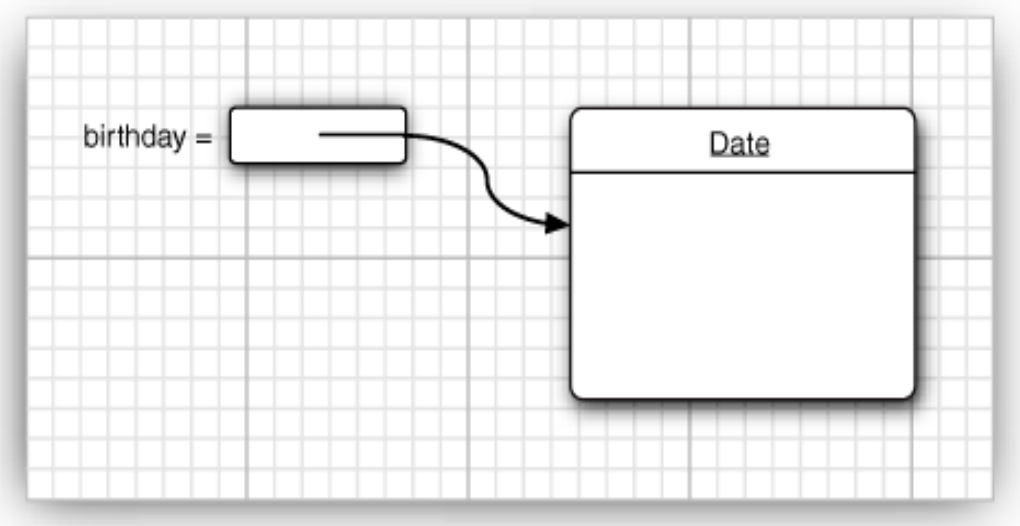
```
System.out.println(new Date());   // pass the object to a method
String s = new Date().toString(); // yield a string of the date
```

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.
```

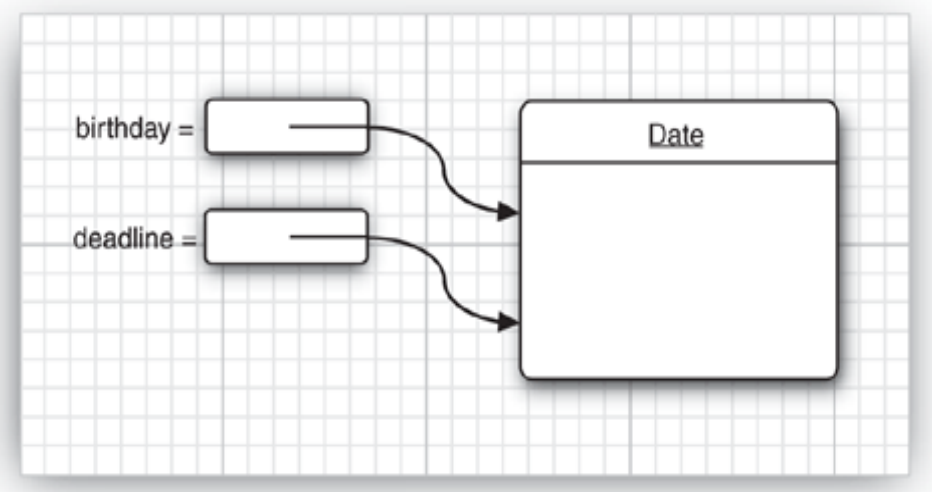
Object vs Object Variable

```
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

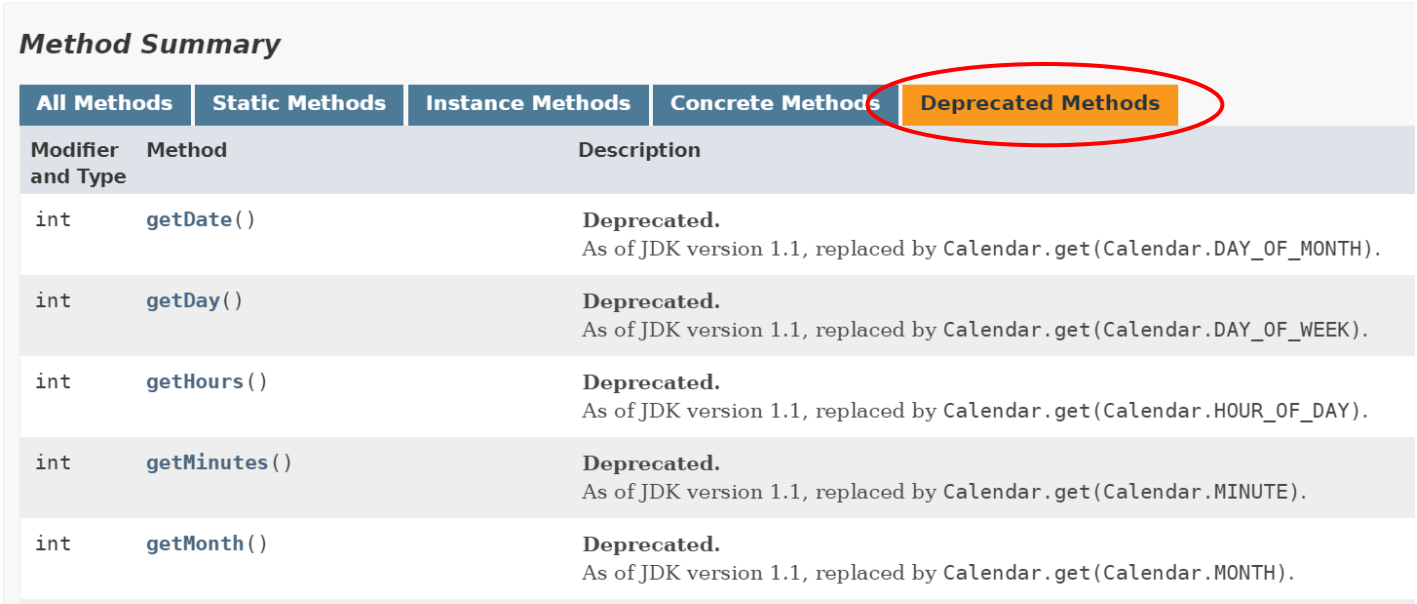
- A Date is a point in time, measured in UTC.
- A LocalDate is a date (day, month, year) in a particular location.
  - Use ***factory methods*** to create instances:

```
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**:

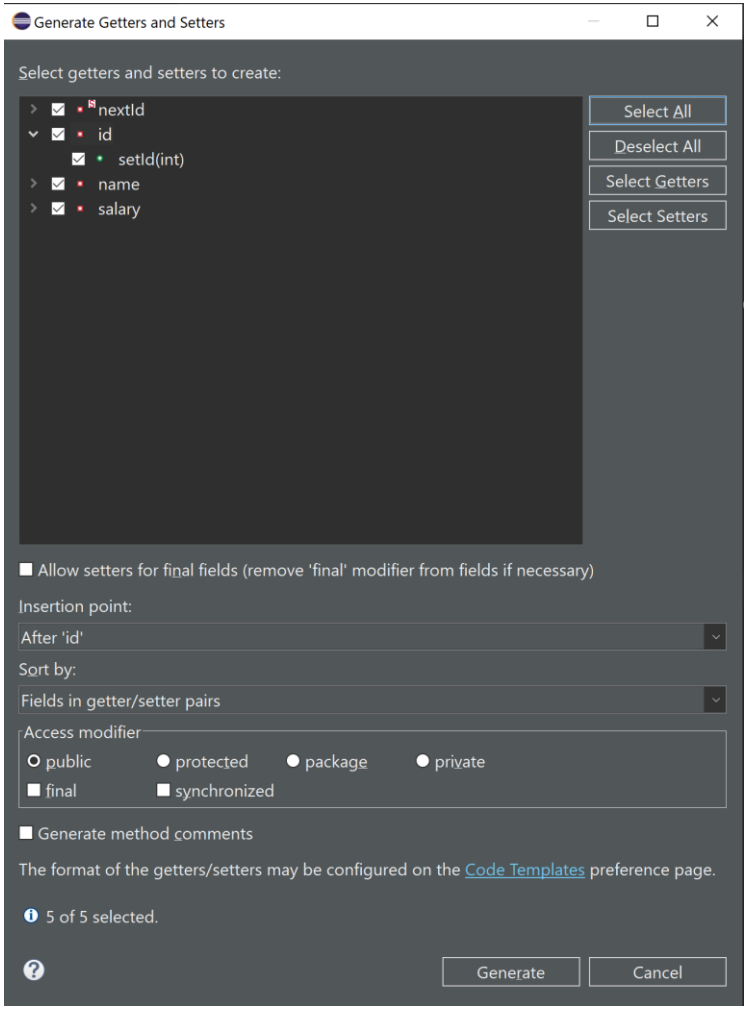
```
private String name;          // instance field
public String getName() {     // accessor method
    return name;
}
```

- 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()`.