Principles of Object-Oriented Programming

The concept of object-oriented programming had its roots in SIMULA 67, a general purpose programming language developed at the Norwegian Computing Center. In 1980s, however, Smalltalk was developed and some consider Smalltalk to be the base model for a purely object-oriented programming language. [3]

What is an object? "An object is an instance of a class." An object is often created by instantiating a class with the `new` keyword and a constructor invocation. [2]

The following enumeration summarizes the properties of objects: [1]

- ... are structures with *state and behavior*.
- ... *cooperate* to perform complex tasks.
- ... can communicate with each other by means of *messages*.
- ... have precise *interfaces* specifying which messages they accept.
- ... have *hidden state*.

What is a class? "A class is a collection of data fields that hold values and methods that operate on those values. A class defines a new reference type." [2]

A class definition consists of a signature and a body. The class signature defines the name of the class. The body of the class is a set of members, such as fields and methods, and may include constructors, initializers, and nested types. Members can be static or nonstatic. A static member belongs to the class itself, whereby a nonstatic member is associated with the instance of a class (an object). [2]

Consider the following class named `Point`, providing two class fields `x`, `y` and three class methods `draw()`, `move()`, and `remove()`:

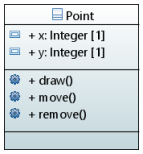

Figure 0.1: Class definition in UML notation

The signature of a class may declare that the class `extends` another class (see Inheritance). The extended class is known as superclass and the extension as the subclass. A subclass inherits the members of its superclass and may declare new members or override inherited methods with new implementations (see Polymorphism). The members of a class may have access modifiers `public`, `protected` or `private`. These modifiers specify their visibility and accessibility to clients and subclasses (see Abstraction, Encapsulation and Information Hiding).

What are the principles of the object-oriented approach? The principles of the object-oriented approach are the following:

- Abstraction
- Encapsulation and Information Hiding
- Inheritance
- Polymorphism

These principles will be discussed in more detail.

Abstraction

"An **abstraction** is a view or representation of an entity that includes only the most significant attributes. In a general sense, abstraction allows one to collect instances of entities into groups in which their common attributes need not be considered." In the world of programming languages, abstraction reduces the complexity of programming; its purpose is to simplify the programming process. [3]

The abstraction paradigm leads to *abstract data types* (ADTs). "An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations." An ADT specifies *what* each operation does, but not *how* it does it. In Java an ADT is expressed by an *interface*. An interface is a list of method declarations, whereby each method has an empty body. Java realizes an ADT by a concrete data structure, which is modeled by a class. The class implements an interface. The class specifies how the operations are performed in the body of each method. [6]

"An abstract data type is a data type whose representation is hidden from the client." [4]

Encapsulation and Information Hiding

Encapsulation means that "different components of a software system should not reveal (dt. zeigen, enthüllen) the internal details of their respective implementations." An advantage of encapsulation is that one programmer can implement the details of a component without explaining these internals to other programmers. However, he has to ensure maintaining the public interface of the component. So, encapsulation allows to change details of parts of the program without effecting other parts. [6]

In programming languages encapsulation is realized in different ways, for instance [5]:

- A procedure is an encapsulation of steps to perform a particular task.
- An array is an encapsulation of several elements of the same type.
- A class - in object-oriented programming - is an encapsulation to bundle data and operations.

Java supports encapsulation in different ways:

- It allows to bundle data and methods that operate on the data in an entity called class.
- It allows to bundle one or more logically related classes in an entity called a package.
- It allows to bundle one or more related classes in an entity called a compilation unit.

In object-oriented programming, encapsulation is simply the bundling of items together into one entity. In contrast to **information hiding**, which is the process of hiding implementation details that are likely to change. So, information hiding is concerned with how an item is hidden. Abstraction is only concerned about which item should be hidden. [5]

Information hiding is particularly important in object-oriented programming. Objects are often complex data structures with many components that should not be accessed by clients. There are a few reasons for such encapsulation [1]:

- *The representation of an object is subject to changes*. During development of a class it may occur that a different set of instance variables is better suited for the purpose of the class. If clients directly access the instance variables, they have to be modified in order to comply with the new representation.
- *Objects are not primarily used as containers.* The object's instance variables are used to represent the state of the object. The object itself is an active entity that can perform certain operations on request.
- *Objects often own other objects*. The ownership is private to the object; the relation between the object and other objects owned by it must not be disturbed by clients.
- *Objects are accessed via polymorphic references*. If an object is accessed via a variable `v`, the object referred to by `v` can change dynamically during execution of the program. We cannot be sure whether the variable `v` will always refer to an object with the same internal structure.

Inheritance

"In object-oriented programming, the mechanism for a modular and hierarchical organization is a technique known as inheritance." [6]

The problem with reusing abstract data types is that, in nearly all cases, the features and capabilities of the existing type are not quite right for the new use. The old type requires at least some minor modifications. A second problem with programming with

abstract data types is that the type definitions are all independent and are at the same level. That means that no parent-child combinations are possible. [3]

Inheritance offers a solution to both problems. The inheritance mechanism let us define a new abstraction by extending an existing abstraction. If a new abstract data type can inherit the data and functionality of some existing type, and is also allowed to modify some of those entities and add new entities, reuse is greatly facilitated without requiring changes to the reused abstract data type. Inheritance provides additionally a framework for the definition of hierarchies of related classes.

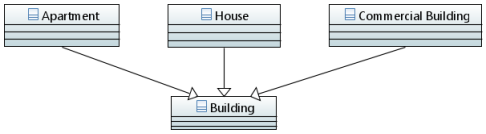

Figure 0.2: An example of an "is a" hierarchy involving architectural buildings.

The abstract data types in object-oriented languages, following SIMULA 67, are usually called classes. As with instances of abstract data types, class instances are called objects. A class that is defined through inheritance from another class is a child class, a subtype a derived class or subclass. A class from which the new class is derived is its parent class, supertype, base class or superclass. The subprograms that define the operations on objects of a class are called methods. The calls to methods are sometimes called messages. The entire collection of methods of an object is called the message protocol, or message interface, of the object. Computations in an object-oriented program are specified by messages sent from objects to other objects. [3]

Polymorphism

According to the Greek words "poly" (many) and "morphos" (form), the term polymorphism "is the ability of an entity (e.g. variable, class, method, object, code, parameter, ...) to take on different meanings in different contexts." The entity that takes on different meanings is known as a polymorphic entity. [5]

"In the context of object-oriented design, it refers to the ability of a reference variable to take different forms." [6]

There are two different categories of polymorphism:

- Ad-hoc Polymorphism ... the types for which code is written are finite and all those types must be known when the code is written. Ad-hoc polymorphism is divided into overloading and coercion (dt. Zwang) polymorphism.
- Universal Polymorphism ... the types for which code is written are infinite and this code will also work for new - not yet known - types. Universal polymorphism is divided into inclusion (dt. Einschluss) and parametric polymorphism.

Overloading Polymorphism Overloading results when a method or an operator has at least two definitions that work on different types. The methods are called overloaded methods and the operators are called overloaded operators. In Java, we are allowed defining overloaded methods. Java has some overloaded operators (e.g + for adding int or String variables), but does not allow to overload an operator for an ADT.

An example of an overloaded method in Java:

```
public class MathUtil {
        public static int max(int n1, int n2) {
        // returns maximum value of two integers }
        public static double max(double n1, double n2) {
        // returns maximum value of two floating-point numbers }
        public static int max(int[] num) {
        // returns maximum value of an array of integers }
}
```

The method `max()` is overloaded. It has three different definitions and each of its definitions perform the same task of computing the maximum value, but on different types.

The following code makes use of the methods:

```
int max1 = MathUtil.max(10, 34); // uses max(int,int)
double max2 = MathUtil.max(10.54, 3.56); // uses max(double,double)
int max3 = MathUtiel.max(new int[]{2, 56, 8, 45}); // uses max(int[])
```

In Java, the only requirement to overload a method name is that all versions of this method must differ in number and/or type of their formal parameters.

Coercion Polymorphism Coercion occurs when a type is implicitly converted to another type. This is done automatically.

Consider the following statements in Java:

```
int num = 808;
double d1 = (double) num; // explicit conversion of int to double by using type cast
double d2 = num; // implicit conversion of int to double (coercion)
```

Inclusion Polymorphism Inclusion occurs when a piece of source code that is written using a type works for all its subtypes. This is the most common type of polymorphism supported by object-oriented programming languages.

Java supports inclusion polymorphism using inheritance.

Parametric Polymorphism In parametric polymorphism source code is written that it works for any types. Sometimes it is also refered to generics.

Java supports parametric polymorphism since Java 5 through generics.

Common Student Questions [4]

Why the distinction between primitive and reference types? Why not just have reference types?

Performance. Java provides the reference types Integer, Double, and so forth that correspond to primitive types that can be used by programmers who prefer to ignore the distinction. Primitive types are closer to the types of data that are supported by computer hardware, so programs that use them usually run faster than programs that use corresponding reference types.

Do data types have to be abstract?

No. Java also allows `public` and `protected` to allow some clients to refer directly to instance variables. The following rules should give some guidance [1]:

- Public instance variables should be used for object characteristics that ...
  - are used frequently.
  - represent elementary object state (such as x and y coordinates of points).
  - are not crucial for the object's integrity.
  - are natural for an object that a change of the implementation is unlikely.
- Protected instance variables should be used in most cases. They are useful for ...
  - components that are only used internally.
  - tightly coupled components.
  - components that must be accessed by heirs in order to extend or override the object's functionality.
- Private instance variables are useful for ...
  - components that should not be known to heirs, such as safety-critical information.
  - auxiliary (dt. Helfer) instance variables that are only used within certain methods for some particular purpose.

What happens if I forget to use new when creating an object?

To Java, it looks as though you want to call a static method with a return value of the object type. Since you have not defined such a method, the error message is the same as anytime you refer to an undefined symbol.

What is a deprecated method?

A method that is no longer fully supported, but kept in an API (Application Programming Interface) to maintain compatibility.

Do It

1. *Object-oriented languages.*

   Describe the characteristic features of object-oriented languages.

2. *Inheritance.*

   What exactly does it mean for a subclass to have an is-a relationship with its parent class?

3. *Overriding.*

   Describe the issue of how closely the parameters of an overriding method must match those of the method it overrides.

4. *Adaptability.*

   Give an example of a software application in which adaptibility can mean the difference between a prolonged lifetime of sales and bankruptcy.

5. *Software Design.*

   Describe a component from a `text-editor` GUI and the methods that it encapsulates.

6. *UML Software Design.*

   Draw a class inheritance diagram for the following set of classes:

   - Class `Goat` extends `Object` and adds an instance variable `tail` and methods `milk()` and `jump()`.
   - Class `Pig` extends `Object` and adds an instance variable `nose` and methods `eat(food)` and `wallow()`.
   - Class `Horse` extends `Object` and adds instance variables `height` and `color`, and methods `run()` and `jump()`.
   - Class `Racer` extends `Horse` and adds a method `race()`.
   - Class `Equestrian` extends `Horse` and adds instance variables `weight` and `isTrained`, and methods `trot()` and `isTrained()`.

7. *UML Software Design.*

   Suppose you are on the design team for a new e-book reader. What are the primary classes and methods that the Java software for your reader will need? You should include an inheritance diagram for this code, but you don't need to write any actual code. Your software architecture should at least include ways for customers to buy new books, view their list of purchased books, and read their purchased books.

References

[1] Guenther Blaschek. *Object-Oriented Programming with Prototypes.* Springer Verlag, 1994.

[2] Benjamin J. Evans and David Flanagan. *Java in a Nutshell.* O'Reilly Media, 2015.

[3] Robert W. Sebesta. *Concepts of Programming Languages.* Pearson, 2012.

[4] Robert Sedgewick and Kevin Wayne. *Algorithms Fourth Edition.* Addison-Wesley, 2011.

[5] Kishori Sharan. *Beginning Java 8 Fundamentals.* Apress, 2014.

[6] Michael T. Goodrich Roberto Tamassi and Michael H. Goldwasser. *Data Structures and Algorithms in Java.* Wiley, 2014.