CSCB07 - Software Design

Object-Oriented Programming (1)

Summer 2021

Object-Oriented Thinking

- Procedural paradigm
  - Focuses on designing methods
  - Data and operations on the data are separate
- Object-oriented paradigm
  - Couples methods and data together into objects
  - Organizes programs in a way that mirrors the real world
  - A program can be viewed as a collection of cooperating objects
  - Makes programs easier to develop and maintain
  - Improves software reusability

Inheritance

- Powerful feature for reusing software
- Helps avoid redundancy
- Different objects might have common properties and behaviors
  - E.g. Person, Employee
- Inheritance allows developers to
  - Define a general class (or superclass). E.g. Person
  - Extend the general class to a specialized class (or subclass). E.g. Employee
- In Java, the keyword ***extends*** is used to indicate inheritance

Casting objects and the *instanceof* operator

- It is always possible to cast an instance of a subclass to a variable of a superclass (known as upcasting)
  - E.g. **Person p = new Employee();**
- When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used
  - E.g. **Person p = new Employee(); Employee e = (Employee)p;**
  - If the superclass object is not an instance of the subclass, a runtime error occurs
  - It is a good practice to ensure that the object is an instance of another object before attempting a casting. This can be accomplished by using the ***instanceof*** operator
- Casting an object reference does not create a new object

Overloading and Overriding

- Overloading
  - Defining methods having the same name but different signatures
    - Signature: method name + types of its formal parameters
  - Overloading methods can make programs clearer and more readable
- Overriding
  - Defining a method in the subclass using the same signature and the same return type as in its superclass
  - The ***@Override*** annotation helps avoid mistakes
  - A static method cannot be overridden (it can be invoked using the syntax SuperClassName.staticMethodName)

The *super* keyword

- Refers to the superclass
- Can be used to invoke a superclass constructor
  - Syntax: ***super***() or ***super***(parameters)
  - Must be the first statement of the subclass constructor
  - A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts super() as the first statement in the constructor
  - If a class is designed to be extended, it is better to provide a no-argument constructor to avoid programming errors
- Can be used to invoke a superclass method
  - Syntax: ***super*.methodName**(parameters)
  - Useful in the case of overridden methods

The *Object* class

- Every Java class has ***Object*** as superclass
- It has methods that are usually overridden
  - ***equals***
  - ***hashCode***
  - ***toString***

The *Object* class: *equals* method

- Header: ***boolean equals(Object obj)***
- The implementation provided by the ***Object*** class checks whether two reference variables point to the same object
  - Does not check "logical equality"
- When you override the ***equals*** method, you must adhere to its general contract:
  - *Reflexive*: For any non-null reference value x, x.equals(x) must return true
  - *Symmetric*: For any non-null reference values x and y, x.equals(y) must return true if and only if y.equals(x) returns true
  - *Transitive*: For any non-null reference values x, y, z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true
  - *Consistent*: For any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false
  - For any non-null reference value x, x.equals(null) must return false

The *Object* class: *hashCode* method

- Header: ***int hashCode()***
- The implementation provided by the ***Object*** class returns the memory address of the object
- The ***hashCode*** method should be overridden in every class that overrides ***equals***
  - Equal objects must have equal hash codes
- A good ***hashCode*** method tends to produce unequal hash codes for unequal objects

The *Object* class: *toString* method

- Header: ***String toString()***
- The ***toString*** method is automatically invoked when an object is passed to ***println*** and the string concatenation operator
- Class ***Object*** provides an implementation of the ***toString*** method that returns a string consisting of the class name followed by an "at" sign (@) and the unsigned hexadecimal representation of the hash code
- ***toString*** is usually overridden so that it returns a descriptive string representation of the object

Polymorphism

- Every instance of a subclass is also an instance of its superclass, but not vice versa
- Polymorphism: An object of a subclass can be used wherever its superclass object is used
- Example

```
public class Demo {
    public static void main(String [] args) {
        m(new Point(1,2));
    }

    public static void m(Object x) {
        System.out.println(x);
    }
}
```

Dynamic Binding

- A method can be implemented in several classes along the inheritance chain
- The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable
  - E.g. **Object x = new Point(1,2);** //declared type: Object, actual type: Point
    System.out.println(x); //which toString is invoked?
- Dynamic binding works as follows:
  - Suppose an object x is an instance of classes C1, C2, . . . , Cn-1, and Cn, where C1 is a subclass of C2, C2 is a subclass of C3, . . . , and Cn-1 is a subclass of Cn,
  - If x invokes a method p, the JVM searches for the implementation of the method p in C1, C2, . . . , Cn-1, and Cn, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked

Encapsulation

- The details of implementation are encapsulated and hidden from the user
- Modules communicate only through their APIs and are oblivious to each others' inner workings
  - E.g. using ***System.out.println*** without knowing how it is implemented
- Advantages
  - Decoupling the modules that comprise a system allows them to be developed, tested, optimized, used, understood, and modified in isolation
  - Information hiding increases software reuse because modules that aren't tightly coupled often prove useful in other contexts

Encapsulation

- The access control mechanism in Java facilitates encapsulation
- There are four possible access levels for members, listed in order of increasing accessibility:
  1) **private**—The member is accessible only from the top-level class where it is declared
  2) **package-private**—The member is accessible from any class in the package where it is declared (default access)
  3) **protected**—The member is accessible from subclasses of the class where it is declared and from any class in the package where it is declared
  4) **public**—The member is accessible from anywhere
- Rule of thumb: *make each member as inaccessible as possible*