## Table of Contents

**Introduction** ................................................................................................................. 2  
1.1 Basic Concepts ........................................................................................................... 2  
1.1.1 Objects .................................................................................................................. 2  
1.1.2 Messages .................................................................................................................. 3  
1.1.3 Encapsulation .......................................................................................................... 4  
1.1.4 Classes ..................................................................................................................... 4  
1.1.5 Composition ............................................................................................................. 5  
1.1.6 Inheritance ............................................................................................................... 6  
1.1.7 Abstract Classes and Interfaces ............................................................................... 6  
1.1.8 Polymorphism ......................................................................................................... 7  

**2 Objects** ................................................................................................................... 8  
2.1 Life Cycle ..................................................................................................................... 8  
2.1.1 Creating Objects ....................................................................................................... 8  
2.1.2 Cleaning Up ............................................................................................................... 10  
2.2 Using Objects .............................................................................................................. 10  
2.2.1 Variables .................................................................................................................... 10  
2.2.2 Methods .................................................................................................................... 11  
2.3 Storing Objects ............................................................................................................ 11  
2.3.1 Arrays ....................................................................................................................... 12  
2.3.2 Collections ............................................................................................................... 12  
2.3.3 Iterators ..................................................................................................................... 13  
2.4 Singly Rooted Hierarchy ............................................................................................... 14  

**3 Classes** .................................................................................................................... 15  
3.1 Creating Classes ......................................................................................................... 15  
3.2 Instance variables ........................................................................................................ 15  
3.2.1 Default Values .......................................................................................................... 16  
3.3 Instance methods ......................................................................................................... 17  
3.3.1 Declaring a method ................................................................................................. 17  
3.3.2 Hidden Implementation ......................................................................................... 18  
3.4 Constructors ............................................................................................................... 19
Introduction

Object-oriented programming approach allows programmers to write computer programs by representing real-world entities in the form of so-called software objects or objects for short. Typically, objects represent both: attributes of real-world entities and their behavior. For example, we can represent a car as an object that has attributes such as color, wheels, gear, brakes, and so on. Additionally, behavior of the car can be represented by means of so-called messages that the software object accepts. The messages can be sent to the object to change the state of its attributes. For example, the car object might accept messages such as change gear, accelerate, or brake.

A typical object-oriented program is simply a collection of a number of objects representing different real-world entities. All the computational work is accomplished through interaction between objects from the collection, i.e., through the messages being exchanged among the objects. For example, imagine that we have another object representing a person driving the car from above. The person object might accelerate the car object by sending it the accelerate message.

The object-oriented approach to writing computer programs has been a huge step forward in development of programming paradigms in comparison to procedural programming approach. It has brought numerous advantages in software design and development process. Let us investigate here shortly only the most obvious advantage of object-oriented programming. For example, in procedural programming we model real-world entities such as cars and persons by means of procedures and sequences of their execution. However, this seems a rather unnatural way of thinking and modeling of real-world entities since humans see a car as an object with an engine, a gas tank, four wheels, etc. rather than a series of procedures that makes it run. Obviously, it is much more natural for humans to think about real-world entities as objects that have certain attributes and behavior.

1.1 Basic Concepts

The basic concepts of object-oriented programming are:

- Objects
- Messages
- Encapsulation
- Classes
- Composition (Aggregation)
- Inheritance (Specialization)
- Abstract Classes and Interfaces
- Polymorphism

1.1.1 Objects

Real-world entities are distinguished through:

- Attributes and their current state
- Behavior.
For example, dogs have attributes such as name, color, or breed. Additionally, each dog has a current state of these attributes, e.g. there is a dog called Snoopy, whose color is white and who is a beagle dog. Finally, dogs have a particular behavior such as barking, fetching, and wagging tail. Similarly, cars have attributes such as gear, speed, or color; the state is defined through the current gear, current speed and the actual color of a car; and cars have behavior such as braking, accelerating, slowing down, or changing gears.

In object-oriented programs objects are modeled closely after real-world entities - they too have attributes, their current state and behavior:

- Attributes are represented as one or more object variables. A variable is an item of data named by an identifier. The current values of the object variables correspond to the current object state.
- Object implements its behavior by means of so-called methods. A method is a procedure (function, subroutine) associated with an object.

Thus, we can define an object as a piece of software that combines variables, their current values and a number of methods. For example, we might represent a car as an object that has variables called gear, speed, or color. The values of these variables indicate its current state: the current gear is the 4th gear; the current speed is 100 km/h; the color is black. In addition to the variables the car object has methods to change its current state: to brake, accelerate or change the current gear. Typically, a method execution leads to changed values of the object variables.

Formally, we call object variables *instance variables*, and object methods are formally known as *instance methods*.

### 1.1.2 Messages

Single objects are not very useful. Typically, an object-oriented program is comprised of a (possibly huge) number of objects. In such a program all computational work is done by interaction between these objects. For instance, the car object from the previous example is not very useful if it is alone in an object-oriented program. However, if we have another object representing a person, the person object may interact with the car object by telling it to change the current gear, to speed up, etc.

Speaking more formally, an object-oriented program consists of a number of objects, which interact with each other by sending so-called *messages* to each other. Thus, whenever one object wants to change the state of another object it sends a message to that second object.

Technically speaking, sending a message to an object means invoking (calling and executing) an instance method of that object, i.e. it is simply a function (procedure, subroutine) call as known from procedural programs.

Sometimes, the receiving object requires more information in order to execute its method. For instance, if the car object has to change its gear it has to know which is the desired gear. Technically, this is accomplished by means of arguments that are passed to an instance method. Summarizing, a message is comprised out of the three following components:

- The object that receives the message (the car object)
- The name of the method to execute (changeGear)
- Additional parameters needed to execute the method (the 4th gear)
1.1.3 Encapsulation

The current values of instance variables make up the current state of an object. These instance variables are internal or private to an object. Other objects from an object-oriented program can not access and manipulate these instance variables directly. The only possibility in which other objects can alter the state of an object is by sending messages to it, i.e., by invoking its methods. Thus, the current state of an object is hidden from other objects. It is however, surrounded by its methods that provide an interaction interface between this object and other objects. Packaging an object's variables within the protective custody of its methods is called data encapsulation. That means the object remains in control of how the outside world is allowed to use it by bundling the complete code (instance variables and instance methods) internally into a single place.

The concept of data encapsulation is one of the most important concepts of object-oriented programming approach and exhibits a number of advantages. Most important benefits for programmers coming from this idea are:

- Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your car to someone else, and it will still work.

- Information hiding: An object has an interface that other objects can use to interact with it. By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world. You don't need to understand the gear mechanism on your car to use it.

- Code reuse: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

- Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

1.1.4 Classes

There are real-world entities that are very often similar to each other, i.e. they are of the same kind. For example, a black and a white car are similar to each other - they are both of the same kind, i.e. they are both cars. They only differ in color, i.e. they only differ in a single aspect of their current state. However, the attributes and behavior are same in both cases.

Similarly, in object-oriented programs we might have a number of objects of the same kind, which have the same instance variables and same instance methods. The only difference between these objects of the same kind is their current state, i.e. they typically have different values of their instance variables.

To model this situation in object-oriented programs programmers define so-called classes. A class can be seen as a prototype or as a template that defines instance variables and instance methods that are common to all objects of the same kind.

For example, a car class might define:

- Instance variables such as gear, speed, color, and so on.

- Instance methods needed to change the state of particular objects belonging to the car class. Thus, methods such as changeGear, break, or accelerate might be defined.
Defining a class in object-oriented software means writing a piece of source code that declares instance variables and declares and implements instance methods for that class of objects. Note here, that the instance variables are internal to this source code and no other piece of code has an access to it. Additionally, through declaration of instance methods a class defines an interaction interface or simply class interface that lists all messages that objects of that class accept. The only possibility to change the state of an object of that class is by invoking one of the methods listed in the class interface. A particular class implementation of methods is hidden within the class source code and other objects do not need to know anything about that implementation, i.e. they are completely independent on it. This is the essence of data encapsulation and enforces modularity, information hiding, code reuse, and pluggability.

Let us clarify another terminology issue here. We already call object variables and object methods instance variables and instance methods. Once, when we have a class (of which we should think as a prototype or as a template) we can use that definition to create any number of objects of that class. For each of these objects the object-oriented system allocates memory to store its instance variables and relate them with the instance methods. Each object manages its current state by means of the current values of its instance variables. In object-oriented terminology objects of a class are typically called class instances.

1.1.5 Composition

Some real-world entities are quite complex. Usually, such complex entities are composed of a (possibly) large number of simpler entities. For instance, a car is composed of (at least):

- An engine
- A frame
- A car body
- Suspension
- Breaks
- Wheels.

Object-oriented programming allows programmers to follow the same composition (aggregation) approach. Thus, it is possible to create more complex objects by combining already existing (simpler) objects into a new one. Technically, a class defining such a complex or composite objects has as its instance variables other objects.

Let us now revisit the car object from above and try to investigate the structure of that object. Obviously, the car object can be seen as a composition of a number of other simpler objects:

- An engine object
- A car frame object
- A breaks object
- Four wheel objects, and so on.

In general, the current state of the car object is determined by the current values of all of its instance variables – in this particular case the state of the car object is determined by the state of all the objects that belong to it. For example, color of the car object as a whole is determined by color of the car frame object belonging to it.

The composition principle greatly facilitates the reuse of software components. We take already existing class definitions and define a new class that composes the existing classes into a
Composite structure. A particular method of the composite class may just call a method of some of its components, and thus reuse the implementation from the component class.

### 1.1.6 Inheritance

Some real-world entities are specializations of other entities. For instance, sport cars are still cars, but they are more specialized cars designed for racing. Similarly, trucks are specialized cars for transporting goods; passenger cars are cars for traveling, and so on.

Object-oriented programming paradigm allows programmers to define more special cases of a class through the concept of subclassing (specialization). The special case of a class is called subclass, and the starting class is called superclass. For example, the car class from our example would be superclass with subclasses such as:

- Sport car class
- Passenger car class
- Truck class, and so on.

In object-oriented programming paradigm subclasses and their corresponding superclasses are tightly related by means of inheritance, another important object-oriented principle. We say that each subclass inherits properties from its superclass. This includes:

- Instance variables - sport cars, trucks and passenger cars share the same instance variables: gear, speed, and the like.
- Instance methods – again sport cars, trucks and passenger cars share the same behavior: braking and changing speed, for example.

However, subclasses are not limited to the instance variables and methods provided to them by their superclass. Subclasses can add variables and methods to those inherited from the superclass. For example, sport cars have two seats and 6 gears, trucks have trailer vehicle and so on. These attributes and all the additional behavior can be reflected by the subclasses.

Subclasses can also override inherited instance methods and provide specialized implementations for those methods. For example, if we have a sport car with an extra set of gears, we can override the "change gears" method so that a driver could use those new gears.

Finally, programmers are not limited to just one layer of inheritance, i.e. a class that is a subclass of another class can have itself a number of subclasses. For example, there are several types of sport cars, such as open-wheelers or touring cars. Such a class hierarchy (inheritance tree) can be as deep as needed. Instance methods and variables are inherited through the tree levels. In general, the farther down in the hierarchy a class appears it possesses more specialized behavior and attributes.

Similar to the composition principle the inheritance principle supports reuse of software components. However, with inheritance we primarily reuse the interface of the superclass, whereas with composition we reused the features provided by the components. The subclass has the same interface as the superclass, thus its instances can receive the same messages as its superclass.

### 1.1.7 Abstract Classes and Interfaces

Often, there is a need to create a superclass called an abstract class that defines only "generic" behavior. Such an abstract class defines a so-called abstract interface and may partially implement the instance methods from that interface. However in a typical case much of the class
is undefined and most of the instance methods are left unimplemented. The details and implementation are provided within specialized subclasses.

Abstract classes declare one or more methods as abstract methods. When the class is inherited the subclass has to implement all abstract methods or the subclass is considered to be abstract as well. It is not possible to create instances of abstract classes.

Let us look on an example to illustrate the usage of abstract classes. In an object-oriented drawing application, we can draw circles, rectangles, or lines. These objects all have certain attributes such as position, orientation, line color, fill color and certain behavior such as move, rotate, resize, or draw in common. Some of these attributes and behavior are the same for all graphic objects, e.g. position, fill color, or move. Others require different implementations, e.g. resize or draw. All objects must know how to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. We can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract superclass that implements all common instance methods. However, each particular subclass will need to implement its specific behavior, thus implementing abstract instance methods such as draw or resize.

Another similar concept is the interface concept. You can think about an interface as an abstract class with all methods declared as abstract. Moreover, an interface explicitly forbids any method implementation, allowing programmers to declare just a number of empty methods. This principle is very useful when the software is developed by a team of developers since such interfaces represent a contract between different developers. Basically, each developer needs to fulfill the contract by implementing the declared interface. However, the advantage here is that the developers can work independently since a particular implementation of the interface is not important to the others.

1.1.8 Polymorphism

In object-oriented programs programmers often want to treat an object as an instance of its superclass. For example, a driver object sends a message to a car object regardless of the specific type of that car. The driver can accelerate, brake or change gears of the car object without knowing if this car is a sport car, a truck, a passenger car or even an instance of a car type that is not defined yet. This allows programmers to write code that does not depend on a specific subclass. Rather this code depends just on the generic car superclass.

However, there is a problem with attempting to treat instances of a subclass as instances of its superclass. Technically, at the run-time the driver object sends a message to an instance of a specific subclass, i.e., the driver steers a sport car, a truck or a passenger car. Thus, the system has to delegate a message from the driver object, which is sent to an instance of the car superclass to an instance of the proper subclass. To achieve this object-oriented systems do not bind a call to a specific method at the compile-time but rather they do it at the run-time when the real identity of objects is known.

This whole principle of interchangeable objects is called polymorphism. Consider again the car example. The family of classes (all based on the same uniform interface) was defined beforehand. To demonstrate polymorphism, we would write a single piece of code that ignores the specific details of subclasses and talks only to the base class. We decouple that code from subclass-specific information, and thus the code is simpler to write and easier to understand. And, if we add a new class, the van class for example, through inheritance, the code we write will work just as well for the new class of cars as it did for the existing classes. Thus, the program is extensible.
2 Objects

From the design point of view, object-oriented programming is just about abstract data typing, inheritance, and polymorphism, but other more technical issues can be at least as important.

Among most important issues are:

- The way how objects are created and destroyed
- Where is the data for an object stored?

There are several different approaches to control such issues. We can group these approaches in the following two groups:

- Approaches that support efficiency
- Approaches that support flexibility.

To support efficiency storage and lifetime of objects can be determined when the program is written. Thus, programmers may choose to place objects on the stack, registers, etc. Programming languages such as C++ support this approach. Such an approach provides for maximum run-time speed, however at the cost of flexibility. Programmer has to know the exact quantity, lifetime, and type of objects while the program is written.

In order to support flexibility objects can be created dynamically in a pool of memory called heap. In this case programmer does not need to know at the time of program writing the exact quantity, lifetime or type of objects. This approach is quite flexible, although much slower than the first approach because the system needs to manage the heap memory - dynamically and at the run-time. Programming languages such as Java support this approach.

Such issues and the manner in which they are solved strongly influence a so-called life cycle of objects. In the next sections we will examine the second approach (Java approach) in more details.

2.1 Life Cycle

Java programming language facilitates the approach that supports flexibility. Whenever a programmer wants to create an object, `new` keyword is used to build an instance of that object within the heap memory, i.e. to dynamically allocate enough of the heap memory to store all of the instance variables.

Now, how does the life cycle of a dynamically created object look like? Obviously, if the Java system creates objects dynamically at the run-time and store it on the heap the programmer has no knowledge and influence on its lifetime, i.e. there is no way for the programmer to destroy the object and free its memory at the time of program writing. Hence, Java provides a feature called a garbage collector that automatically discovers when an object is no longer in use and destroys it. Therefore, in Java there is no need to destroy an instance of a class explicitly. Rather the system takes care of all objects that are no longer in use by destroying them and freeing the memory that these objects allocated.

2.1.1 Creating Objects

A class provides a prototype for creating objects; thus we create objects or instances of classes. The following statements provide examples of creating objects in Java:

```
Point origin = new Point (100, 100);
```
Circle circle = new Circle (origin, 50);

Rectangle rectangle = new Rectangle (origin, 80, 50);

The first statement creates an instance of the Point class; the second statement creates an instance of the Circle class and the third statement creates an instance of the Rectangle class.

Each of these three statements consists of the three fundamental parts:

- Variable declaration - it associates a name with a type
- Instantiation – the operator `new` creates a new object, i.e., it dynamically allocates memory space for it
- Initialization – the new operator is followed by the call to a constructor, which initializes the newly created object.

Let us examine now all of these fundamental parts in more details. To declare a variable we write:

Type name

Thus, we will use `name` to refer to a variable, which type is `Type`. In the example from above the part of the code in bold declares variables:

Point origin = new Point (100, 100);

Circle circle = new Circle (origin, 50);

Rectangle rectangle = new Rectangle (origin, 80, 50);

Note that a variable declaration does not create an object yet - it just declares a variable with that name and that type. The variable will refer to an object, once when this object is created, i.e., variables in Java of a non-primitive data type (e.g. an object of a particular class) are technically references and they keep addresses of memory where a particular object is stored.

Instantiation is carried out by means of `new` operator. The `new` operator creates an instance of a class by allocating memory for this new object. It requires a single, postfix argument: a call to a constructor. The name of the constructor corresponds to the name of the class to instantiate. The constructor initializes the new object.

The `new` operator returns a reference to the object that has been created and this reference is assigned to a variable of the appropriate type.

Point origin = new Point (100, 100);

Circle circle = new Circle (origin, 50);

Rectangle rectangle = new Rectangle (origin, 80, 50);

To initialize objects each class definition provides a number of constructors, which lets you provide the initial values for instances of that class. For example, the Rectangle class provides a constructor that requires those three initial values:

- Origin
- Width
- Height.
The constructor itself implements the initialization of instance variables with the values that are supplied to it by assigning the passed values to the instance variables.

The code in bold is where the initialization occurs:

```java
Point origin = new Point (100, 100);
Circle circle = new Circle (origin, 50);
Rectangle rectangle = new Rectangle (origin, 80, 50);
```

### 2.1.2 Cleaning Up

The Java platform allows programmers to create as many objects as they want (limited, of course, by what your system can handle), and they don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called garbage collection.

An object is eligible for garbage collection when there are no more references to that object, i.e. when there are no variables left holding at least one reference to the object. A reference is typically dropped when the variable that holds it goes out of scope. Programmers can explicitly drop an object reference by setting the variable to the special value `null`. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

### 2.2 Using Objects

Once when objects are created in an object-oriented program, we use them to perform different tasks. We might use objects in two different ways:

- Manipulating and inspecting its variables
- Calling its methods.

#### 2.2.1 Variables

To access an object’s variable in Java we write:

```java
object_reference.variable_name
```

The first part of the code, `object_reference`, must be a reference to an object. The second part of the code above, `variable_name`, is the valid name of an instance variable of that object.

In order to access objects’ variables in the way that is presented above, these instance variables must be declared to be `public` variables. However, **declaring a public instance variable is strongly discouraged.** It obstructs the principle of *data encapsulation*, i.e. instance variables are not encapsulated within an object. Such objects allow direct manipulation of their instance variables, instead of providing methods that to manipulate their instance variables, thus endangering the basic object-orientation principles. For example, modularity of the code is not longer guaranteed since there can be code outside of the object’s code that manipulates its instance variables, i.e. the object’s code is not self-contained and modular anymore.

Java provides a so-called access control mechanism to control access to its variables. Thus, variables might be declared to be `public`, `private` or `protected`. As a rule of the thumb, all
instance variables must be declared either as **private** or **protected**. The only situation when the **public** access modifier might be used is to declare a constant.

The difference between **private** and **protected** is the following – private instance variables are visible only to the instances of that particular class; protected instance variables are visibly to the instances of the class in question and to instances of all subclasses of that particular class.

### 2.2.2 Methods

To call a public method of an object in Java we write:

```java
object_reference.methodName (arguments_list);
```

If the method does not take any arguments we write:

```java
object_reference.methodName ();
```

For example, imagine that the Circle class from above provides the following two public methods:

- Move method to modify the origin of a circle; as an argument this method takes a Point object which becomes the new origin of that circle
- Scale method to alter the radius of a circle; as an argument this method takes a real number that determines the scale factor.

We could invoke these methods by writing:

```java
circle.move (new Point (100, 50));
circle.scale (0.5);
```

Note, that calling a method of an object is equivalent to sending a message to that object.

Sometimes, methods return values. For example, imagine a method of the Circle class, which calculates the area of a particular circle. Obviously, such method would return a real number.

```java
double circle_area = circle.getArea ();
```

All methods we showed until now are **public** methods of the Circle class, i.e. they are a part of the public interface of that class. Any object can invoke a public method of a particular object. Sometimes, a class needs to restrict access to the methods of its instances. For example, a class might have a method that only instances of its subclasses are allowed to call – in that case the method is declared as **protected**. Similarly, if only instances of the class in question are supposed to call a particular method the access modifier for that method should be declared as **private**.

### 2.3 Storing Objects

In an object-oriented program a number of objects at the run-time can become very large. Obviously, we need to store and manage these objects in a way that we can easily access and use them. A particular mechanism for storing and managing objects depends on the type of objects, as well as the number of created objects. Very often, we store objects in:

- **Arrays**
- **Collections**

For an easy and fast access to the available objects so-called **iterators** are typically facilitated.
2.3.1 Arrays

An array is a structure that holds a fixed number of values of the same type. Thus, an array is a fixed-length structure.

An array element is one of the values within an array and is accessed by its position within the array.

```java
String[] array = { "String One", "String Two", "String Three" };
```

The above expression creates a String array with three elements. To access the elements of this array we would write something like this:

```java
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i].toLowerCase());
}
```

The code in bold accesses the i-th element of the array. Thus, an element within an array can be accessed by its index. Indices begin at 0 and end at the length of the array minus 1.

Summarizing, the length of the array must be specified when it is created. We can use the `new` operator to create an array, or we can use an array initializer. Once created, the size of the array cannot change. To get the length of the array, we use the `length` attribute. The code in bold gets the length of an array:

```java
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i].toLowerCase());
}
```

2.3.2 Collections

If we want to store data of different types in a single structure, or if we need a structure with a non fixed-length, we use a collection implementation instead of an array. Usually, a collection is an object that holds references to other objects. Every good object-oriented programming language comes with a set of different implementations of collection classes in its library.

Java also has collections in its standard library. Moreover, the Java library has different types of collections for different needs: a vector (called an ArrayList in Java) for consistent access to all elements, and a linked list for consistent insertion of elements, etc. Thus, the programmer can choose the particular type that fits his/her needs.

Let us look on an example. We use ArrayList as an example of collections in Java:

```java
List list = new ArrayList();
list.add(" dog");
list.add(" cat");
list.add(new Integer(10));
```

The first expression creates an empty ArrayList object. The next three statements add objects to the list object. First two objects are String objects, whereas the third object is an instance of Integer class. There are a number of other methods supported by the ArrayList class that we can
use to add element to an instance of that object. For example, if we want to insert an object into a
list on a specific position we would write something like this:

\[
\text{list.set(1, new Rectangle(new Point(0, 10), 100, 50));}
\]

The above expression inserts an instance of the Rectangle class at the second position in the list.
Note that list indices begin also at 0, as it is the case with arrays in Java.

To access an object from a list we write the following code:

\[
\text{Rectangle rect = (Rectangle) list.get(1);}
\]

Since collections can hold any type of objects, we need to cast the returned object to its real type.
The cast in the above example is represented by the code in bold. This is a known disadvantage of
all collection objects: the type of objects that they hold is unknown to collections, i.e., once you
put your objects into a collection their type is “lost”. All that is known to a collection is that it
holds objects (of any type).

Collections also support deleting of objects that they hold. For example:

\[
\text{list.remove(1);}
\]

The above statement deletes the second element from the list.

### 2.3.3 Iterators

In the above list example we saw how we can use an ArrayList object to store objects. However,
Java offers a number of other collection classes, such as LinkedList, HashSet, and so on that we
can also use to store objects. Obviously, each of these collection classes provides a specific public
interface with a set of different methods to manipulate collection elements.

However, there is a drawback to this - we need to know the exact type of the collection in order to
use it. In order to overcome this drawback we apply the concept of an *iterator*.

An iterator is an object that can move through a sequence (of any type) of objects and select each
object in that sequence. The programmer does not need to know or care about the underlying
structure of that sequence.

Let us look on an example:

\[
\text{Iterator iterator = list.iterator();}
\]

\[
\text{while(iterator.hasNext())}
\]

\[
\text{iterator.next().toString();}
\]

First, we ask a list object for an iterator object. Once when the iterator object is obtained, it can be
used to traverse through the elements of a particular list (the code in bold).

Let us look at the following code to see the advantages of iterator approach. Suppose that we
created an instance of HashSet class named `hash` beforehand:

\[
\text{Iterator iterator = hash.iterator();}
\]

\[
\text{while(iterator.hasNext())}
\]

\[
\text{iterator.next().toString();}
\]

Thus, the same code is used as with the list object.

All collection classes in Java provide methods to obtain different standardized iterators.
2.4 Singly Rooted Hierarchy

We mentioned before that collections keep objects of any type. Internally, collections maintain an array of objects that can be of different type. However, arrays in Java hold only objects of a single type. How is it possible for collections to keep objects of different types if they maintain an array of objects? The answer to that question is a rather simple one. Ultimately, all objects in Java are of the same type, because all classes in Java are subclasses of a single base class. In Java the name of this ultimate base class is simply **Object**. Such a class hierarchy is called *singly rooted hierarchy*.

It turns out that the benefits of the singly rooted hierarchy are numerous:

- All objects in a singly rooted hierarchy have an interface in common, so they are all ultimately the same type.
- All objects in a singly rooted hierarchy can be guaranteed to have certain functionality. For example, the Object class defines the method `toString` used to print out an object. With a singly rooted hierarchy any Java object can be printed out by means of `toString` method.
- A singly rooted hierarchy makes it much easier to implement a garbage collector (which is conveniently built into Java). The necessary support can be installed in the base class, and the garbage collector can thus send the appropriate messages to every object in the system.
- Since run-time type information is guaranteed to be in all objects, you’ll never end up with an object whose type you cannot determine.
3 Classes

We learned how to create, destroy and use objects. Now it is the time to see how to create class definitions, which we use as prototypes for objects.

3.1 Creating Classes

As we already mentioned a class definition:

- Declares a number of instance variables
- Defines a number of public methods that we use to manipulate the instance variables

In addition to that, a class definition might also define a number of so-called constructors, which are used to initialize instances of that class, at the creation time. Some object-oriented programming languages, such as C++, require from programmers to define also destructors, which are used to free up the memory that objects have occupied. However, the garbage collector in Java takes care of cleaning-up objects, thus there is no destructor definitions required in Java.

The first thing each class definition starts with is the declaration of the class:

```java
public class Stack{
    // class definition
    // ....
}
```

The code in bold declares a public class with the name Stack. Public class means that other classes, either for inheritance or for composition, might use this class.

A class definition follows to a class declaration in a so-called class body. Thus, a class body declares instance variables; methods for manipulating instance variables and might additionally define constructors.

If a class definition does not define a constructor, Java provides this class with a default empty public constructor.

3.2 Instance variables

To declare an instance variable for our Stack class we could write a line of code that might look as follows:

```java
public class Stack{
    private ArrayList items_;
    // ....
}
```

The code in bold declares a single instance variable in the Stack class. Note that this declaration starts with the `private` access modifier to encapsulate the instance variable. By declaring an instance variable to be private we forbid objects of other classes to access this variable directly,
thus making it possible for other objects to manipulate this instance variable just through the instance methods.

However, declaring an instance variable as private forbids also objects of subclasses to access that particular instance variable directly. Sometimes it is more convenient to allow subclasses to access instance variables of their superclass directly. The `protected` access modifier supports this situation.

Lastly, an instance variable might be declared as `public`, but as discussed above, this seriously damages the encapsulation principle and is strongly discouraged. In special cases when an instance variable is actually a constant the `public` access modifier might be used:

```java
public class Math{
    public final static double PI = 3.14;
    // …
}
```

Note that the PI variable is also declared as `final` and `static`, which makes it constant. `final` ensures that no new value can be assigned to that variable, and `static` declares this variable as a so-called class variable available to all instances of the class – technically this means that there is only one single memory address storing that variable for all instances of the class. To access this constant we can write something like this:

```java
double area = radius * radius * Math.PI;
```

### 3.2.1 Default Values

When we declare an instance variable we can also initialize it to a certain value. In the case of the Math class we initialized the PI constant with a double value.

If we do not supply a value when we declare an instance variable then this instance variable obtains a default value. Java has default values for each primitive type, as well as for object references. Here is the table of default values for variables in Java:

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>char</td>
<td>‘\u0000’</td>
</tr>
<tr>
<td>byte</td>
<td>(byte)0</td>
</tr>
<tr>
<td>short</td>
<td>(short)0</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>0L</td>
</tr>
<tr>
<td>float</td>
<td>0.0f</td>
</tr>
<tr>
<td>double</td>
<td>0.0d</td>
</tr>
<tr>
<td>reference</td>
<td>null</td>
</tr>
</tbody>
</table>
Software Architecture

The other way in which we can initialize instance variables is to implement constructors. Constructors take arguments, thus we can initialize instance variables with parameterized values.

3.3 Instance methods

Classes define methods for manipulating their instance variables. That means that a class provides method declaration, i.e., it declares the name of the method, its access modifier, its arguments and its return values. In addition to that a class implements the method body, i.e., it provides a complete definition of its instance methods.

Sometimes, when a method is declared abstract and hence the whole class is declared to be abstract then the class just declares the method and does not provide the method body, i.e., it does not implement it. In this case, subclasses of an abstract class are supposed to provide this implementation.

3.3.1 Declaring a method

Let us look on an example of methods declaration:

```java
public class Stack{
    private ArrayList items_;

    public void push(Object item){
        // method body
    }

    public Object pop(){
        // method body
    }

    public boolean isEmpty(){
        // method body
    }
}
```

The code in bold declares three instance methods. Let us investigate this code more closely. First thing to notice is the access modifier of these three methods. In all three cases methods are declared to be `public` methods. That means that objects of other classes might invoke these methods, they can send these messages to instances of the Stack class. Thus, the Stack class provides a public interface, which is the single point of access when other objects want to manipulate its instance variables. This follows the basic object-oriented principles as discussed above.

However, there are situation when a class wants to restrict access to its methods, thus declaring them to be `protected` or `private`. This can be case whenever a method is used just for some internal manipulation of the data. The difference between protected and private is same as it was
case for instance variables. Hence, a private method is accessible only from instances of the class itself, whereas a protected method might be accessed from instances of all subclasses as well.

The next important thing in a method declaration is the declaration of the return value. If the method does not return a value we declare it as `void` as in the case of the `push` method. Otherwise we declare a method to return a value of certain type, such as `Object` in the case of the `pop` method or `boolean` in the case of `isEmpty` method.

Finally, we provide the name for a method and list all arguments that a method takes. For example in the case of the `push` method we take an argument of type `Object`, whereas two remaining methods do not take any arguments.

### 3.3.2 Hidden Implementation

The method body provides a particular implementation. Note that this implementation is *hidden* from all other objects. All that other objects know about instances of a particular class is their public interface. They do not know and are in fact not interested in a particular implementation provided by a class – this greatly supports modularity in object-oriented programs. Moreover, a particular implementation can be easily replaced by say a more efficient one, as long as the interface is satisfied and the expected results remain the same as before.

Let us again look on the Stack example:

```java
public class Stack{
    private ArrayList items_; 

    public void push(Object item){
        items_.add(item);
    }

    public Object pop(){
        int length = items_.size();
        if(length_ == 0) throw new EmptyStackException();
        Object item = items_.get(length – 1);
        Items_.remove(length – 1);
        return item;
    }

    public boolean isEmpty(){
        return (items_.size() == 0);
    }
}
```
Software Architecture

The code in bold is a particular implementation of three public methods of the Stack class. Note that a method can throw a so-called exception (EmptyStackException in the pop method) if something goes wrong during its execution. The code of the class that calls this method should take steps to handle such exceptional states.

3.4 Constructors

The last part of a class definition covers the definition of constructors. We define constructors in order to provide a parameterized initialization of instances of a class. If we don’t have the need for parameterized initialization we can create a class definition without defining constructors. In that particular case, Java provides a so-called default empty constructor. Thus, the code without a constructor for the Stack class from above is in fact equivalent to this one:

```java
public class Stack{
    private ArrayList items_;
    
    public Stack(){
    }
    
    public void push(Object item){
        items_.add(item);
    }
    
    public Object pop(){
        int length = items_.size();
        if(length_ == 0)
            throw new EmptyStackException();
        Object item = items_.get(length – 1);
        Items_.remove(length – 1);
        return item;
    }
    
    public boolean isEmpty(){
        return (items_.size() == 0);
    }
}
```

The default constructor is shown in bold. In general, constructors have the following properties:
- Access modifiers (public, private, protected) similar to instance variables and methods
- Name of a constructor is same as the name of the class
- Constructors don’t have return values
- Constructors can take arguments

Since the default constructor does not take any arguments we must define our own constructor if there is a need to initialize instance variables. Typically, the constructor body implements such an initialization. For example, the Stack class has as its only instance variable items_ variable of type ArrayList. Since items_ is a reference to an instance of the ArrayList class by default the value of that variable is set to `null`. Since all methods of the Stack class operate on items_ variable we must initialize this instance variable and assign it a certain value. The following code excerpt provides examples of variable initialization.

```java
public class Stack{
    private ArrayList items_;  
    public Stack(){ 
        items_ = new ArrayList();  
    }
    public Stack(int initial_size){ 
        items_ = new ArrayList(initial_size);  
    }
    public void push(Object item){ 
        items_.add(item); 
    }
    public Object pop(){ 
        int length = items_.size(); 
        if(length_ == 0) 
            throw new EmptyStackException(); 
        Object item = items_.get(length – 1); 
        Items_.remove(length – 1); 
        return item; 
    }
    public boolean isEmpty(){ 
        return (items_.size() == 0); 
    }
}
```
Thus, we extended the default constructor by defining the constructor body and additionally we defined another constructor as well. The default constructor initializes now the instance variable of the Stack class. The new constructor takes as an argument an integer value and initializes the instance variable by using this value. Thus, it creates an instance of the ArrayList class with the specified initial size.