KAZAKH-BRITISH TECHNICAL UNIVERSITY

ALMATY 2013

Content

Introduction 7

1 Object-oriented Approach 9

1.1 Paradigm of Object-oriented Programming 9

1.2 Java as an Object-oriented language 13

2 Introduction to Java programming language 18

2.1 Why java ? (Java features) 19

2.2 Java Virtual Machine (JVM) 21

2.3 Compiling and running Java program 22

2.4 Java Syntax 23

3 Fundamentals of Objects and Classes 39

3.1 What is a class? 39

3.2 What is an object? 43

3.3 Class Modifiers 46

3.4 Class members and instance members 46

3.5 More on Methods 48

3.6 Encapsulation 51

3.7 Constructors 53

3.8 *This* Keyword 55

3.9 Initialization Blocks 57

3.10 Class Importation 58

3.11 Class Abstraction 60

3.12 Enumerations 60

3.13 Wrapper Classes 61

3.14 Core Java classes 62
3.15 Defining your own classes 65
3.16 Class design 70
4 Inheritance, Polymorphism and Abstract Classes 78
4.1 Concept of Inheritance. Subclass and Supeclass 78
4.2 Class Hierarchy 79
4.3 Special Keyword *super* 80
4.4 Phases of Object Creation 81
4.5 Overloading and Overriding Methods 83
4.6 Polymorphism 84
4.7 Type conversion 89
4.8 Abstract classes and methods 93
4.9 Object class 96
4.10 Design hints for inheritance 98
4.11 Detailed example 99
5 Interfaces 110
5.1 Interfaces : Basic Concept 111
5.2 Interface members 112
5.3 Implementation Procedure 113
5.4 Extending interfaces 114
5.5 Why do we use interfaces? 117
5.6 Marker Interfaces 119
5.7 Object Cloning 120
5.8 Interfaces vs Abstract Classes 123
5.9 Example : Stack 125
5.10 Inner classes 126

5.11 Interfaces & Inheritance. When to use what? 127
5.12 Principles of Object-oriented Design 129
6 UML diagrams 136
6.1 Use Case Diagrams 137
6.2 Sequence Diagrams 138
6.3 Class Diagrams 140
6.4 Association 144
6.5 Generalization 146
6.6 Dependency 147
6.7 Realization 149
6.8 Aggregation 149
6.9 Composition 151
6.10 Important issues – multiplicity, constraints and notes 153
6.11 UML Packages 154
6.11 Tips 155
6.12 Summary on Class Relationships 156
7 Collections and Data Structures 165
7.1 Root Interface - Collection 165
7.2 *Iterator* and *ListIterator* 167
7.3 *List* Interface 169
7.4 *Set* and *SortedSet* Interfaces 170
7.5 *Map* and *SortedMap* Interfaces 173
7.6 *Collections* Class 174
8 Files and Streams 180
8.1 Hierarchy of Streams in Java 180
8.2 Data Streams 181

8.3 Print Streams 183
8.4 Buffered Streams 184
8.5 Serialization 185
8.6 Sequential and Random Access Files 188
8.7 The File Class 189
8.8 *StringTokenizer* 190
9 Exceptions 196
9.1 Exceptions Hierarchy 196
9.2 Checked and Unchecked Exceptions 197
9.3 Try – Catch Block 198
9.4 Claiming Exceptions 199
9.5 Throwing Exception 200
10 Recent Advances in Component Software – Does Scala Beat Java? .. 208
10.1 From Deductive Systems to Programs 209
10.2 Polymorphism and Every Day Programming 211
10.3 Mixing Approaches to Encapsulation 217
10.4 Have Object-Oriented and Functional Languages Converged? 223
Conclusion 224
Acknowledgements 226
References 227

Introduction

Object Technology has been in development for over forty years. It is now embedded in such diverse areas as requirements engineering, software architecture, analysis, design, programming, testing, deployment and maintenance. The most widely used modern programming languages C++, Java and C#.Net all embrace an object oriented approach. This tutorial examines the application of the object oriented paradigm to programming. The fundamental concepts of object oriented programming are explained using the Java programming language. So, the main objective of this book is to develop an understanding of the key principles underpinning object oriented programming and apply object-based approaches using Java programming language.

Having studied the tutorial student will obtain the knowledge of fundamental principles of Object-Oriented programming. In particular, he will become familiar with the concepts of classes, objects, interfaces, inheritance, nested and abstract classes, polymorphism and data encapsulation. Special attention in this book is paid to UML (Unified Modeling Language). Moreover, the final sections of the book provide information concerning the basic Java constructs – java collections (sets, lists, maps), exceptions, input / output, java exceptions, etc.

The tutorial is mostly targeted on students studying the object-oriented programming as well as on everyone who wishes to master OOP principles. In this book it is assumed that a reader has an understanding of the basic programming constructs, like variables, repetitions, conditions, etc.

Don't resort to memorization. Understanding is the only key to being object-oriented. Spend some time considering how you could take already-developed code and transform it into a properly structured system, using the OOP principles, like encapsulation, abstraction, inheritance, polymorphism. The key indicator of performance here is being able to identify objects for the system, their states and behaviors, being able to point out behaviors that can be encapsulated and inherited (identifying the hierarchical relationships).

One more issue to clarify straight at the beginning is system requirements for most of the programs you meet in this tutorial. So, in order to complete the exercises in this tutorial, you will need to install and set up a development environment consisting of:

- JDK 6 (or higher) from Sun/Oracle

- Eclipse IDE for Java Developers

The instructions for installation and set up are omitted in this book. vAll the software listed is free and can be downloaded from official sites along with the installation instructions.

Good luck!

1 Object-oriented Approach

"The object-oriented version of 'Spaghetti[1] code' is, of course, 'Lasagna code'. (Too many layers)."
Roberto Waltman

"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."
Donald Knuth

This section guides you through the essential concepts of object-oriented programming on the Java platform. Particularly, we will consider existing programming paradigms and specify the role of object-oriented approach among them. Furthermore we will look at how these object-oriented constructs are implemented in Java programming language.

1.1 Paradigm of Object-oriented Programming

In the world of programming there exist a number of paradigms and *object-oriented* programming is one of them. Other programming paradigms include the *imperative* programming paradigm (supported by languages such as Pascal or C), the *logic* programming paradigm (exemplified by Prolog), and the *functional* programming paradigm (represented by languages such as ML, Haskell, Lisp and recently born F# and Scala). For example, in functional programming paradigm we treat computation as the evaluation of mathematical functions.

Generally, programming languages can be divided into two types: imperative languages and declarative languages. Imperative style describes *how-to* knowledge, whereas declarative style is *what-is* knowledge. Therefore, a program is *declarative* if it describes *what something is like*, rather than *how to* create it. Obviously, this is a totally different approach from traditional imperative programming, which requires the programmer to specify the step-by-step recipe, or precise algorithm to be run. In other

[1] Spaghetti code represents an unorganized code with no clear structure, beginning or ending.

words, imperative programs make the algorithm explicit while leaving the goal implicit, whereas declarative programs make the goal explicit, but leave the algorithm implicit.

Object-oriented programming represents an attempt to make programs more closely model the way people think about and deal with the world (1). An imperative programmer who solves some problem must first identify a computing task to perform in order to solve the problem. Programming then consists of finding a sequence of instructions that will accomplish that task (1). In contrast, at the heart of object-oriented programming, instead of tasks we find objects – entities that have behaviors, hold information, and can cooperate with one another. So, programming consists of designing a set of objects representing real or abstract entities in the problem domain. This is supposed to make the design of the program more natural and thus easier to understand and develop.

Object-orientation provides a set of tools and methods that allows to build reliable, user friendly, maintainable, well documented, reusable software systems that fulfills the requirements of its users (2). Working with an object-oriented mentality opens the door to a new manner of programming, in which a software system is seen as a community of objects that cooperate with each other by passing messages in solving a problem.

Nowadays it is claimed that the problem-solving techniques used in object-oriented programming more closely model the way humans solve day-to-day problems. As a simple illustration for that, let's consider how we solve an everyday problem: suppose you wanted to order a cake to your mom for her birthday with an inscription "Happy Birthday, Mom!". To solve this problem you simply find a person who takes orders for cooking cakes, suppose his name is Aziz. You tell Aziz the type of cake to cook (e.g. honey cake), its weight (e.g. 4 kg), and all other necessary details (e.g. inscription is "Happy Birthday, Mom!"). You can be assured that the cake will be cooked. Now, let's examine the mechanisms used to solve your problem.

1. You first found an appropriate *agent* (Aziz, in this case) and you passed to this agent a *message* containing a request.
2. It is the *responsibility* of Aziz to satisfy the request.
3. There is some *method* (an algorithm or set of operations) used by Aziz to do this. However, you do not need to know the particular methods (cooking techniques) used to satisfy the request— such information is *hidden* from you.

Of course, you do not need to know the exact details, but on investigation you may find that Aziz (suppose he is a chief cook), while performing your request, passed several another messages to another

cookers in his team, who are his subordinates (say, for preparing a crème and dough). If we will think a bit more we can imagine that chief cook, besides interacting with customers who order cakes, has also interactions with wholesalers of flour, butter, chocolate and so on, who, in turn, has interactions with other people.

This small yet very useful example illustrates the potential power of an object-oriented approach and leads us to our first conceptual picture of object-oriented programming: an object-oriented program is structured as a *community* of interacting agents called *objects*. Each object has a role to play. Each object provides a *service* or performs an action used by other members of the community (3). Each object has an obvious role in the system, and the beauty of that is that when a change is required or a bug occurs, the object you need to modify is apparent. Since each object maintains control over the manipulation of its own attributes the resultant code is clear. Furthermore, very often we may face the problem when some object wrongly references a variable that does not pertain to it. Data hiding can help us to avoid such errors from taking place by ensuring the appropriate visibility among messaging objects.

From the example provided earlier we have learned that members of an object-oriented community make requests to each other by the way of transmission of a *message* to an agent (an *object)* responsible for the actions. The message contains the request for an action and any additional information (parameters) needed to carry out the request. In response to a message, the receiver will perform some *method* (or set of methods) to satisfy the request.

There are some important issues to highlight here:

- The client sending the request message doesn't need to know the means by which the request is carried out (*information hiding*).
- An important principle implicit in message passing is the idea of finding someone else to do the work i.e. reusing components that may have been written by someone else. Like in our example, chief cook delegates the responsibility to prepare a crème and dough to other cookers, although he is also able to do it.
- Client's requests for actions only indicate the desired outcome. The receivers are free to employ any methods and techniques for achieving it. This principle enables greater independence between objects.
- Agents have responsibilities that they are able to fulfill on request. The collection of responsibilities associated with an object is often called a *protocol* .

Clearly, breaking code into a series of objects that can message one another support the development of flexible architecture, and using OOP helps to avoid unmanageable, or Spaghetti code. Flexible architecture bring benefits in the sense that it is easier to modify it due to the fact that the objects that make up the application possess distinct boundaries, that greatly simplifies substituting among the objects you're working with. Architecture of the system can be modeled using UML[2], that depicts the main agents, their responsibilities and connection between them. UML is a subject of discussion of Chapter 6.

No doubt, imperative programming can work well if you're the sole developer on a project, since you're familiar with your code. However, when more programmers are involved, it can be burdensome for them to become familiar with one another's code and surfing through the thousands of lines of code to see where a modification needs to be done. With OOP, each behavior in the application is contained in a unique class, providing a more elegant way to view object collaborations. Because each unique class possesses a name, it's easy to track down.

Going deeper, let's consider the key OOP principles that are vital for helping to make the code more organized and flexible. Namely, they are *encapsulation*, *polymorphism*[3], *inheritance*, and *data hiding* (4). Let's take a deeper look at them.

The crucial idea that is carried out by the **encapsulation**[4] is that what you need to know per problem domain must be clearly separated from what you don't need to know, so you aren't bored with irrelevant information. This way, you can maintain your focus. As an illustration, in order to watch TV we don't need to know precisely (or even at all!) how it works. All information we need to possess is how to use it – that is, how to turn it on/off, switch channels, change loudness, etc. Meanwhile, the internal working mechanisms of TV are a black box for us.

Furthermore, it is obvious that creating boundaries among your system's objects allows for interchangeability among other behaviors with similar method parameters, method name, and return type. These three components of a method are the contracts that proper messaging requires and together are referred to as a signature. So, **polymorphism** assures that as long as the signatures between varied implementations remain the same,

[2] Unified Modeling Language (UML) – the standard representation used to build models for computer applications.
[3] Latin meaning for many faces, or forms; a process that enables interchangeability among objects having the same interface.
[4] Inheritance is the principle of separating and localizing behavior into an object

the behaviors can be swapped to achieve various results without having to modify much code.

The concept of **inheritance** is modeled in object-oriented languages, enabling developers to write code in the form of hierarchical relationships. As a result, programmers can encapsulate a collection of behaviors and attributes into a superclass, that can be later used when you create additional classes, which you can do by deriving them from the original class. Much like children who benefit from the possessions of their mother and father, so can objects benefit through inheritance. A child object in a hierarchy of objects is referred to as a *subclass*, or derived class and its parent is referred to as its *superclass*, or base class.

Data hiding is the act of concealing information from a user of the application and a possible problem domain. It helps to maintain proper encapsulation and is enforced by the use of access modifiers, like private, public, protected, package, which will be discussed later.

Now that we've covered the principles of OOP, let's focus on their implementation into an object-oriented language.

1.2 Java as an Object-oriented language

Object-oriented languages are intended to facilitate structuring code at high levels of abstraction. One of the crucial features of these languages is the ability to structure code at the level of *classes*. In Java language, all code occurs in a *method*, all methods belong to a class, all classes in turn belong to a *package*.

Java, as an Object-Oriented language, provides support for all object-oriented concepts discussed earlier, like objects and classes, inheritance, polymorphism, encapsulation, etc.

In simple words, *Encapsulation* can be defined as a protective barrier that prevents data from being randomly accessed by other code defined outside the class. It is reinforced by different visibility (or access) modifiers in Java, like *private*, *protected*, *public* and *package* (no modifier). The Table 1 provided below gives the information about them.

As it can be easily seen from the table, protected is the most restrictive modifier and protected is a slightly less restrictive than private. It is important to note that by default, behaviors and attributes have package modifier unless declared otherwise.

Access Modifier	Definition
private	Attributes and behaviors can be accessed only in the scope in which they are declared.
protected	Behaviors and attributes can only be used within the class that defined them and within its subclasses.
Package (no modifier)	Can be viewed by any class in the same package
Public	Fields and methods can be viewed by any class of any package.

Table 1. Access Modifiers in Java

Notice that the public methods are the access points to private class's fields from the outside world. Normally these methods are referred as getters (accessors) and setters (mutators). Therefore any class that wants to access the variables should access them through these getters and setters. Accessor and Mutator methods will be discussed later in the subsequent sections.

Inheritance in Java defines an *Is-a* relationship between a superclass and its subclasses. This means that an object of a subclass can be used wherever an object of the superclass can be used. This provides the benefit of building new classes from existing ones. As an illustration, consider a class *Cat* that inherit some properties from a general *Mammal* class. Here we find that the base class is the *Mammal* class and the subclass is the more specific *Cat* class. According to Java syntax, subclass must use the *extends* keyword in order to derive from a superclass.

```
class Mammal {

}
class Cat extends Mammal{

}
```

The subclass inherits members of the superclass and hence promotes code reuse. Moreover, the subclass itself can add its own new fields and methods. The *java.lang.Object* class is always at the top, it is the root ancestor of all java classes. By the way, Java doesn't have a direct support for multiple inheritance. This topic as well as other ones associated with inheritance are discussed further in deep details in Chapter 4.

Polymorphism is an important object-oriented concept and is widely used in Java and other state-of-the-art programming languages. Java has an

excellent support for polymorphism in terms of interface, abstract class, method overloading and method overriding. you should also not forget that Inheritance also gives you ability to implement polymorphism by providing you substitution capability where a Base class can hold reference of derived class.

Method overriding allows the programmer to execute a method based on a particular object at run-time instead of declared type during coding. This is referred to as a *dynamic binding*.

As a demonstration for that, consider a simple example containing 3 hierarchically connected classes – a superclass Person and two deriving classes Student and Employee. Notice that all classes have a method names showDescription.

```
class Person{
        ...
        public void showDescription(){
                return "a person with name "+name;
        }
}
class Student{
        ...
        public void showDescription(){
                return "a student with id "+id;
        }
}
class Employee{
        ...
        public void showDescription(){
                return "an employee with salary "+salary;
        }
}
```

But how can you use that? Now this structure allows you to use super type in method argument that will give you leverage to pass any implementation while invoking method. For example:

```
public void showDescription(Person studentOrEmployee){
          studentOrEmployee.showDescription();
      }
```

Polymorphism allows you to pass as a parameter to a method above all classes extending Person, so you can pass Student or Employee. As a result,

you might have a code that will work even for classes not implemented yet, the only requirement is that they must derive from Person.

Note that in this section we've just introduced the notion of polymorphism. Please, read more about it in Section X.

KEY CONCEPTS YOU NEED TO GRASP FROM THE SECTION

Message *Responsibility* *Agent* *Object*
Access Modifiers *Encapsulation* *Polymorphism*
Inheritance *Object-orientation* *UML* *public* *private*
protected *Imperative programming* *Java*

TESTS

1. Which of these access modifiers are *not* accessible in other packages?

a) private b) public

c) protected d) no access modifier

2. … allows you to derive new classes from existing classes.

a) inheritance b) abstraction

c) encapsulation d) generalization

3. The basic notion of object-oriented programming is…

a) method b) field c) inheritance d) object

4. The term used to describe the internal representation of an object that is hidden from view outside the object's definition?

a) polymorphism b) composition

c) encapsulation d) inheritance

5. When an object has many forms, it is …

a) inherited b) scalable

c) encapsulated d) polymorphic

6. Objects from different classes communicate with each through…

a) inheritance b) polymorphism

c) messages d) responsibilities

7. Which Java keyword is used to specify inheritance?

a) extends b) public c) implements d) inherits

8. Which class is a root of all classes in Java?

a) Interface b) Class

c) Object d) Collection

9. How can you call a class that is derived from another class?

a) superclass b) nested

c) subclass d) cloneable

10. Which of the languages presented below are imperative?

a) Lisp b) C c) Java d) Pascal

PROBLEMS

1. Encapsulation in OOP is used to conceal data from the user and mediate changing of mutable states. Imagine a class *Person* with private field *age*. Think, how to make it impossible to set *age* less than 0 or more than 90.

2. In the section studied we claimed that OOP resembles the way we solve problems in our real life. As a proof we provided a simple illustration for how we solve an everyday problem – ordering a cake for the mom for her birthday with an inscription "Happy Birthday, Mom!". Identify objects (or agents) in this system and their responsibilities.

3. When we were talking about inheritance we demonstrated it on the example of *Mammal* and *Cat* (notice that *Mammal* itself can be a child class of *Animal*). Specify similar hierarchical relationships, but not in the animal world, but in any organization you like (KBTU can be a good choice to consider).

2 Introduction to Java programming language

"Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program."
Linus Torvalds

"Java is the most distressing thing to happen to computing since MS-DOS."
Alan Kay

"If you learn to program in Java, you'll never be without a job!"
Patricia Seybold

This chapter intends to introduce some of the basic concepts and techniques involved in Java program design and development, including fundamental Java syntax and its use.

Nowadays Java technology is used to develop applications for a wide range of environments, from consumer devices to heterogeneous enterprise systems (3). In previous section we have already described the programming paradigm employed by Java. In this section, you'll learn the Java syntax you are most likely to meet professionally and standard Java programming idioms you can use to build the robust applications.

Keep in mind that programming is not simply a question of typing code. Instead, it involves a substantial amount of planning and careful designing. Poorly designed program hardly ever works correctly. Therefore remember that careful design of the program must precede coding. This is particularly true for object-oriented programs.

Another common mistake you can make is to type the entire program code and then compile and run it. More often than not this will result in dozens of errors that can be difficult and so time-consuming to fix. The solution for that is to use the principle of *stepwise refinement* [5]. For example, you could write the code for a single method or for some small logical part and test that piece of code before moving on to another part of the program. By doing this, small errors will be caught before moving on to

[5] Stepwise refinement means that the program is coded in small stages, and after each stage the code is compiled and tested

the next stage. There is a famous proverb - *a stitch in time saves nine*. This means that a timely effort will prevent more work later.

Due to the fact that originally Java had a role of a language for programming microprocessors embedded in consumer appliances (5), it has been designed with a number of interesting features described in subsection 2.1.

2.1 Why java ? (Java features)

Java is a relatively young general-purpose object-oriented programming language. It is of interest to note that initially it was named "Oak" after a tree outside the office of its developer, James Goslin. Subsequently, it was renamed to "Green" and then, finally, to "Java" from *Java coffee*, which is said to be consumed in large quantities by the java's creators (6).

It was originally designed by Sun Microsystems in 1991 as a language for embedding programs into electronic consumer devices, such as microwave ovens and home security systems (4). Nevertheless, the fantastic popularity of the World Wide Web led Sun to transform Java to a language for writing embedded programs into web-based applications.

Besides web-applications, Java also generated tremendous interest in the business communities, where it has proved to be commercially highly beneficial. Moreover, Java language is also a good choice for writing distributed software and providing services to employees and customers on private corporate networks (intranets) (7).

Now let's consider some of the central Java's features that make it extremely useful and competitive language:

- **Java is object-oriented**. As we have already learned in Section 1, object-oriented languages divide programs into separate modules, called objects, that encapsulate the program's data and operations. Unlike the C++ language, Java was designed from scratch as an object-oriented language.
- **Java is robust**. That means that errors in Java programs don't cause system crashes as often as errors in other programming languages. Certain features of the language enable many potential errors to be detected before a program is run.

- **Java is portable (platform independent)**. Java's trademark is *"Write once, run anywhere."*[6] This means that a Java program can be run without changes on virtually any platform (e.g. Windows and Linux). This is not true for other high-level programming languages.
- **Java is reliable.** This feature is supported by the extensive compile-time and runtime error checking (e.g. bytecode cheking). In addition, there are no pointers but real arrays. Therefore, memory corruptions or unauthorized memory accesses are impossible. Finally, Java supports dynamic memory management via automatic garbage collection which is intended to track objects usage over time (8).
- **Java is interpreted.** During compilation phase, java compiler generates byte-codes, not native machine code. Obviously, the compiled byte-codes are platform-independent. Then, at runtime java bytecodes are translated on the fly to machine readable instructions (Java Virtual Machine, discussed further in subsection 2.2) .
- **Java is dynamic.** Java is designed to adapt to evolving environment. Particularly, libraries can freely add new methods and variables. Moreover, interfaces provide flexibility and reusability in code by specifying a set of methods an object can perform, without specifying how these methods should be implemented.
- **Java has a rich API**. Java comes with an extensive collection of rich code libraries—software that has been designed to be used directly for particular types of applications.
- **Java is secure**. Designed to be used in networking environments, Java contains features that protect against untrusted code—code that might introduce a virus or corrupt your system in some way. In particular, access restrictions are forced (private, public) and array access bounds are checked (in contrast to C++).
- **Java is multithreaded.** This generally means that multiple concurrent threads of executions can run simultaneously
- **Java is simple.** The language constructs are intuitive and straightforward. Besides, Java libraries are easily accessible and extensible.

Despite the list of attractive Java features presented above, probably the best reason for choosing Java is its potential for bringing fun and

[6] Programmers joke on that principle*: "Java is write once, run away".*

excitement into learning how to program especially in object-oriented style. The simplicity of Java's design bring such accomplishments within reach of the most novice programmers (4).

2.2 Java Virtual Machine (JVM)

The JVM is a piece of software which is written specifically for a particular platform. The Java Runtime Environment (JRE) includes the JVM, code libraries, and components that are necessary for running programs written in the Java language.

Firstly, Java program code is compiled and transformed into an intermediate representation called Java bytecode (i.e. sequence of zeros and ones), instead of directly to platform-specific machine code.

Bytecode instructions are designed to be interpreted by a JVM written specifically for the host hardware. In adding this level of abstraction, the Java compiler differs from other language compilers, which write out instructions suitable for the CPU chipset the program will run on. So, at run time, the JVM reads and interprets .class files and executes the program's instructions on the native hardware platform for which the JVM was written (3).

The JVM is the heart of the Java language's *"write-once, run-anywhere"* principle. So, your code can run on any chipset for which a suitable JVM implementation is available (7). JVMs are available for major platforms like Linux and Windows.

A major benefit of using bytecode is portability (discussed earlier). This ensures that programs written in Java run similarly on any supported OS platform. Nevertheless, interpreted programs almost always run more slowly than programs compiled to native executables. Using a VM is slower than compiling to native instructions. Actually, JIT (Just-in-time) compilers convert Java bytecode to machine language during runtime. In addition, safety/security slow things down (all array accesses require bounds check, many I/O operations require security checks). These are the reasons why programs written in Java have a reputation for being slower that the ones written in C++.

2.3 Compiling and running Java program

Now it's time to have a look at how to compile and launch a Java program (illustrated in Figure 1). There are 2 commands for these purposes – javac and java:

- **Javac -** the Java compiler, it reads source code and generates bytecode, as discussed above. We normally write source code in .java files and then compile them. The compiler checks the code against the language's syntax rules, then writes out bytecodes in .class files.
- **Java** - the Java interpreter, it actually runs bytecode. Remember that you tell the interpreter a class to run, not a file to run!

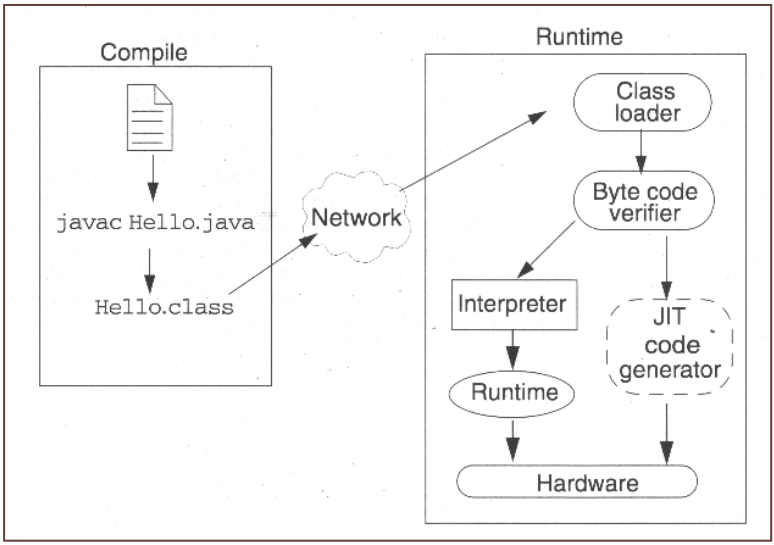


Figure 1. Compile and Runtime operations

It needs to be emphasized that Eclipse IDE which you are advised to use for this course, performs compilation automatically – every time you save the program (by Ctrl+S) or add some code. In contrast, launch of the program must be done manually.

It is important to emphasize that when your Java program creates an object instance at run time, the JVM automatically allocates heap memory

for that object (4). Furthermore, the Java garbage collector[7] which runs in the background, keeps track of the objects that application no longer needs and reclaims memory from them. This approach to memory handling is called implicit, or dynamic memory management because it doesn't require a programmer to write any memory-handling code. Garbage collection is one of the essential features of Java platform performance (5).

2.4 Java Syntax

In this section we will acquaint you with some of the key elements of the Java syntax by describing the details of a small program. We will look at how a program is organized and what the various parts do. Our main intention is to introduce the basic language elements, many of which will be explained deeper in later sections.

Structurally, the Java language starts with packages. A package is the Java language's namespace mechanism. Within packages are classes, and within classes are methods, variables, constants, and so on.

For our example we will look at a Java version of a traditional HelloWorld program—"traditional" because practically every introductory programming text begins with it. When it is run, the HelloWorld program just displays the greeting "Hello World!" on the console.

```
public class HelloWorld {  // class header
        private String greeting = "Hello world";
        public void greet(){  // method definition
                /* output
                statement*/
                 System.out.println(greeting);
        }
        public static void main(String[] args) {
                // declare and create an object
                HelloWorld hw = new HelloWorld();
                hw.greet();  // method call
        }
}
```

[7] Garbage collection is the process of automatically finding memory chunks that are no longer used by the program ("garbage"), and making them available again. In contrast to manual deallocation that is used by many languages, (e. g. C and C++) Java automates this error-prone process (6).

Note that when you save your public class declaration in a file, the file name must be the class name followed by the ".java" extension. Therefore, for our application, the file name is HelloWorld.java.

Furthermore, *System.out* is known as the standard output object. This method just prints passed text without moving output cursor to a new line.

Now let's discuss some basic but still important Java constructs based on the example provided above (HelloWorld.java).

- **Comments**

The first thing you probably noticed in the HelloWorld program is the use of comments. As it is well knows, comment is a non-executable portion of a program mainly used to document the program (3). So, since comments are not executable instructions they are just ignored by the compiler. Their only goal is to make the program easier for thedeveloper to read and understand.

The HelloWorld program contains examples of two types of Java comments:

1. Multiline comment. Any text contained within /* and */ is considered a comment. As you can see in HelloWorld, this kind of comment can span over several lines.

2. Single-line comment. A second type of comment is any text that follows double slashes " // " on a line. This type of comment can't be extended beyond a single line (4). A single-line comment must be contained on one line, although you can use adjacent single-line comments to form a block.

It is a good style to include a summary at the beginning of the program in order to explain what the program does, its key features, its supporting data structures, and any unique techniques it uses (8).

- **Keywords**

Like any programming language, the Java language contains certain words – so called keywords - that the compiler recognizes as special, and as such you are not allowed to use them for naming of your own variables.

The Java language contains 48 predefined keywords presented in Table 2 below (5). These are words that have special meaning in the language and whose use is reserved. For instance, the keywords used in our *HelloWorld* program are: class, private, public, static, and void.

abstract	default	goto	package	this
boolean	do	if	private	throw
break	double	implements	protected	throws
byte	enum	import	public	transient
case	elses	instanceof	return	try
catch	extend	int	short	void
char	final	interface	static	volatile
class	finally	long	super	while
const	float	native	switch	
continue	for	new	synchronized	

Table 2. Java keywords

Obviously, because their use is restricted, keywords cannot be used as the names of methods, variables, or classes.

- **Naming conventions**

In addition to the syntax rules governing identifiers , Java has certain naming conventions in making up names for classes, variables, and methods. This helps to unify the programming style and identify easily whether an entity is a method, class or variable (2).

Names in Java are case sensitive. Therefore, two different identifiers may contain the same letters in the same order. As an illustration for that, *currentValue* and *CurrentValue* are two different identifiers.

Major Java naming conventions are summarized and presented in Table 3 below.

	Rule	**Examples**
Class names	Capitalize the first letter of each word in the name	HelloWorld ArrayList Person
Variable and method names	Begin with a lowercase letter but also use capital letters to distinguish the words in the name	getName() setMessage() hello() myCounter radius
Constants	Capitalize all letters	PI WIDTH MAX

Table 3. Naming conventions in Java

Looking at the *HelloWorld* example program, we see that all the variables, methods and class names follow the naming conventions.

As it was mentioned, the significant advantage of these conventions is that it is easy to distinguish the different elements in a program—classes, methods, variables—just by how they are written. Sticking to these guidelines will ensure that your code is more accessible to other developers following the same conventions (6).

Another important convention that you always need to keep in mind is to choose meaningful and descriptive names. Even if you rely on your memory and you are confident that a month later you will be able to remember that variable you called *var1* stores the number of books in a library, and *var2* stores number of sections in a library, for example, it is an extremely bad programming style. In real life developers work in teams and they constantly need to cooperate with each other. So, your code becomes shared and might be needed to be modified by another programmer, who, surely, won't possess such an extraordinary intuition to guess what you meant by *var1* and *var2*.

- **Variables**

As it is known, a variable is a location in the memory where a value can be stored in order to be used later in a program. Clearly, variables must be declared with a name and a type before they can be used.

A *declaration statement* is a statement that declares a variable of a particular type. In turn, an *assignment (or initialization) statement* is a statement that stores (assigns) a value in a variable. Variable declaration statement specifies the name and type of a variable, for example:

```
String greeting;
int count;
```

Variable assignment statement initializes the variable with a special value, for instance:

```
greeting = "Hello world";
count = 883;
```

In our *HelloWorld* program we combined declaration and assignment into one statement, like:

```
String greeting = "Hello world";
```

Finally, in order to declare a constant variable, use *final* keyword:

```
final double PI = 3.14159;
```

- **Primitive and reference types**

There are two basic data types used in Java, namely, primitive types and reference types.

Primitive types are the *boolean* type and the numeric types. The numeric types are the integral types - *byte, short, int, long, and char*, and the floating-point types - *float and double*. The reference types are *class* types, *interface* types, and *array* types. There is also a special null type.

```
class Point { int[] metrics; }

interface Move {
       void move(int deltax, int deltay);
                    }
```

Values of primitive types are stored directly on the stack rather than on the heap, as commonly true for objects. This was a conscious decision by Java's designers for performance reasons (1). The value of a variable of primitive type can be changed only by assignment operations on that variable. ALWAYS use double equal ("==") operator to compare for equality.

As already mentioned, the reference values are pointers to either array, interface or class objects, and a special null reference, which refers to no object. Because reference type variable store just a pointer on the stack, and the value itself is stored on a heap, it is impossible to change its value by "==" operator. Therefore, in order to compare two objects for equality, use equals() method (discussed in the next section).

Table 4 below summarizes the key information about the primitive and reference types.

	Primitive types	Reference types
Variable contains	value	reference (pointer)
Stored on	stack	heap
Initialization	0, false, '\0'	null
Assignment	copies the value	copies the reference

Table 4. Primitive and Reference types

An important issue to pay attention to is assignment operation for reference types. Because we store just pointer to a reference variable on a stack, not the value itself, when you try to assign one reference variable to the other you actually copy this address (pointer), not the value. It means that now you have two variables pointing to one location in memory. Therefore, when you change one of them, the value will change for the other variable too. Let's think of a real-life example for that. Suppose you and your friend have a common TV and two remote controls for it. So, a pointer is like a remote control in this example and value is like TV. Obviously, when your friend switches the channel, you both see changes happening on TV, and vice versa.

On Figure 2 below you can see the illustration for assignment operation for primitive (on the left) and reference (on the right) type variables.

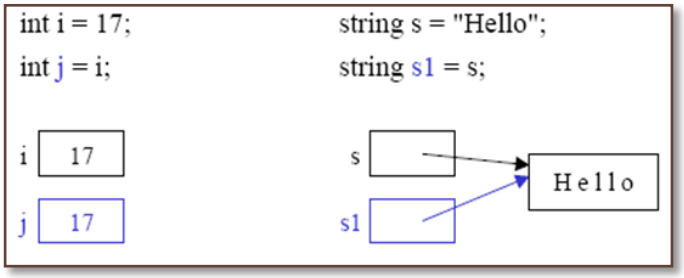


Figure 2. Assignment operation for primitive and reference types

- **Methods**

We know that method is a collection of statements that are grouped together to perform an operation. Let's look at a concrete example of method implementation in java:

```
public static int max(int num1, int num2)
      {
        if (num1 > num2)
          return num1;
        else
          return num2;
      }
```

The method above returns the maximum of two values provided. The

Figure 3 below can help you to revise information you know about methods and identify main method parts.

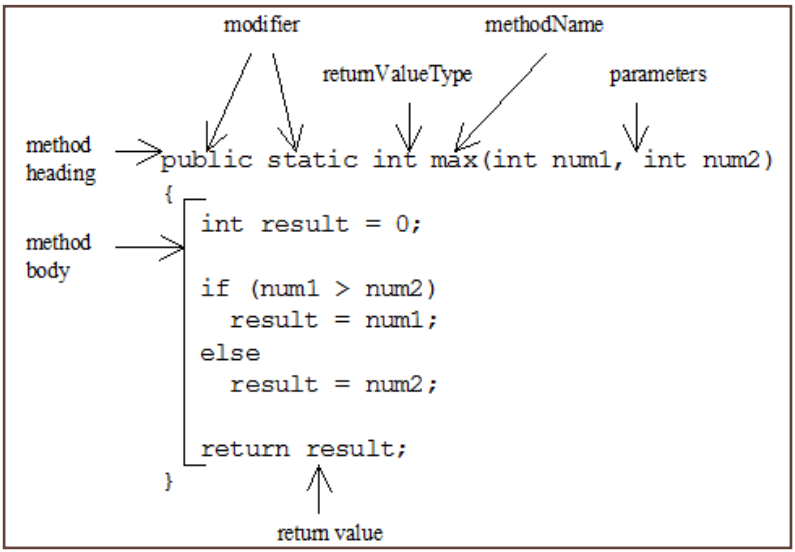


Figure 3. Method structure

Java supports methods overloading, and can distinguish between methods with different signatures. Therefore, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. Nevertheless, the return value is not seen as part of the *signature* of the method (6). It means that you cannot overload a method by changing only the return value. So, for example, the following code will fail to compile, since we have duplicate method greet (same signatures):

```
public void greet() {
        System.out.println("Hello!");
    }

public String greet() {
        return "Hello!";
    }
```

Now let's overload the max method defined earlier. The defined max method accepts two integers and finds maximum of them. But what if we

want to find maximum of two doubles? For this purpose, novice programmers define new methods with names like max2, maxx, _max, etc. But if we will employ the overloading concept, we can write directly:

```
double max(double num1, double num2)
        {
          if (num1 > num2)
            return num1;
          else
            return num2;
        }
```

Finally, it is important to pay attention to the concept of method abstraction. Actually, you can think of the method body as a black box that contains the detailed implementation for the method (3). The scheme for that is provided in Figure 4 below. As a simple illustration, in our real life you don't need to know how the TV is built in order to watch it. You just know about the inputs (different channels' numbers) and output (channel on the TV), that is all you need to carry about. Everything else is a black box for you. In much the same way, any method from Java API has input parameters, output value and in order to use it, you are not obliged to know how it is implemented.

.

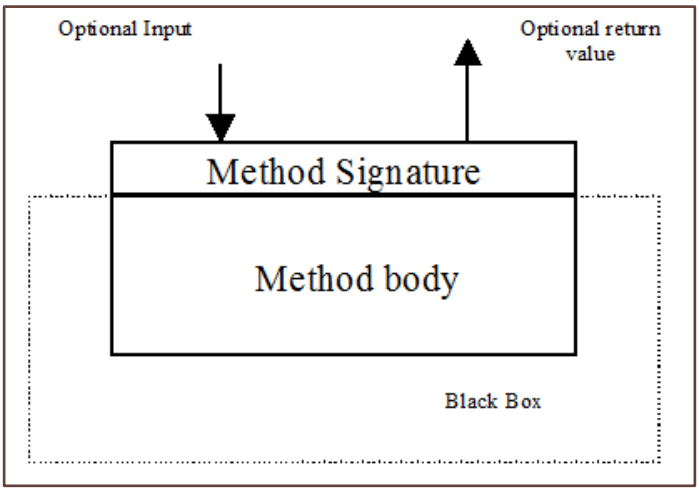


Figure 4. Method abstraction

Finally, let's look at one more simple Java program before going further to classes and objects.

The program provided below reads two numbers from command line and outputs their sum. In order to get the input from user we need to create an object (i.e. instance) of Scanner class referring to System.in. In order to use this class, we need to import it (first line). You can also import Scanner class together with other classes from java.util package, you will just need to replace Scanner by '*', that asterisk means "all". By the way, Scanner can also refer to a file.

Furthermore, we use nextInt() method to read the integers inputted. Eventually, we sum the integers and print the output to the console. You can check Java API for other methods the Scanner class has.

```
import java.util.Scanner;
public class Addition {
      public static void main(String[] args) {
        // create Scanner to get input from console
            Scanner in = new Scanner(System.in);
            int num1;
            int num2;
            int sum;
            // prompt user
            System.out.println("Enter first integer");
            num1 = in.nextInt(); // read 1st number
            System.out.println("Enter second integer");
            num2 = in.nextInt(); // read 2nd number
            sum = num1+num2; // add numbers
            System.out.println("The sum is: "+sum);
      }
}
```

This program can be written in much compressed way. Let's look at one of them:

```
public class Addition {
  public static void main(String[] args) {
   Scanner in = new Scanner(System.in);
   System.out.println("Enter two integers");
   System.out.println("The sum is: "+
                      (in.nextInt()+in.nextInt()));
     }
}
```

- **Java API**

Java Application Programming Interface (API) — (also called the Javadoc) – the official online documentation that most Java developers constantly reference. A great strength of Java is its rich set of predefined classes that programmers can reuse rather than "reinventing the wheel." In our example (Addition program) we use Java's predefined *Scanner* class from package java.util.

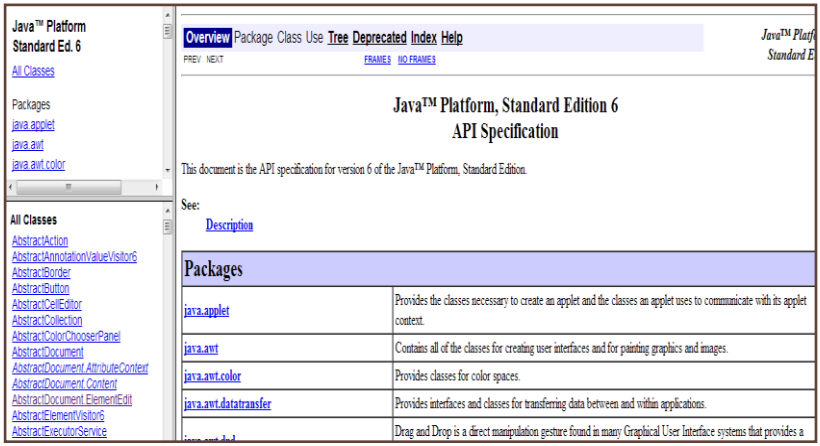


Figure 5. Java API site

Now let's look at the structure of the documentation site (see Figure 5) (9). By default, you can see three frames in the Javadoc. The top-left frame shows all of the packages in the documentation, and underneath there are classes in each package. So, when you choose some package its classes are displayed there.

The main frame (situated on the right) shows details for the currently selected package or class. For example, if you select the java.util package in the top-left frame and then select the LinkedList class listed below it, in the right-hand frame you will see details about LinkedList, including a description of what it does, how to use it, and its methods (3). A Web-based version of this documentation can be found at

http://docs.oracle.com/javase/6/docs/api/

Also, you can download this documentation to your own computer in order to have access to it when you are offline.

As we have learned in this chapter, Java, like any programming language, has its own structure, syntax rules, and programming paradigm.

As we have known, the Java language's programming paradigm is based on the concept of object-oriented programming (OOP).

KEY CONCEPTS YOU NEED TO GRASP FROM THE SECTION

Stepwise refinement *Robustness* *Portability* *JVM*
Naming conventions *bytecode* *JIT* *Interpretation*
API *Eclipse IDE* *Primitive type* *Reference type*
Overloading *Methods* *Variables* *Scanner*

TESTS

1. Which of the following are reserved words in Java?

1. **run**
2. **import**
3. **function**
4. **implements**

a) 1 and 2 b) 2 and 3 c) 3 and 4 d) 2 and 4

2. What will be the output?

```
String s1 = new String("Hello");
String s2 = new String("Hello");
if (s1 == s2) System.out.println("True");
else System.out.println("False");
```

a) True b) False

c) True False d) The code will fail to compile

3. What will be the value of variable *c*?

```
int a=8;
int b=3;
int c=0;
c = ++b+a+b++;
```

a) 11 b)15 c)16 d) 0

4. Guess what the output of the program is going to be.

```
for (int i=0; i<5; i++)
        if(i==2) break;
        else System.out.println(i);
```

a) 0 1 2 3 4 5 b) 0 1 3 4 5 c) 0 1 3 4 d) 0 1

5. What will be the output of the following program?

```
public class Example {
      static String s;
      public static void main(String[] args) {
      System.out.println(">>" + s + "<<");
                        }
}
```

a) >>null<< b) >><< c) >>string s<< d) >>s<<

6. Java applications are typically compiled to bytecode that can run on:

a) Java Runtime Environment b) JDK

c) Java Virtual Machine d) any program.

7. Which of the following refers to dynamic memory management?

a) JVM b) Just-in-Time compiler

c) Input/Output operations d) garbage collection

8. Java compiler generate

a) machine code b) compiler code

c) bytecode d) error code

9. What is variable?

a) components that are necessary for running programs written in the Java language

b) location in the computer's memory where a value can be stored for later use in a program

c) pointers to some objects

d) class instance or an array.

10. How can you compare two variables of reference type for equality?

a) compare() method b) equals() method

c) using "==" operator d) using "=" operator

11. Choose the correct version of methods overloading:

a) `int method_name(int ) , float method_name(int)`

b) `int method_name(double x ,double y) , int method_name (int x)`

c) `Sring method1(String a) , int method1(int s)`

12. What is the result of the following code:

```
int x=0;
switch(x)
{
case 1: System.out.println("One");
case 0: System.out.print ("Zero");
case 2: System.out.println ("Two");
}
```

a) One b) Zero c) Two d) ZeroTwo

13. The visibility of Java modifiers increases in which of the following order?

a) private, package, protected, and public.

b) private, protected, package, and public.

c) package, private, protected, and public.

d) package, protected, private, and public.

14.Classes can be grouped in a collection called :

a) group b) package c) collection d) folder

15. Choose correct importation of all classes from java.io package:

a) import java.io.*; b) import java.io*;

c) import java.io.all; d) include java.io.*;

PROBLEMS

1. Which advantages do you see in platform independence?

2. In our short version of Addition program we wrote:

```
System.out.println("The sum is: "+(in.nextInt()+in.nextInt()));
```

Will the program still work correctly if we will write instead:

```
System.out.println("The sum is: "+in.nextInt()+in.nextInt());
```

If no, why? What will be the output for numbers 7 and 8, for example?

3. Guess what the output of the program is going to be.

```
for (int i=0; i<5; i++)
        if(i==2) continue;
        else System.out.println(i);
```

What about this one:

```
int i=0;
while(i<5){
      if(i==2) continue;
      else System.out.println(i++);
       }
```

4. There is an array of many elements, but the actual data is stored in the first n elements, after which there are just zeros. How to find

the last element? Write a special method that does this job. Think carefully of input and output parameters

5. Write a method to reverse elements of the array without using any special methods from Java API and extra memory?

6. Consider 3 methods below.
Which pairs out of the 3 methods represent method overloading? Give your own plausible example.

```
A:
public void setPrice(double newPrice){
        price= newPrice;
}

B:
public void setPrice (Pencil p){
        price = p.getPrice();
}

C:
public double setPrice (Pencil p){
        price = p.getPrice();
}
```

7. Implement the method that returns a string of all capital letters from an original string. For instance, for string "HeLLo, ALiCe" return "HLLALC". You are not allowed to use Character.isUpperCase() method or regular expressions.

8. Develop a method that sums up all digits of a multiple-digit integer. Use only basic Java arithmetics. Don't use string methods.

9. Write a method that converts given binary number (in a String variable) into an integer. Don't use any special conversion methods for that.

10. Implement your own split() method, that splits inputted string into an array of string by some separator(s).

11. Create the *calculateDuplicates* method which counts how many times you encounter an element in an array and returns an array of pairs with element and its count.

Example:

[1,1,2,3,1,2] -> [[1,3], [2,2], [3,1]]

12. Implement the *numberOfEvenNumbers* method which counts how many times you encounter an even element in an array and returns an array of pairs with even element and its count.

Example:

[4,6,1,1,2,3,1,2,2,2,4,4] -> [[0,0], [2,4], [4,3], [6,1]]

13. Create a program that calculates an area, perimeter, and the length of diagonal of a square with a side *a*, that your program must read from user input using Scanner class.

14. Write a program to find the roots of quadratic equation. Use Scanner in order to get *a,b,c* parameters from user input. Do not forget to show error message in case D is negative. Moreover, be sure that some computations are not carried out twice, that wastes running time.

15. Write a program that displays a grade (A, A-, B+…) according to a number that a user enters on the console screen. It's up to you, which conditional statements to use. (Use KBTU grading system!).

3 Fundamentals of Objects and Classes

We introduced the basic terminology and concepts of object-oriented programming in Chapter 1. Later on in Chapter 2 we passed through the main principles driving programming in Java. Now you are well-prepared to explore in details what an object is, how objects are grouped into classes, how classes are related to each other, and how objects use messages to interact with and communicate with each other. Moreover, you will learn how to use predefined classes from Java API and how to write your own classes, create objects and add behavior to them.

3.1 What is a class?

The standard Java library supplies several thousand classes for such diverse purposes as user interface design, dates and calendars, and network programming (2). Nevertheless, in order to describe the objects of the problem domains of your applications you still have to create your own classes.

In object-oriented world it's possible to have many objects of the same kind that share characteristics. For instance, students, rectangles, employees, books, and so on. A class is a software blueprint for such objects – it is used to create objects. It defines the variables and the methods common to all objects of a certain kind.

So, the class can have two types of members, namely fields and methods. *Fields* are data variables which determine the state of the class or an object. In turn, *methods* are executable code of the class built from statements. They allow us to change the state of an object or access the value of the data member. So, variables reflect the state of the object, whereas methods determine behavior of the object.

After you've created the class, you can create any number of objects from that class. A class is a kind of factory for constructing objects (1). When you construct an object from a class, it means you have created an instance of the class.

As a simple illustration, let's consider two examples - classes representing a telephone directory entry and a pencil. It might be defined like:

```
public class Entry {
        private String name; // name as characters
        private String number; //phone number
}
```

Really, all telephone entries contain the name and a corresponding number. These variables were reflected in a class Entry.

Furthermore, the Pensil.java class might look like:

```
class Pencil {
        public String color = "red";
        public int length;
        public double diameter;
        public void setColor (String newColor){
                color = newColor;
        }
}
```

Using the Pencil example provided above let's look at how the fields need to be declared, for example, the one representing the color of the Pencil:

```
public String color = "red";
```

So it is clearly seen that to define a field you first put the access modifier, then a type name followed by the field name. Note that field declarations can be preceded by different modifiers, namely:

- access control modifiers (public, private, package, protected)
- static
- final

Let't consider each of them in details.

- **Access Modifiers**

Access modifiers were already presented in previous sections. Nevertheless, let's look at a summarizing table (Y stands for "Yes", N for "No"):

Access Modifiers	*Same Class*	*Same Package*	*Subclass*	*Other packages*
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no access modifier	Y	Y	N	N
private	Y	N	N	N

Table 5. Access modifiers

From table above we can conclude, that, for example, private members are accessible only in the class itself, while public ones are seen anywhere the class is accessible. As an example for private access modifier let's enrich our Pencil class defined earlier and add a private variable price:

```
public class Pencil {
	public String color = "red";
	public int length;
	public double diameter;
	private double price;
	public static long numberOfPencils = 0;
	public void setPrice (float newPrice) {
		price = newPrice;
	}
}
```

You might have noticed that, besides price we added a new variable *numberOfPencils* (discussed later). Now, having a private variable *price* we cannot change its value directly, using the field *price*:

```
public class CreatePencil {
	public static void main (String args[]){
		Pencil p1 = new Pencil();
		p1.price = 0.5f;
	}
}
```

This code will result in a following error during compilation:

```
%> javac Pencil.java
%> javac CreatePencil.java
CreatePencil.java:4: price has private access in Pencil
             p1.price = 0.5f;
                ^
```

This happens due to the protection level put on variable *price*, which is *private.* The problem can be solved by changing the visibility of the *price* variable (making it *public*, for example), which is not a very good idea. Another solution is to use mutator - *setPrice* method.

```
Pencil p1 = new Pencil();
p1.setPrice(0.5f);
```

This technique refers to Encapsulation (discussed further in this chapter).

- ***Static* keyword**

The rule of thumb you need to remember is that only one copy of the static field exists and it is shared by all objects of the class. The variable having a static modifier can be accessed directly in the class itself. Access from outside of the class must be preceded by the class name as follows:

```
System.out.println(Pencil.numberOfPencils);
```

Note that outside the class non-static fields must be accessed through an object reference, not the class name.

Read more on that in a subsequent section *Class members and instance members*.

- ***Final* keyword**

Final keyword indicates the constant variable - once initialized, its value cannot be changed. So, *final* is often used to define named constants.

It is important to understand that static final fields must be initialized when the class is initialized, while non-static final fields must be initialized when an object of the class is constructed.

Note that if a variable was not initialized, then a default initial value is assigned depending on its type. In particular, variables of numerical types (*byte, short, int, long, double*) have a default value of *0*, boolean – *false*, object reference – *null* value (5).

3.2 What is an object?

Just as in the real world, an object is any thing whatsoever. An object can be a physical thing (e.g. Bus, Clock, ATM), or a mental thing (e.g. Event, Idea). It can be also a natural thing, such as an Animal, Student, Worker, etc. For example, the program managing an ATM would involve *BankAccount* and *Customer* objects. A chess program is likely to involve a *Board* object and *Piece* objects (4).

In the world of programming, object is an instance of a class. Any object has a class which defines its data and behavior..

Going back to the example from Chapter 1 (ordering a cake for the mom's birthday), we see that Aziz is an instance of a category or class of people i.e. Aziz is an instance of a class of cookers. The term cooker represents a class or category of all cookers. Aziz is an object or instance of a class. We interact with instances of a class but the class determines the behavior of instances. *We can tell a lot about how Aziz will behave by understanding how cookers behave*. That is the key point in understanding the concept of classes and objects. We know, for example, that Aziz, like all cookers, can cook cakes of different types and sizes.

Now let's make a correspondence between software objects and real-world objects. As we see, objects in our life share two characteristics: they all have state and behavior. For example, dogs have state (name, color, breed, hungry or not, …) and behavior (barking, running, wagging tail, …). Students have state (name, student id, courses they are registered for, gender, mode of payment,…) and behavior (take tests, attend courses, register for courses, write tests, party, …).

In OOP we create software objects that model real objects. Programming objects are modeled after real-world objects in the sense that they too have state and behavior. A software object maintains its state in one or more **variables** (fields) and implements its behavior with **methods** (some function associated with an object). Therefore we can say that an object is a software bundle of variables and related methods.

In a running program, there may be many instances of an object. For example, there may be many *Student* objects. Each of these objects will have their own instance variables and each object may have different values stored in their instance variables. For example, each *Student* object will have a different number stored in its *Id* variable (1).

Think of objects as floating around independently in the computer's memory. In fact, there is a special portion of memory called the heap where objects live (1). Instead of holding an object itself, a variable, which is stored in a stack, holds the information necessary to find the object in

memory. This information is called a reference or pointer to the object, which represents the address of the memory location where the object is stored. Program uses the reference in the variable to find the actual object.

An important point you need to keep in mind is that, declaring a variable does not actually creates an object. Remember that in Java, no variable can ever hold an object. A variable can only hold a reference to an object. Objects are actually created using an operator called new, which creates an object and returns a reference to that object.

As a simple illustration to notions described above, let's assume that we have a class Student with fields for *name (String), test1, test2, test3* (all of type *int)* and a method for calculating the average mark. So, declaration statement looks like:

```
Student s1, s2;
```

And now let's create the object itself by using the ***new*** keyword:

```
s1 = new Student()
s2 = new Student();
```

You need to understand that *s1* and *s2* store just a reference to the object. The statements above would create new objects which are instances of the class Student, and they would store references to these objects in the variables *s1* and *s2* respectively. Again, keep in mind that the value of the variable is a reference to the object, not the object itself! Therefore, when we copy value of one variable into the other one, we copy pointers, not the objects themselves:

```
s2 = s1;
```

The above statement will copy the reference value stored in **s1** into the variable **s2**. You can also store a null reference in a variable, like:

```
Student s3 = null;
```

Let's see how to set the values of some instance variables:

```
s1.name = "John Smith";
s2.name = "Mary Jones";
```

As it was already mentioned, other instance variables will have a default initial values of zero if they will not set manually. Figure 6

demonstrates what the situation in the computer memory is going to be after the computer executes these statements:

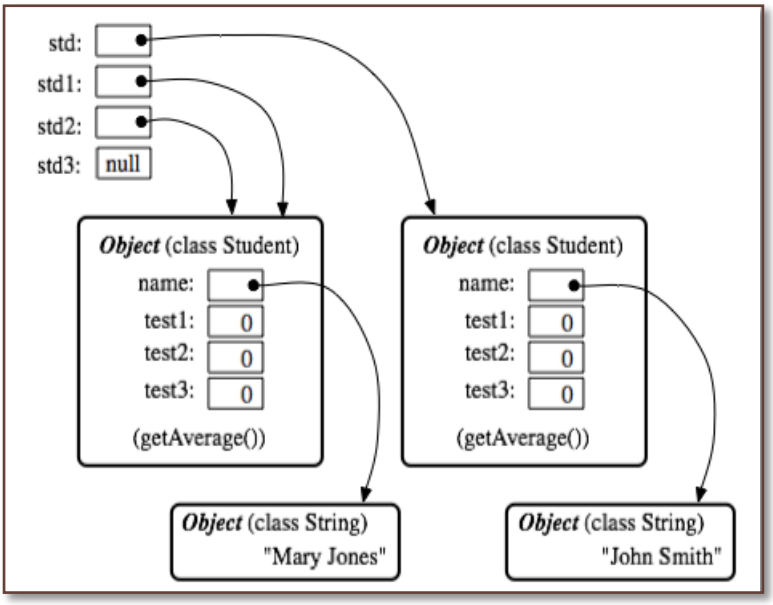


Figure 6. Situation in the computer memory

From the figure above we can see that when one object variable is assigned to another, only a reference is copied. The object referred to is not copied. When the assignment "**s2** = **s1**;" was executed, no new object was created. Instead, **s2** was set to refer to the very same object that **s1** refers to.

You can't test objects for equality and inequality using the operators == and **!=**. The test "**if (s1 == s2)**", tests whether the values stored in **s1** and **s2** are the same. However, the values are references to objects, not objects. So, you are testing whether **s1** and **s2** refer to the same object, that is, whether they point to the same location in memory. But the object is not in the variable, two identical objects can be stored in different locations. That is why this method is not suitable for checking objects' equality. In order to compare the real objects, you need to use equals() method (described further in this chapter) (1).

3.3 Class Modifiers

Class can also have modifiers. Their description is provided in Table 6 below:

Modifier	Description
public	The class is publicly accessible. Without this modifier, a class is only accessible within its own package.
abstract	No objects of abstract class can be created. All of its abstract methods must be implemented by its sublclass. Otherwise that subclass must be declared abstract too.
final	Can not be subclassed (Inheritance is going to be discussed in the next Chapter).

Table 6. Class modifiers

Normally, a file can contain multiple classes, but only one public one. The file name and the public class name should be the same.

3.4 Class members and instance members

The non-static members of a class (fields and methods) are also known as **instance variables** and **methods**. Besides, there are *static* members also known as **class variables** and **methods** (1).

Each object (instance) of a class has its own copy of the *instance variables* defined in the class. So, when you create an instance of a class, the system allocates enough memory for the object and its instance variables.

In addition to instance variables, classes can declare *class variables*. A class variable has a *static* modifier and contains information that is shared by all objects of the class (2). So, in case one object changes the variable, it changes all other objects of that type.

You can invoke a class method or access class field directly from the class, whereas you must invoke instance methods and access instance variables on a particular instance. For example, the methods in the *Math* class are static and can be invoked without creating an instance of the *Math* class. So we can say :

```
double d = Math.sqrt(x);
```

Looking at our *Pencil* class, we see that there are *color*, *length*, *diameter*, *price* instance variables and one static variable *numberOfPencils*. So, each pencil will have its own color, length, etc., but *numberOfPencils* will be common for all pencils. When a new Pencil object is created, *numberOfPencils* will be incremented.

As already mentioned, there is only one copy of a static variable, so the initialization of that variable is executed just once, when the class is first loaded. Remember that if you don't provide any initial value for an instance variable, a default initial value is provided automatically (8) - instance variables of numerical type (*int*, *double*, etc.) are automatically initialized to zero if you provide no other values, while boolean variables are initialized to false.

Finally, let's look at the example of how a variable can be shared. As you remember, *numberOfPencils* variable has a static modifier. Have a look at the following code:

```
Pencil p1 = new Pencil();
Pencil.numberOfPencils ++;
System.out.println(p1.numberOfPencils);
//Result? 1

Pencil p2 = new Pencil();
Pencil.numberOfPencils ++;
System.out.println(p2.numberOfPencils);
//Result? 2

System.out.println(p1.numberOfPencils);
//Result?    Again 2!
```

First, we create instance of Pencil and call it p1. Then, using class reference, we increment the number of pencils. At this moment, there is 1 pencil. Furthermore, we create one more instance of Pencil – p2 and again increment the number of pencils. For now, getting the value of *numberOfPencils* through p2 instance returns 2. But even if you will access this value through p1, it will be still 2, because the variable is shared, only one copy exists. Although it hardly makes sense, you can also change the value of static variable using object reference, like:

```
Pencil p1 = new Pencil();
p1.numberOfPencils ++;
```

3.5 More on Methods

We have already considered the concept of a method, which corresponds to an action or a behavior that object possesses. In other words, it is a named chunk of code that can be called upon or invoked to perform a certain pre-defined set of actions (3). Any method has a header (includes access modifiers, abstract or not, static or not, final or not) and a body (code itself).

Methods are invoked as operations on objects/classes using the dot (.) operator:

```
reference.method(arguments)
```

Remember that if the method is static, "reference" can either be the class name or an object reference belonging to the class. On the other hand, if the method is non-static, "reference" must be an object reference.

As we already know, a class can have more than one method with the same name as long as they have different parameter list, due to method overloading which is supported in Java. So, let's overload the *setPrice* method that we introduced before:

```
public void setPrice (double newPrice) {
        price = newPrice;
}

public void setPrice (Pencil other) {
        price = other.getPrice();
}
```

So, as you see, we have two methods with the same names, but the first one accepts variable of type *double*, while the second one - variable of type *Pencil*. Both of them set the value of the variable **price,** but do it in a bit different way. First method does this directly, whereas second one sets the price to a price of another pencil. For better understanding, let's look at a similar example from real life. Suppose you have a small shop where you sell office accessories. When you set a price for a pencil, for instance, you might think of all associated expenses and put some number, let's say 100 tenge (1st method in the code listing). Another way you can set price is when you understand that the pencil you want to set price has

approximately the same cost as some other pencil you already calculated expenses for (2nd method).

What do you think, how does the compiler know which method you're invoking? Actually, it does this by comparing the number and type of the parameters and uses the matched one.

Finally, let's look at how the method parameters are passed. Parameters are always passed by value, for example:

```
public void method1 (int a) {
    a = 6;
}
public void method2 ( ) {
    int b = 3;
    method1(b);      // now b = ?
}
```

As a result of execution the value of b will be 3. The value of b is not changed by the method, which is equivalent to:

```
a = b;
a = 6;
```

However, in case you have a reference type variable in place of b, it's value is going to be changed. So, what is going to happen after call to a *method2*:

```
public void method1 (Student a) {
    a.setName("Aray");
}
public void method2 ( ) {
    Student b = new Student();
    b.setName("Adiya");
    method1(b);      // now s is ?
}
```

The value of b is not changed. But the name of *Student* **b** is changed, since it is equivalent to:

```
a = b;
a.setName("Aray");
```

It is important to realize that an object variable doesn't actually contain an object. It only refers to an object. So, in our case we have two

references (a and b) pointing to the same Student. Both of them can equitably modify the state of the object (remember example with two remote controls for one TV?). So, a.getName() and b.getName() will both return "Aray".

Let's look at the other example where parameter is an object reference. As you can see from the Figure 7 below, we have an instance of class Pencil called *plainPencil.* We set it's color to be plain. At the right part of the figure you can observe what is happening in memory at every step. So, according to the picture, we have one reference (*plainPencil)* to the object of Pencil which color is plain. Have a look at a *paintRed* method. This method accepts some pencil and paints it to a red color. After that, it makes a reference p null (think, why?). Now return back to our step-by-step analysis of code. We call *paintRed* passing our newly created plain pencil. You see that in a method, pencil passed is denoted as p. It means that now we have two variables – *p* and *plainPencil* which are pointing to our Pencil object. Now, inside of the method, we use p reference in order to paint the pencil. From the third picture at the right you can see that for both references (plainPencil and p) the color has been changed. So, p made it's work, we don't need it anymore. So, we make the p reference null. Actually, these interesting processes actually take place when you work with objects in this way.

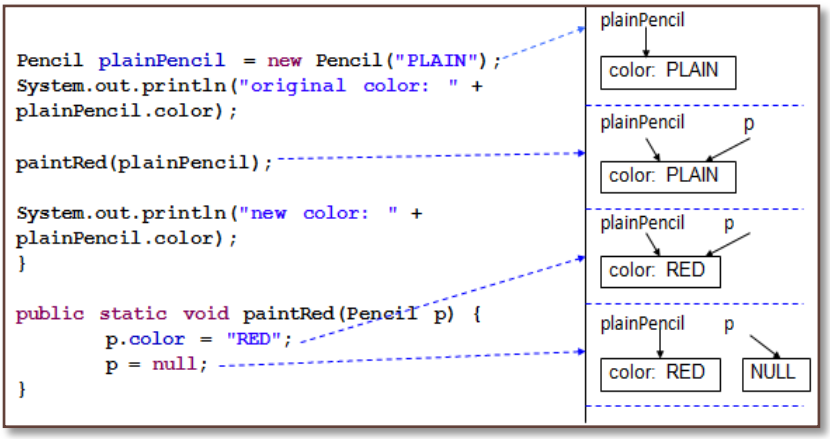


Figure 7. Passing object as a parameter

Therefore, when the parameter is an object reference, it is the object reference, not the object itself, getting passed. From this example, you need

to understand and keep in mind this simple rule: if you change any field of the object which the parameter refers to, the object is changed for every variable which holds a reference to this object.

Finally, a word about main method. The system locates and runs the main method for a class when you run a program (it must contain a public class). Other methods get execution when called by the main method explicitly or implicitly (3). Remember that it must be public, static and void.

3.6 Encapsulation

We've already touched some of the basics of the encapsulation concept. If you employ this technique, you can achieve the following:

- The fields of a class can be made read-only or write-only (by using accessor and mutator methods).
- A class can have a total control over what is stored in its fields. However, the users of a class do not know how the class stores its data (6).

Let's look at the encapsulation concept essence. In the object diagram (you can see below in Figure 8) it is shown that an object's variables (circles, starts, triangles, etc.) make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. So, objects communicate with each other by sending messages to each other (by virtue of methods). Objects package their variables within the protective custody of their methods, and this exactly refers to encapsulation phenomena (1).

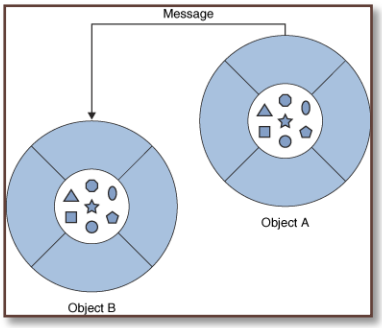


Figure 8. Object diagram

In this way, you maintain the information hiding. That is, an object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting other objects that depend on it (1). This is possible, because you change the implementation, not the public interface.

Many programmers claim that almost all member variables should be declared private. This allows you to control what can be done with the variable. Even if the variable is private, you can allow other objects to find out what its value is by providing a public accessor method that returns the value of the variable. For example, if your class *Employee* contains a private member variable *salary* of type *double*, you can provide an *accessor get* method hat returns the value of the salary.

```
public double getSalary() {
       return salary;
              }
```

You can also provide a mutator method in order to make salary editable. That is, you might want to make it possible for other classes to specify a new value for the variable:

```
public void setSalary(double salary) {
       this.salary = salary;
              }
```

Note that *set* method should have a parameter with the same type as the variable and get method must return a variable of that type.

As a rule, the name of an accessor/mutator method for a variable is obtained by capitalizing the name of variable and adding "get" / "set" in front of the name. So, for the variable salary, we get an accessor method named "get" + "Salary", or getSalary(). A getter method provides "read access", whereas setter method provides "write access" to a private variable.

By convention, it is desirable to provide both an accesor and a mutator (getter and a setter) methods for a private member variable.

You might want to ask a question - since providing get/set methods allows other classes both to see and to edit the value of the variable, why not just make the variable public? The answer is that getters and setters are not restricted to simply reading and writing the variable's value. For example, a getter method might keep track of the number of times that the variable has been accessed:

```
public double getSalary() {
       salaryAccessCount++;
       return salary;
                    }
```

Moreover, a setter method might check that the value that is being assigned to the variable is valid :

```
public void setSalary(double salary) {
       if(salary > 0)
              this.salary = salary;
                    }
```

There is a conceptual difference between the get and set methods. The get method only looks up the state of the object and returns it. The set method, in contrast, modifies the state of the object (2). So, they are called *accessor* and *mutator* methods. Let's define getters and setters for *Entry*:

```
public class Entry {
       private String name; // name as characters
       private String number; //phone number

       public void setName(String person) {
              name = person;
       }
       public String getName() {
              return(name);
       }
       public void setNumber(String phone) {
              number = phone;
       }
       public String getNumber() {
              return(number);
       }
}
```

Remember that encapsulating the member variables is a good programming practice.

3.7 Constructors

Constructor allows you to initialize an object. It is the only way to instantiate an object of some class into your program.

Every class has at least one constructor. Even If you don't provide any constructor in a class, then the system will provide a *default no-arg*[8] constructor for that class. This default constructor is used to allocate memory and initialize instance variables (3). It sets all the fields which have no initialization to be their default values(remember, for numerical types it is 0, for Boolean – false).

The definition of a constructor looks much like the definition of any other method, with certain differences:

1. Constructors have no return type.
2. The name of the constructor must be the same as the name of the class
3. The only modifiers constructor can have are the access modifiers public, private, and protected (a constructor can't be declared static) (1)

A class can have multiple constructors provided they have different parameter list (discussed later).

Remember Entry class. Let's define the constructor for this class that sets the values for name and number fields.

```
public class Entry {
	private String name; // name as characters
	private String number; //phone number
	public Entry(String person, String phone) {
		name = person; // initialise name
		number = phone; // initialise number
	}
}
```

Then Entry might be set up as follows:

```
Entry newEntry = new Entry("Ainur", "+77013467222");
```

Remember the order of parameters of the constructor when creating object via this constructor.

[8] Constructors with no arguments are called no-arg constructors

3.8 *This* Keyword

This keyword can be used in two ways: as a reference to the current object and to invoke another constructor in the same class. Let's look at these two ways deeper.

- **This as a reference to the current object**

Inside of the instance method and constructor **this** serves as a reference to the current object (that is, the object whose method or constructor is called). It allows you to refer to any member of the current object from a method or constructor (9).

Usually fields have the same names as a method or constructor parameters. In this case, you can use *this* to distinguish between them. Actually, that is the most common reason for using the *this* keyword. For example, the Point class can be written like this:

```
public class Point {
    int x = 0;
    int y = 0;
   //constructor
    public Point(int newX, int newY) {
        x = newX;
        y = newY;
    }
}
```

However, sometimes it might be useful to use the same names for parameters. So, using *this* keyword we can write the class in a bit different way:

```
public class Point {
    int x = 0;
    int y = 0;
    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
      }
}
```

This can also be used to invoke a constructor within another constructor.

- **This as an invoker of the constructor**

Within a constructor, *this* serves to invoke another constructor in the same class. This refers to *explicit constructor invocation.*

Consider the following class Circle that can be used to represent a 2D circle. Notice that it contains a variable of type Point (defined earlier) to represent a center of the circle in the coordinate system:

```
class Circle {
        double radius;
        Point center;
        double findArea(){
                return radius*radius*Math.PI;
        }
}
```

It is inconvenient to set up the fields individually, so an explicit constructors would also be useful as well as field methods:

```
public Circle(){
        radius = 1.0;
        center = new Point(0,0);
}
public Circle(double r) {
        radius = r;
}
public Circle(Point center, double r){
        this(r);
        this.center = center;
}
```

This class contains two constructors. Each constructor initializes some or all of the circle's member variables. First one sets the value for radius, while the second one - for the radius and center. In order not to duplicate the code we use this here to call the constructor that sets the radius and then set the center.

As you might guess, the compiler determines which constructor to call, based on the number and the type of arguments.

Now you can create circles in various ways depending on information you have:

```
Circle defaultCircle = new Circle();
Circle myCircle = new Circle(5.0);
Circle coolCircle = new Circle(new Point(883,1729),5.0);
```

By convention, if you use *this*() it must be the first statement in the constructor body if exists.

Keep in mind that this can't be used in a static method (see question at the end of the chapter).

3.9 Initialization Blocks

Initialization block constitutes a block of statements used to initialize the fields of the object. It needs to be placed outside of any member or constructor declaration. Initialization blocks are always executed **before** the body of the constructors (7).

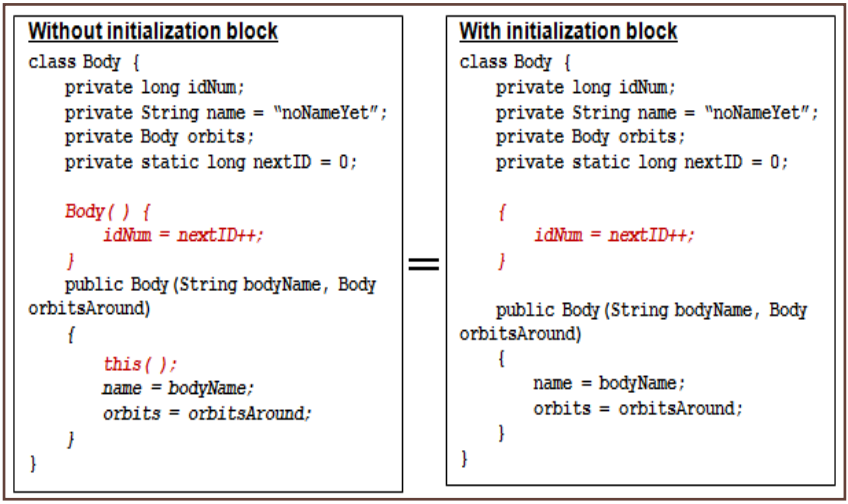


Figure 9. Using initialization blocks

From the figure provided above you can see how class definition might look like with and without initialization block. When we do not employ initialization block, we have to create an empty no-arg constructor in order to execute code that must be executed for every object. Then, in each and other remaining constructors we use this() to execute no-arg

constructor . The disadvantage of this approach is that in case you have one hundred overloaded constructors, you have to call this() for each of them.

Now have a look at the right side of the figure, where we employ initialization block in order to increment *id* for newly created object. All we have to do is to put the code that initializes variables into a block surrounded by curly braces. This code will be executed before the body of constructor for all objects.

In addition, there exist a static initialization block. It resembles a non-static initialization block (already discussed) except that it is declared static and can only refer to static members (8). It gets executed when the class is first loaded.

3.10 Class Importation

You have already seen an example of how to import classes you use in a program. Any complex object uses other objects for some functionality. An import statement tells the compiler where to find classes you use inside of your code. An import statement should be placed at the top of your source files.

To create an import statement you specify the import keyword followed by the class that you want to import followed by a semicolon (5). The class name include its package. So, an import statement usually looks like this:

```
import ClassNameToImport;
```

For example:

```
import java.util.Date;
```

If you want to import all classes within a package, you can put .* instead of the name of the class .[9]

There is actually one more way you can access public classes situated in other packages. Besides importing a single class or all the public classes (*), you can explicitly give the full package name before the class name, like in this example:

[9] Note that * is used to import classes at the current package level. So, it won't import classes in its subpackages.

```
java.util.Date today = new java.util.Date( );
```

There are several subtle issues we need to look at . One of them is connected with name conflict. Consider the following code:

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

Date today = new Date( );//ERROR:java.util.Date
                          //or java.sql.Date?
```

We have an error because both packages have a class Date. So, how to solve this problem? If you only need to refer to one of them, import that class explicitly, like:

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

Date today = new Date( ); // java.util.Date
```

In case you need both Data classes, you have to use the full package name before the class name, like:

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

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

For the static members situation is a bit different, you need to refer them as *className.memberName*, like in the example below:

```
import java.lang.Math;

public class importTest {
       double x = Math.sqrt(1.44);
              }
```

So, you cannot use *sqrt* without class name, since it is a static method.

Starting from J2SE 5.0 version, importation can also be applied on static fields and methods, not just classes (6). You can directly refer to them after the static importation. As an illustration for that, let's import all static fields and methods of the Math class:

```
import static java.lang.Math.*;
double x = PI;
```

Additionally, you can import a specific field or method:

```
import static java.lang.Math.abs;
double x = abs(-1.0);
```

This saves your time if you frequently use methods and fields of such kind of packages.

3.11 Class Abstraction

Class abstraction means to isolate class implementation from the use of the class. In other words, the implementator of the class provides a description of the class in order to let the user know how the class can be used. Therefore, to use the class, the user does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user. So, for example, if you want to write a class Employee and test it, it is strongly recommended to put them in different classes Employee.java (the implementation, not public class) and EmployeeTester.java (public class), for example.

3.12 Enumerations

An **enum** is a type that has a fixed set of possible values, specified when the enum is created. The definition of an enum types has the following form:

```
enum enum-name { value1, value2, …};
```

For example, the following enumeration might be used to represent the year seasons:

```
enum Season { SPRING, SUMMER, AUTUMN, WINTER };
```

The enum definition cannot be inside a method. You can put it outside the main() method of the program.

Another examples when it can be useful to use enumerations:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
                   FRIDAY, SATURDAY };
enum Gender { MALE, FEMALE};
enum Size { XS, S, M, L, XL, XXL};
```

You need to understand that *enum* values are not variables. Actually, each *enum* value is a constant (remember that constant names are written uppercased!) that always has the same value. You can refer to them as Season.SPRING, Season.SUMMER, etc.

You can create *enum* objects in exactly the same way as for other types. For example, you can create the following variables of the enums provided above:

```
Gender g = Gender.MALE;
Day d = Day.SATURDAY;
Season s = Season.SPRING;
```

You can print an *enum* value with the usual System.out.println() statement and the output value will be the name of the *enum* constant.

Finally, let's look at some methods you can use when working with enumerations. One of the useful methods is *ordinal()*. It simply returns the position of the value in the list (starting from 0). That is, **Season.SUMMER.ordinal()** is the int value 1. In addition, there is *values()* method returning the list containing all the constants that make up the enumeration (1).

3.13 Wrapper Classes

Sometimes there is a necessity to manipulate the primitive types as if they are objects. In this case you can use one of the wrapper classes provided by Java API or define your own. As a result, you will be able to create objects that represent values of primitive type. For example, Java API contains the classes Double, Integer, Character, Boolean,… (which wrap a single double, int, char, Boolean, …) .

These classes contain various static methods including *Integer.parseInteger* that is used to convert strings to numerical values. Integer contains constants like *Integer.MIN_VALUE* and *Integer.MAX_VALUE*, which are equal to the largest and smallest possible values of an integer can have. They are −2147483648 and 2147483647

respectively. The similar methods are provided by other wrapper classes representing numerical types.

To create the object of a wrapper classes, you use the same syntax as for usual classes:

```
Integer count = new Integer(0);
```

So, the value of *count* contains the same information as the value of type *int*, but it is an object. If you want to retrieve the *int* value that is wrapped in the object, you can call the method *count.intValue()*. Similarly, you wrap the *double* in an object of type Double, a *boolean* value in an object of type Boolean, etc.

3.14 Core Java classes

You got acquainted with the concept of a class. Classes can be grouped in a collection called package. Java's standard library consists of hierarchical packages, such as java.lang and java.util (7). Let's look at some useful packages you might frequently use while programming in Java.

Package	Description
java.lang	Contains core Java classes, such as numeric classes, strings, and objects. This package is imported by default to every Java program
java.io	Contains classes for input and output streams and files
java.util	Contains many utilities, such as date and time, tokenizer, random, legacy collection classes
java.net	Classes for supporting network communications.

Table 7. Core packages

Now let's look how some classes from these packages can be used. However, here we provide just short summary of usage. For deep information, please find see java API.

- **GregorianCalendar**

This class provides the standard calendar used by most of the world

(9). It has several useful constructors. The expression new GregorianCalendar() constructs a new object that represents the date and time at which the object was constructed.

You can construct a calendar object for midnight on a specific date by supplying data for the corresponding year, month, and day, like:

```
GregorianCalendar birthday =
new GregorianCalendar(2011, 06, 15);
```

It is of interest to note that the months are counted from 0. Therefore, 06 is July. To avoid confusion, you can use constants, like Calendar.JULY, for example. Using Calendar class allows you to set the time inside of constructor:

```
GregorianCalendar birthday = new
GregorianCalendar(2011, Calendar.JULY, 15, 14, 21, 59);
```

GregorianCalendar exposes very useful methods. For example, you can get the month, weekday, etc. of some date. The code below does this for current date.

```
aGregorianCalendar now = new GregorianCalendar();
int month = now.get(Calendar.MONTH);
int weekday = now.get(Calendar.DAY_OF_WEEK);
```

Check the API to find the list all the constants that you can use.
In order to make changes to the state of the date use the set method:

```
birthday.set(Calendar.YEAR, 2011);
birthday.set(Calendar.MONTH, Calendar.SEPTEMBER);
birthday.set(Calendar.DAY_OF_MONTH, 25);
// or in one call
birthday.set(2011, Calendar.SEPTEMBER, 25);
```

When working with dates you might need to add a certain number of days, weeks, months and so on to it. You can even add a negative number. In this case, the calendar will move backwards (8):

```
myDate.add(Calendar.MONTH, 4);
myDate.add(Calendar.MONTH, 2);
```

Check out API to investigate the additional functionality of *GregorianCalendar*.

- **String**

The String class is used to represent character strings. Strings are immutable in Java meaning that their values cannot be changed after they are created. If you need dynamic strings, you can use StringBuffer (9) or StringBuilder.

The following main methods are included in a class String :

- **charAt()** - for examining individual characters of theString
- **compareTo()** - for comparing strings alphabetically
- **substring()** - for extracting substrings
- **toLowerCase(), toUpperCase()** - for creating a copy of a string with all characters translated to uppercase or to lowercase.
- **length()** - returns the number of characters contained in the string object.
- **split()** - Splits the string by some separator

For more methods, visit Java API site. A small example of some String methods is provided below.

```
for(int i=0; i<s.length(); i++){
   for(int j=i+1;j<=s.length();j++){
      System.out.println(s.substring(i,j).toUpperCase());
   }
}
```

In this example we print all uppercased substrings of a given string s:

- **Vector**

Vector class represents a growable array of objects (dynamic array). Similar to array, it contains objects that can be accessed by index. The difference is that the size of a Vector can change as needed in accordance with addition and removal of items (9).

```
Vector<String> students = new Vector<String>();
students.add("Elmira");
students.add("Guizal");
for(String cur: students)
    System.out.println(cur);
```

Note that Vector class can store any object – String, Integer, or any object defined by you. For example, you might want to store Student objects instead of String.

Vector has a wide range of utility methods, like clearing the content, removing, adding, inserting elements, checking the element for presence, etc.

3.15 Defining your own classes

In this part of the Chapter we will put all the concepts introduced together and create two classes: Body and Employee.

- **Body**

Consider a row version of a Body class, that can be used to represent a cosmic object:

```
class Body {
              private long idNum;
              private String name = "empty";
              private Body orbits;
              private static long nextID = 0;
       }
```

As it is seen from the listing above, any cosmic Body has an id, name, and reference to a body it orbits around. All Body instances share the variable *nextID*, which reflects the number of bodies and is used to assign *idNum* each time.

Remember that an object is created by the new keyword. Only after this statement the runtime system will allocate enough space in order to store the new object. So, let's create it. Stop! In order to create objects, we need to have constructor(s). Although default no-arg constructor is created implicitly, let's define other ones that initialize the Body's variable:

```
Body( ) {
        idNum = nextID++;
}

Body(String name, Body orbits) {
        this( );
        this.name = name;
        this.orbits = orbits;
}
```

Now we can create a couple of Body objects:

```
Body sun = new Body("Sol", null);
Body earth = new Body("Earth", sun);
```

Values of sun and earth variables are illustrated in figure 8 below.

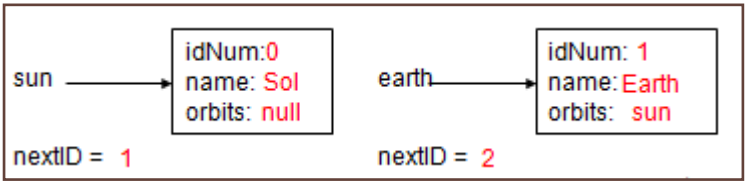


Figure 10. Situation in memory

You might want to store all body objects in some collection, say Vector. For that, you need to create static Vector object and add every newly created Body to this Vector:

```
private static Vector<Body> bodies = new Vector<Body>();

private void addBody() {
              bodies.add(this);
}
```

Now let's define get/set methods for the instance fields, which are private. Note that in case we defined them public, they would be exposed to change by every body. But now, having them private does not allow access the fields. In order to allow fields to be accessed, let's implement gettes:

```
public long getID() {return idNum;}
public String getName() {return name;};
public Body getOrbits() {return orbits;}
```

After this piece of code the fields idNum, name and orbits became

read-only outside of the class. But we are still not able to change the value of the fields. So, let's do that by providing mutator set methods:

```
public void setName(String newName) {
            name = newName;
      }
      public void setOrbits(Body orbitsAround) {
            orbits = orbitsAround;
      }
```

Now it's time to look at the other class – Employee.

- **Employee**

Now let's consider the next example - simplified, version of an Employee class:

```
class Employee {
      private String name;
      private double salary;
      private Date hireDay;
      public Employee(String name){
            this.name = name;
            GregorianCalendar now =
                  new GregorianCalendar();
            this.hireDay = now.getTime();
      }
      public Employee(String name, double salary){
            this(name);
            this.salary = salary;
      }
}
```

From the code listing above we see there are 3 private instance fields – name, salary and the Date when employee was hired. We also defined two constructors – the one accepts a single parameter of type String (representing name) and initializes the name. Besides, it sets the hire date to be the current date. Really, creation of the employee object must happen when he is hired. We also have a second constructor, that, apart from name , accepts a variable of type double in order to set the salary. Besides salary, we need to set name and hire date. As it can be seen, we use **this** keyword to call the first constructors, in which we already did this work.

Furthermore, we need to define get/set methods for the private fields. Hope you can do it already, so let's do this for one field – salary:

```
public double getSalary(){
        return salary;
}

public void setSalary(double salary){
        this.salary = salary;
}
```

Note that we do not need to define the set method for name – it is illogical. Make it read-only, providing only get method. By doing this, you'll have a guarantee that this field will never be corrupted.

If some Employee is working perfectly for some period, chef might want to raise his salary. Let's define the method for that:

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

Now consider a method equals that s used to compare two employees:

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

This method states that two employees are the same when they have the same name. The typical usage look like this:

```
if (zilan.equals(jack)){}
```

We implemented the raw version of our Employee class. Now it's time to define a test class for it. First of all, let's create a storage (vector) for our employees (call it workers). Then, create several instances of Employee and store them in a Vector:

```
Vector<Employee> workers = new Vector<Employee>();
Employee zilan = new Employee("Zilan", 2000);
Employee ali = new Employee("Ali");
// salary is not yet defined for Ali
workers.add(zilan);
workers.add(ali);
// pass newly created object instantly
workers.add(new Employee("Madina", 1500));
```

Let's be kind and raise salary of each employee by 10 %. Use foreach for that purpose:

```
for (Employee e : staff)
        e.raiseSalary(10);
```

Print information about employees. Check that salaries have been updated:

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

A bit inconvenient way to print information about an employee, isn't it? In Java we have a special method toString() of class Object (remember that Object class is a parent of any class in Java). It is used to return a string representation of the object. If you do not provide any implementation of this method, it will return the memory location of the object. Let's define it:

```
public String toString(){
            return "name is " + name +
                 ", salary is " + salary +
                 " and hireDay is " + hireDay;
      }
```

After addition of this code to Employee, we can print information in this way:

```
for (Employee e : staff)
            System.out.println(e.toString());
```

Or, even simpler, in this way:

```
for (Employee e : staff)
            System.out.println(e);
```

When you print the name of the variable storing the object, toString() method is called implicitly.

3.16 Class design

The algorithm for effective class design is straightforward. First, you need to look at the main actors and objects in the system and identify classes based on them. Secondly, describe the attributes and methods for each of them. Then, when you already have an outline of classes, try to establish relationships among them (There can be various types of relationships between classes, like Association, Aggregation, Generalization (Inheritance), Realization, among others). Finally, you need to implement classes.

KEY CONCEPTS YOU NEED TO GRASP FROM THE SECTION

Enumeration *equals()* *Wrapper class* *get/set methods* *Vector* *Constructor* *Class variable* *Instance variable* *Initialization block* *Importation* *this keyword* *Encapsulation* *Object state and behavior*

TESTS

1. The compiler creates a default constructor only when there are no other constructors for the class. True or False?

a) True b) False

2. This process is also known as the act of information hiding. It conceals the functional details of a class from objects that send to it.

a) Polymorphism b) Inheritance

c) Encapsulation d) Abstraction

3. It contains abstract characteristics of an object including its states and behaviors. Also refers to a yemplate for creating objects.

a) Instance b) Class c) API d) Record

4. Class member which determines behavior of the object is:

a) field b) function

c) interface d) method

5. What is the return type of the constructor?

a) null b) default value 0

c) user-defined d) no return type

6. Suppose you have an enum Season { WINTER, SPRING, SUMMER, AUTUMN }. How we can refer to enum:

a) Season.WINTER

b) Season vacation; vacation.enum();

c) Season vacation; vacation = Season.WINTER;

d) a) and c) are correct

7. What is true about this keyword?

a) Can be used in all methods

b) Must be 1 statement in the constructor body if exists

c) Can be used as a reference of the current object

d) a) b) c) are correct

e) b) c) are correct

8. What is true about this statement:

Animal dog = new Animal(«Shurik»);

a) Dog is an object of class Animal

b) Class Animal has a only one constructor with parameter of String type;

c) Class Animal may have many constructors, one of them has one parameter of type String

d) a) and c) are correct

9. Choose the correct method:

a) public void setName(String newName) {return name;}

b) public String getName() {return name;}

c) public String getName(){ name=new name;}

10. Which statement about objects is true?

a) One object is used to create one class

b) One class is used to create one object

c) One object can create many classes

d) One class can create many objects

11. Suppose you have a method *setValue* that assigns the value *coolName* to the instance field called *name*. What could you write inside of *setValue*?

a) name = coolName b) this.name = coolName;

c) coolName == name d) a and b are correct

12. Given the declaration Rectangle r = new Rectange(), which of the following statements is most accurate?

a) r contains a reference to a Rectangle object.

b) You can assign an int value to r.

c) r contains an int value.

d) r contains an object of the Rectangle type.

13. How can you compare two objects of a class Student (s1 and s2)?

a) if (s1 == s2) {…}

b) if (s1.equals(s2)) {…}

c) if (equals(s1, s2)) {…}

14. Choose a wrapper class:

a) double b) Double

c) Int d) Char

15. Which of the following fields can be static inside of a class Employee?

a) name b) experience

c) salary d) companyName

PROBLEMS

1. Try to perform an object-oriented analysis of the interaction between a student, librarian, and a books database in case student wants to take a book(s) out of the library.

2. Static final fields must be initialized when the class is initialized, whereas non-static final fields must be initialized when an object of the class is constructed. Explain why.

3. Think of a *Dog* and implement corresponding class. Make several constructors to this class.

4. Constructors are used to initialize a class. If constructor is private, it means, that it won't be seen outside of the class. So, why we might need private constructors**?**

5. Why **this** can't be used in a static method?

6. Write a program that displays a calendar for the current month. Mark the current day by an asterisk (*).

Hint: How to compute the length of a month and the weekday of a given day?

```
Sun Mon Tue Wed Thu Fri Sat
                        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  31
```

7. Implement your own class for handling dates – Date. Any Date has a year, month, day, hour, minutes and seconds. Provide no-arg constructor which sets the date to a current date. Then create an

instance of your class, call it *birthday*. Next create another instance - *deadline*, and equate *deadline* to *birthday*. Then, move deadline by 3 days. Draw the situation in memory (show references, objects, their values, etc.).

8. Implement a class *Student.* Student has a *name*, *id* and a *year of study*. Provide a constructor with two parameters and create methods to access name, id and increment the year of study.

9. Write a class *StarTriangle* that can be used to generate the following triangular shape (at the right).

[*]

[*][*]

[*][*][*]

...

- Your class should have a constructor with a parameter width, that indicates the number of [*] in the last row of the triangle.
- Moreover, there must be a method toString() that computes a string representing the triangle and returns a string consisting of [*] and newline characters.

Class usage:

```
StarTriangle  small = new StarTriangle(3);
System.out.println(small.toString());
```

The output is the same as in illustration.

10. Write a class Data that computes information about a set of data values. It should have:

- 3 private fields: 2 of type double and 1 of type int (you need to guess what to store in these fields)
- Constructor that constructs an empty data set.
- Method that adds a value to a data set.
- Method that returns average of the added data or 0 if no data has been added.
- Method that returns the largest of the added data.

Note: You are not allowed to store ALL values ! Do not use arrays or something like this!

Then write a class Analyzer that uses *Data* class described above to compute the average and maximum of a set of input values.

- Use Scanner to take data from user
- The process of taking inputs from user must continue until the user types "Q" instead of a number.

For example:

```
Enter number (Q to quit): 10
Enter number (Q to quit): 0
Enter number (Q to quit): -1
Enter number (Q to quit): Q
Average = 3.0
Maximum = 10.0
```

11. Create a class *Interval* that represents interval on the x-axis. An interval is the set of points in the range [left, right]. Include check that the left endpoint is no greater than the right endpoint and a method intersects() so that a.intersects(b) returns true if the intervals a and b intersect, and false otherwise.
Class usage:

```
Interval i = new Interval(3,5);
```

Remember that point is also an interval.

12. Create an enumeration *Days* to store days of week. Also, create a class *Time* with fields *hours*, *minutes*. Then create a class *Lesson* having an instance of your enumeration, instance of *Time* and a field *name* (String). So that we can write:

```
Lesson oop = new  Lesson ("oop", Days.MONDAY,
new Time(14,30));
```

13. Design and implement a class Polynomial, that represents a polynomial with real coefficients. The coefficients of the polynomial should be passed as an array parameter with array type double in the constructor of your class. Hint: the resulting Polynomial is defined by `coefficients[0] + coefficients[1] * x + coefficients[2] * x2 + ... + coefficients[n] * xn`

Implement 3 methods:

- to multiply a double value to a polynomial
- to return the first derivative
- to return the value f(x) of a polynomial for a given a given value x.

14. You need to write a *Temperature* class that has two fields: a temperature value (a double number) and a character for the scale, either 'C' for Celsius or 'F' for Fahrenheit. Make sure that these two fields can only be accessed through the accessor methods outside of the class.

Constructors:

The class should have four constructors:

- one for each instance field (assume zero degree if no value is specified and Celsius if no scale is specified)
- one with two parameters for the two instance variables
- default constructor (set to zero degrees Celsius).

Methods:

- Two methods to return the temperature: one to return the degrees in Celsius, the other to return the degrees in Fahrenheit. Use the following formulas for conversion:
  degreesC = 5(degreesF – 32) / 9
  degreesF = (9(degreesC/5) + 32
- Three methods to set the fields: one to set the value, one to set the scale ('F' or 'C'), and one to set both.
- Method to return scale.

15. Implement a class *Car*. Car has a certain fuel efficiency, measured in km/liters and a certain amount of fuel in the gas tank. The efficiency is specified in the constructor, and the initial fuel level is 0.

- Supply a method drive() that simulates driving the car for a certain distance, reducing the amount of gasoline in the fuel tank.
- Also create a method getGasInTank(), returning the current amount of gasoline in the fuel tank, and addGas(), to add gasoline to the fuel tank.

Note: You can assume that the drive method is never called with a distance that consumes more than the available gas. Also create a *CarTester* class that tests all methods.

16. Write a class *Time* that has the following fields : *hour*, *minute*, *second* . Also, you need to have a corresponding constructor and method that set the time according to provided hour, minute and second parameters (check for any invalid input). Moreover, you need to have two methods for time format conversion (Universal and Standard) and method to add two Time objects. It can be either static (in this case there will be 2 parameters of type Time, or it can be non-static instance method taking one Time parameter.

17. There is a very scary dragon living in Almaty city near KBTU. Everyday he needs to eat several young students for a launch. He usually kidnaps them one by one in the morning and eats at a launch time, having put them in a line at a cell in his prison. But sometimes he has problems with his launch, because students vanish! He still doesn't know that pair of boy and a girl (B-G) can disappear if they stand together exactly in this order (B-G) due to the magic of love. After that line becomes smaller. So, there is a possibility that no one will be left for a dragon launch!

You need to model a dragon launch. It is to have:

- Enumeration Gender that is used to distinguish boys / girls.
- Class Person containing an instance variable of type Gender, method toString() and any fields you want.
- Main class – DragonLaunch, with methods:
  - **`kidnap(Person p)`**
  - **`willDragonEatOrNot()`**

For example, for a line BBGG, there will be no launch, because firstly middle pair will vanish, after that two corner persons will become BG pair, and vanish in the same way. However, a line GBGB leaves 2 persons for a launch.

Note: In your main class, overload the method *willDragonEatOrNot* to accept a String of boys and girls. Then Use Random class in order to generate 10 random sequences consisting of boys and girls. For each sequence, print the answer.

4 Inheritance, Polymorphism and Abstract Classes

"General propositions do not decide concrete cases".
Oliver Wendell Holmes

As we already studied in previous chapter, a class represents a set of objects sharing the same structure and behaviors. The class defines the structure of objects by specifying variables that are contained in each object of the class, and behavior is expressed via the instance methods. This is a powerful idea. Nevertheless, something similar can be done in most programming languages. The breakthrough idea in object-oriented programming, distinguishing it from traditional imperative programming – is the ability to express the similarities among objects that share *some, but not all,* of their state and behavior. Such similarities can be expressed by virtue of **inheritance** and **polymorphism**. Despite the fact that fundamental ideas of OOP are reasonably simple and clear, unfortunately, beyond them there are a lot of details (1).

4.1 Concept of Inheritance. Subclass and Supeclass

The concept of inheritance allows developers to write code in the form of hierarchical relationships. So, a collection of behaviors and attributes can be encapsulated into an isolated body known as an object. Consequently, this object can be used when you create additional objects by deriving them from the original object. Just like horses inherit the parameters and behaviors associated with mammals and vertebrates, or even children who benefit from the possessions of their mother and father, so can objects benefit through inheritance and inherits the attributes and behaviors of superclasses (4).

Once more, the main idea behind the inheritance is that you can create new classes (subclasses) that are built on existing classes (superclasses). Through the way of inheritance, you can reuse the existing class's methods and fields, and you can also add new methods and fields to adapt the new classes to new situations.

Subclass and *superclass* can also be referred to as a *Base* class and *Derived* class, *Parent* class and *Child* class. Parent and child classes have a *IsA* relationship: an object of a subclass *IsA(n)* object of its superclass, for example, *Student* (subclass) is a *Person* (superclass).

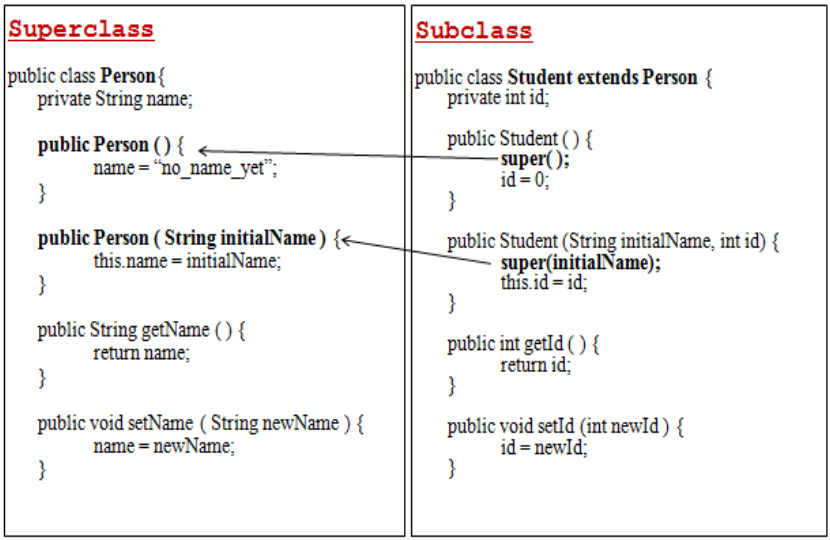


Figure 11. Student (subclass) is a Person (superclass)

It must be admitted that the inheritance relationship is transitive: if class B extends class A, then class C which extends class B, will also inherit from class A which is the parent of its parent B.

4.2 Class Hierarchy

Every class is an extended (inherited) class, whether or not it's declared to be. If a class does not declared to explicitly extend any other class, then it implicitly extends the **Object** class (described in details further in this section). So, the class hierarchy of previous example is:

Object
↓
Person
↓
Student

Obviously, an object of an extended class contains two sets of fields and methods: ones which are defined locally in the extended class and the ones which are inherited from the superclass.

In order to indicate the parent class you need to use ***extends*** keyword. Examples are provided below:

```
class Transport {…}
class Car extends Transport{…}
class Audi extends Car {…}
class AudiA6 extends Audi {…}

class Person {…}
class Student extends Person {…}
class Employee extends Person {…}
class KBTUStudent extends Student{…}
```

4.3 Special Keyword *super*

A constructor of the extended class can invoke one of the superclass's constructors by using the *super* method. If no superclass constructor is invoked explicitly, then the superclass's no-arg constructor *super( )* is invoked automatically as the first statement of the extended class's constructor.

Keep in mind that ***constructors are not methods and are NOT inherited!***

So, the keyword **super** refers to the superclass of the class in which *super* appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

Figure 11 above provides an illustration of using the *super* keyword to invoke the superclass constructor. Do not get confused with **super**() and **this**(). *This* is used as a reference to a current object and can be used to call a constructor inside of another constructor of the same class. So, *this* is completely unrelated to superclass and very different from *super*.

The reason *super* exists is so you can get access to members in the superclass that are hidden by subclass members. For example, *super.x* always refers to an instance variable named *x* in the superclass. Practically, this can be extremely useful in case a class contains an instance variable with the same name as an instance variable in its superclass. In this case, an

object of will actually contain two variables with the same name: one defined as part of the class itself and one defined in the superclass. The subclass variable does not replace the variable of the same name in the superclass, it just hides it. So, the variable defined in the superclass can still be accessed, using **super** keyword. The same holds for methods: when you write a method in a subclass that has the same signature as a method in its superclass, the method from the superclass is hidden in much the same way. All in all, you need to understand that although we say that the method in the subclass overrides the superclass method, it can still be accessed (1).

For example, suppose that within the *Student* class that extends the basic *Person* class, you wanted to concatenate the result of both the default and the new *toString()* methods. The default *toString()* method can be invoked using the super keyword, as in the example below:

```
class Person{
        String name;
        . . .
        public String toString(){
                return "I am "+name;
        }
}
class Student extends Person{
        . . .
        public String toString(){
                return super.toString()+"student at KBTU";
        }
}
```

4.4 Phases of Object Creation

When an object is created, memory is allocated for all its fields, which are initially set to be their default values. It is then followed by a **three-phase construction**:

- Invoke a superclass's constructor
- Initialize the fields by using their initializers and initialization blocks
- Execute the body of the constructor

The invoked superclass's constructor is executed using the same three-phase constructor. This process is executed recursively until the Object

class is reached. In Figure 12 you are provided with a stepwise event logs connected with object creation.

```
class X {
     protected int xOri = 1;
     protected int whichOri;
     public X() {
          whichOri = xOri;
     }
}
```

```
class Y extends X {
     protected int yOri = 2;
     public Y() {
          whichOri = yOri;
     }
}
```

```
Y objectY = new Y();
```

Step	What happens	xOri	yOri	whichOri
0	fields set to default values	0	0	0
1	Y constructor invoked	0	0	0
2	X constructor invoked	0	0	0
3	Object constructor invoked	0	0	0
4	X field initialization	1	0	0
5	X constructor executed	1	0	1
6	Y field initialization	1	2	1
7	Y constructor executed	1	2	2

Figure 12. An example illustrating the construction order

As we mentioned, a subclass extends properties and methods from the superclass. You can also add new properties, add new methods and override the methods of the superclass. So, suppose you have a class *Circle* having its own fields (*radius*) and methods (e.g. *findArea*()). A class *Cylinder* extending the circle has all fields and methods inherited from *Circle*. In addition, it also has a field *length* and, besides get/set methods for it, Cylinder has a method to find the volume. Since volume is the result of multiplication of length and circle area, we can call the method of the superclass (you can do this with or without the *super* keyword, in case your child class has a method with the same name) to find the area and multiply it by length. Figure 13 provides a more detailed explanation for that.

In order to ensure that all definitions of all superclasses are linked, the constructor method initiates and invokes the constructor of its superclass, and so forth, until the Object class's constructor is initialized. This is called an *inheritance chain* (3).

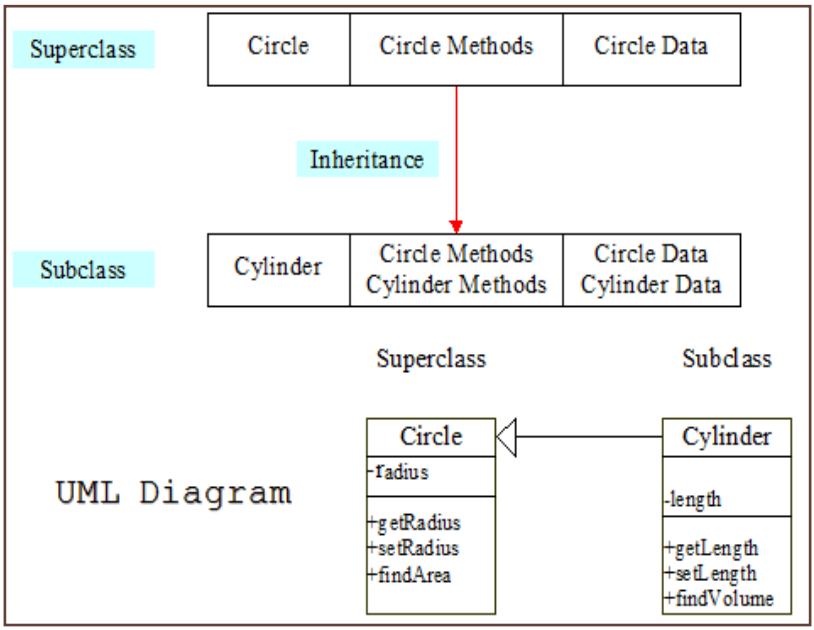


Figure 13. Superclass (Circle) and Subclass (Cylinder)

4.5 Overloading and Overriding Methods

As we already know, *overloading* refers to providing more than one method with the same name but different parameter list. Therefore, *overloading* an inherited method means simply adding new method with the same name and different signature. In contrast, by ***overriding***, you replace the superclass's implementation of a method with your own design. For example, in the code listing above inside of the Student class we overridden the *toString()* method defined in the superclass. Several issues that are important for *overriding*:

1. Both the parameter lists and the return types must be exactly the same
2. If an overriding method is invoked on an object of the subclass, then it's the subclass's version of this method that gets implemented

3. An overriding method can have different access specifier from its superclass's version, but only wider accessibility is allowed.

By the way, a superclass method can be overridden only if it's accessible in the subclass. Therefore, *private* methods in the superclass cannot be overridden. So, if a subclass contains a method which has the same signature as one in its superclass, these methods are totally unrelated

Furthermore, *package* methods in the superclass can be overridden if the subclass is in the same package as the superclass. *Protected* and *public* methods always can be overridden.

4.6 Polymorphism

The concept of *polymorphism* is used in many areas of the sciences, as well as in everyday life (physics, economics, programming etc.). The word *polymorphism* comes from a Greek word meaning "many forms". So, in general terms, polymorphism refers to the ability to appear in many forms

There are two types of polymorphisms - one is *static* and another one is *dynamic* (3). *Overloading* is example for static polymorphism and *overriding* is example for dynamic polymorphism. Method overloading refers to static polymorphism, since it resolves on compile time while method overriding is resolved at runtime. An object of a given class can have multiple forms: either as its declared class type, or as any subclass of it. Clearly, an object of an extended class can be used wherever the original class is used.

So, **Polymorphism** refers to the ability to determine at runtime which code to run, given multiple methods with the same name but different operations in the same class or in different classes. This ability is also known as ***dynamic binding***.

Remember that when you invoke a method through an object reference, the *actual class of the object* decides which implementation is used. In contrast, when you access a field, *the declared type of the reference* decides which implementation is used during run time. Let's look at the following example in order to clarify those statements:

```
    class SuperShow {
          public String str = "SuperStr";

          public void show( ) {
                System.out.println("Super.show:" + str);
          }
    }

class ExtendShow extends SuperShow {
       public String str = "ExtendedStr";

       public void show( ) {
             System.out.println("Extend.show:" + str);}

       public static void main (String[] args) {
          ExtendShow ext = new ExtendShow( );
          SuperShow sup = ext;
          sup.show( );//1
          ext.show( );//2 methods invoked through
          //object reference
          System.out.println("sup.str =" + sup.str); //3
          System.out.println("ext.str = " + ext.str);//4
          // 3, 4 field access
       }
}
```

What will be the output of the code above ? Taking into account two rules mentioned above, the output should be:

```
Extend.show: ExtendStr
Extend.show: ExtendStr
sup.str = SuperStr
ext.str = ExtendStr
```

As you see, when you invoke methods, the ones belonging to actual class are executed (for both instances, *ext* and *sup*, *ExtendShow* is an actual class). However, when you want to access a field, the fields of declared class are actually accessed (for *ext*, the declared class is *ExtendShow*, whereas for *sup*, declared class is *SuperShow*).

Although there are many ways to define polymorphism and it highly depends on the context of usage, there is one thing that we can say confidently about *polymorphism*, is that one method, object can take several forms. So, *speak()* can take the form of *"Woof", "Quack", "Meow",* etc.

Because we know that almost all animals have some kind of sound they produce, we can define a method speak() in a class *Animal*. Then, inside of all classes extending the Animal (Dog, Cat, Crocodile, etc.) we override the definition of speak() to correspond to particular animal. Note that in case some animal does not produce any sound and so you do not provide the implementation for it, the call to speak() will still be valid and result in s String "No voice". This happens because default method implementation was inherited from a parent class.


```
class Animal{
        public void speak(){
                System.out.println("No voice");
        }
}
class Dog extends Animal{
        public void speak(){
                System.out.println("Woof");
        }
}
```

In the example above we used method *overriding*, since subclass replaces the implementation of its parent's method.

```
public class Test {
        public static void main(String[] args) {
             m(new A());  // B
             m(new B());  //B
             m(new C());  //C
             m(new Object());    //java.lang.Object@192d342
        }
        public static void m(Object o) {
          System.out.println(o.toString());
        }
      }

      class A extends B {}
      class B extends C {
        public String toString() {     return "B"; }
      }
      class C extends Object {
        public String toString() {     return "C"; }
      }
```

Method *m* takes a parameter of the *Object* type. So, you can invoke *m* with any objects (e.g. new A(), new B(), new C(), and new Object()). An object of a *subclass* can be used by any code designed to work with an object of its *superclass*.

When the method *m* is executed, the *o* object's **toString** method is invoked. This *o* may be an instance of A, B, C, or Object. Classes A, B, C, and Object have their own implementation of the **toString** method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*. Consider another example:

```
class Person{
	String name;
	public Person(String name){
		this.name = name;
	}
	public String toString(){
		return "I am "+name+", person at KBTU";
	}
}
class Student extends Person{
	public Student(String name){
		super(name);
	}
	public String toString(){
		return "I am "+name+", student at KBTU";
	}
}
class Employee extends Person{
	public Employee(String name){
		super(name);
	}
	public String toString(){
		return "I am "+name+", employee at KBTU";
	}
}
```

As it can be seen, both *Student* and *Employee* override the *toString*() method defined in a superclass *Person*. This can be extremely useful to have some method defined in a parent class and override this method in child classes by providing more specific implementation. This allows us, for example, to store all objects of child classes of Person (Student, Employee, Manager (subclass of Employee), Guard, Dean, etc.) in one storage (list or array, for example) and call this method without even knowing the actual type of the object:

```
Vector<Person> people  = new Vector<Person>();
people.add(new Student("Aliya"));
people.add(new Person("Aizhan"));
people.add(new Employee("Maral"));
for(Person p : people)
	System.out.println(p);
```

You see, the vector *people* has various types of objects, and you can use *foreach* operator to traverse over the list without calling *toString*() separately for each type. Note that in this context *p* is equivalent to *p.toString()*, since

toString() method is used to describe the object, provide its string representation. The output of the code above will be the following:

```
I am Aliya, student at KBTU
I am Aizhan, person at KBTU
I am Maral, employee at KBTU
```

In this subsection we introduced the idea of polymorphism. The key issue you need to take from this subsection is, by virtue of polymorphism, a method call, such as *obj.toString()*, can have different behaviors depending on the type of object, *obj*, on which it is called. So, the main advantage polymorphism provides is that it enables multiple objects of different subclasses to be treated as objects of a single parent class, automatically selecting the corresponding method implementation to apply to a particular object based on the subclass it belongs to. This makes your code easier for you to write and easier for others to understand

4.7 Type conversion

If you think of any subclass/superclass, e.g., *Person* and *Employee* , it can be easily seen that it is always possible to convert a subclass to a superclass - *Employee* is also *Person*, for sure. Due to this reason, explicit casting can be omitted. For example,

```
Circle myCircle = myCylinder;
```

is equivalent to:

```
Circle myCircle = (Circle)myCylinder;
```

Nevertheless, explicit casting must be used when casting an object from a superclass to a subclass. You need to understand that this type of casting may not always succeed.

```
Cylinder myCylinder =(Cylinder)myCircle;
```

You already know about type hierarchy. Classes which are higher up the type hierarchy are said to be *wider*, or *less specific* than the types lower down the hierarchy. Similarly, lower types are said to be *narrower*, or *more specific*.

Widening conversion refers to assigning a subtype to a supertype. This type of conversion can be checked at compile time. In contrast, *Narrowing conversion* means to convert a reference of a supertype into a reference of a subtype. With this type of conversion, a programmer must explicitly convert the object by using the explicit *cast* operator - a type name within parentheses, before an expression. Let's consider some examples on widening and narrowing conversion, to make the distinction between them clearer.

- **Widening conversion** (It is not necessary to provide cast operator and it's a safe cast):

```
String str = "test";
Object obj1 = (Object)str;  //ok
Object obj2 = str;     // ok
```

- **Narrowing conversion** (cast operator must be provided and it's an unsafe cast)

```
String str1 = "test";
Object obj = str1;
String str2 = (String)obj; // ok
Double num = (Double)obj; // no
```

You can ask – why we need to create hybrid instances, why not just create a usual object of a Person or Student, for example. The answer will be clear after the following illustrative example-analogy from our real life. Some student wakes up in the morning, goes to metro station and pays in 80 tenge for the metro service, like any Person. We don't care in this context, who is that person – student, manager or artist. When he enters KBTU, he needs to show the identification card, starting from this moment his declared class becomes *Student*. Later he studies Mathematics in university like a *Student*. It means that the student's behavior is directly related to the situation.

The next example is more challenging. Before moving on further, make sure you understand why particular errors happened at lines 17, 19, 20, 21 and no errors were produced on other lines. In the example followed, *Student* is a subclass of *Person*. Obviously, student, besides having the field *name* inherited from its parent *Person*, has a field *studentNumber* and corresponding get/set methods for it.

```
public class Test {
    static Person[] p = new Person[10];
    static
    {
            for (int i = 0; i < 10; i++) {
               if(i<5)
                  p[i] = new Student();
               else
                  p[i] = new Person();
            }
    }

    public static void main (String args[]) {
       Person o1 = (Person)p[0];
       Person o2 = p[0];
       Student o3 = p[0]; // line 17, comp-on error
       Student o4 = (Student)p[0];
       Student o5 = p[9]; // line 19, comp-on error
       Student o6 = (Student)p[9]; // line 20,runtime error
      int x = p[0].getStudentNumber(); //comp-on error
    }
}
```

So, first 5 objects in an array are hybrid - their **declared** class is *Person*, but the **actual** one is *Student*. The remaining 5 objects are pure persons. Now let's look what will happen if you attempt to compile the code above:

```
%> javac typeTest.java
typeTest.java:17 incompatible types
found     : Person
required : Student
                Student o3 = p[0];
                                  ^
typeTest.java:19 incompatible types
found     : Person
required : Student
                Student o5 = p[9];
                                  ^
typeTest.java:21: cannot resolve  symbol
symbol   : method getStudentNumber ()
location:  class Person
                  int   x   =   p[0].getStudentNumber()
^
3 errors
```

Why we do meet such errors? Concerning the error at line 17. Remember, you cannot convert a superclass to subclass. Declared instance of p[0] is *Person*. So, you can't convert it to *Student* instance. Why? Because the actual class of p[0] might be , for example, *Employee*. In other words, not every person is a student. The same story for line 19, because the declared class of p[9] is *Person*, you can't convert it to *Student*. As for line 21, you can't access the field *studentNumber*, because the declared type of p[0] is *Person*, whereas *studentNumber* is a field of *Student* class. Even if you commenting out these problematic three lines, you will have no problems at compilation time, but at runtime, you will get the following errors:

```
%> java typeTest
Exception in thread "main"
java.lang.ClassCastException: Person
                at typeTest.main(typeTest.java:20)
```

In order to understand better, let us consider another simple example. Suppose there is a class *Food*. What kinds of food do you know? Banana, Chicken, Bread. You are absolutely right. All of these are food. Can we say that *Banana* is the *Food*? The obvious answer is yes! Is it possible to say that any *Food* in the world is *Banana*? No, of course! The code sample below demonstrated this in Java:

```
Food food = new Food();
Food banana = new Banana();
Food banana1 = new Banana();
Banana banan = new Food();//WRONG! Comp-on error
food=banana;
banana1=food; //WRONG! Comp-on error
```

Actually, you can test an object's actual class by using the **instanceof** operactor:

```
if ( obj instanceof String)
{
   String str2 = (String)obj;
}

Circle myCircle = new Circle();

if (myCircle instanceof Cylinder)
{
  Cylinder myCylinder = (Cylinder)myCircle;
  ...
}
```

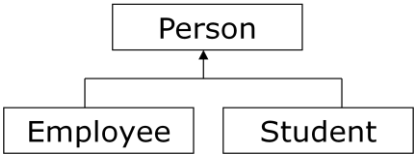
4.8 Abstract classes and methods

In abstract classes, there can be methods which are only declared, but no concrete implementations are provided. They need to be implemented by the extending classes. The simplest possible example is *Person* class and its subclasses – *Employee*, *Student*, *Baby*, *Pensioner*, etc. In this situation, *Person* is an abstract class. An **abstract method** is a method declaration with no body; for example:

```
abstract class Person {
       String name;
       //. . .
       public abstract String getDescription();
       //. . .
}

class Student extends Person {
       private String major;
       //. . .
       public String getDescription() {
              return "a student major in " + major;
       }
       //. . .
}
```

```
class Employee extends Person {
        private float salary;
        //. . .
        public String getDescription() {
                return "an employee with a salary
                        of $ " + salary;
        }
        //. . .
}
```



The rules of thumb you need to remember are:

- Each method which has no implementation in the **abstract** class must be declared **abstract.**
- Besides that, a class having at least one **abstract** method must be declared **abstract.**
- If a subclass of an abstract superclass doesn't implement all the abstract methods of its parent, then the subclass *must* be declared abstract. In other words, in a non-abstract subclass extended from an abstract class, all the abstract methods must have an implementation.

So, when you extend an abstract class, there are two possible situations:

- You leave some or all of the abstract methods undefined. Then the subclass must be declared as *abstract* as well
- Define concrete implementation of all the inherited abstract methods. Just in this case the subclass is no longer *abstract.*

The main rule you need to learn is that an **object of an abstract class can NOT be created**. However, it is still allowed to declare object variables of an abstract class is still, but such a variable can only refer to an object of a non-abstract subclass, e.g:

```
Person p = new Student( );
```

Now let's look at the example of *Food*. What could be the *Food*? It could be tasty or tasteless, sweet or bitter, salty or acidic. Each meal has a

taste, but each has its OWN taste. Let's try to create a method getTaste() in the abstract superclass, and specify it in the subclass.

```
abstract class Food {
        public abstract void getTaste();
}
class Lemon extends Food {
        public void getTaste() {
                System.out.println ("Sour");
        }
}
```

This technique is called overriding, as you already know.

Key issues you need to keep in mind is that the abstract class cannot be instantiated and should be extended and implemented in subclasses. The abstract method represents just a method signature without implementation.

An abstract class cannot be instantiated using the *new* operator, but you can still define its constructors, which are invoked in the constructors of its subclasses.

A subclass can override a method from its superclass to declare it *abstract*. This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass (2). In this case, the subclass must be declared abstract. It is of interest to note that although you can't create instances of abstract classes, you can still define constructors for abstract classes! Sounds strange. Really, we use constructors in order to create objects, but if we are not able to create abstract class instances, then why do we need constructors there? The answer is pretty simple – to call them from subclasses via *super*. An illustration for this is provided in the example below:

```
abstract class Point {
        private int x, y;
        public Point(int x, int y) {
                this.x = x; this.y = y;
        }
        public void move(int dx, int dy){
                x += dx; y += dy;
                plot();
        }
        public abstract void plot();
}
```

So, we have an abstract class Point with position specified by *x* and *y*. It

has methods to move and plot the point (abstract one, without implementation). The method *plot()* is abstract, since we yet don't know the details necessary for drawing – style, color, etc. Let's define it in a subclass **ColoredPoint**:

```
class ColoredPoint extends Point {
        private int color;
        public ColoredPoint(int x, int y, int color){
                super(x, y);
                this.color = color;
        }
        public void plot() {
                // code to plot a SimpleColoredPoint
        }
}
```

Class *ColoredPoint* demonstrates the call of the constructor of abstract *Point*. It is very useful, since it allows to avoid code duplication and ensure consistency among common behavior of subclasses.

Sometimes you might need to forbid the inheritance option of a class you define. In this case, you need to use the *final* keyword. The **final** class cannot be extended, the *final* variable is a constant:

```
final static double PI = 3.14159;
```

The *final* method cannot be modified (overriden) by its subclasses.

4.9 Object class

Just in the same way as humans can be classified as mammals, any subclass in OOP can be generalized as a particular collection of characteristics and behaviors of any of its ancestors. In order to enable the polymorphic behavior among various objects, we need some kind of generalization among varied implementations.

In previous chapter you were already introduced the concept of the **Object** class, which is the root of all Java classes: absolutely every class in Java extends **Object.** This class provides a number of utility methods, listed below.

- **equals()** - returns whether two object references have the same value

```
public boolean equals(Object other){
            if (other == null) return false;
            if (other == this) return true;
            if (!(other instanceof Person))return false;
            Person otherPerson = (Person)other;
            if(this.name.equals(otherPerson.name))
                      return true;
            return false;
        }
```

So, inside of the equals method you just needs to check the parameters used to check the objects for equality, e.g. two persons are equal if their names are equal. Note that this method must accept the parameter of type *Object*, since we want to override the method, not overload it. The usage can be the following:

```
if(somePerson.equals(anotherPerson)) {

}
```

Note that *somePerson* and *anotherPerson* are instances of Person class.

- **clone()** - returns a clone of the object. Sample implementation is provided below.

```
public Object clone() throws
                    CloneNotSupportedException{
        Employee cloned = (Employee)super.clone();
        return cloned;
}
```

Cloning is a very tricky and hot topic. We will talk more on cloning in the following Chapter, "Interfaces", in the part devoted to *Cloneable* interface.

- **getClass()** - return the run expression of the object's class, which is a Class object. Using this, you can check the actual type of the object:

```
if(p.getClass().getName().equals("Person")) {...}
```

At first glance, you might think this method is identical to using

instanceof operator. However, there exist an important difference between them: *instanceof* tests whether the thing is an instance of the type or some subtype. In contrast, *getClass()* tests whether the types are identical.

- **toString() -** return a string representation of the object
    - The toString() method returns a string representation of the object.
    - The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and the address (abc100) where the object is stored in memory, for example, Student@abc100.

We have already had a look at toString() implementation:

```
public String toString(){
        return "I am "+name+", employee at KBTU";
}
```

4.10 Design hints for inheritance

There is a number of simple rules you need to follow in order to make the architecture of your application flexible, well-defined, secure and extensible. Keep in memory these guidelines:

1. Hide private data and private methods.
2. A property that is shared by all the instances of the class should be declared as a class property. That is, it should have a **static** modifier**.**
3. If you need, your class might have various constructors, but try to always provide a public default constructor. Moreover, you should override the **equals()** method and the **toString()** method defined in the **Object** class whenever possible.
4. Try to choose informative names for your variables and methods, and follow consistent styles. It will make your code more readable.
5. Remember, that a class should describe a single entity or a set of similar operations.
6. It is required to place common operations and fields in the superclass.

7. Use inheritance to model a **IsA** relationship, do not replicate the code!
8. Although Inheritance brings many advantages, don't use it unless all inherited methods make sense
9. Don't change the expected behavior in case you override a method
10. Use polymorphism, not type information. For example, consider a code provided below:

```
if (x is of type1)
       action1(x);
else if (x is of type2)
       action2(x);
```

Before writing such a code, ask yourself - *do action1 and action2 represent a common concept?* If so, make the concept a method of a common superclass or interface of both types, and then just call *x.action().*

4.11 Detailed example

Consider an application that lets a user to draw various shapes on a canvas, like circles, ovals, triangles, squares, rectangles, etc. At a design stage we can identify the following classes in the system:

- **Canvas** — represents the 'drawing area'
- **Color** — to represent different colors for our shapes
- **Shape** — *abstract* class used to represent the drawn shapes, with the following subclasses:
  - **Circle**
  - **Rectangle**, etc.

  Abstract **Shape** has the following fields and methods:
  - **Color** color
  - **Point** anchor - some location that represents the shape position on the drawing canvas e.g. the centre of a Circle or the top-left corner of a Square, etc. So, we use it to locate the shape
  - **move**(int dx, int dy) — changes the position of a shape on the canvas by dx, dy.
  - **draw**(Canvas c) — draws the shape on the canvas

Application needs to somehow store all the drawn shapes, in an array o r Vector:

```
Vector<Shape> shapes;
```

When you refresh the canvas, you need to draw all the shapes. Polymorphism allows us to draw shapes in a *generic* way, it's fantastic:

```
for (int i=0; i < shapes.length; i++) {
      shapes.get(i).draw(theCanvas);
      }
```

Nevertheless, actually drawing the shapes is not *generic*, for example, the code for drawing a *Circle* will be very different from the code used to draw a *Square*, and so on. What we need to do is to provide an abstract method in a *Shape*, which will be overridden specifically in its subclasses *Circle*, *Triangle*, *Square*, etc.

```
abstract class Shape {
      private Color color;
      private Point anchor;
      public void move(int dx, int dy) {
      anchor.x += dx;
      ancnor.y += dy;
      }
      public abstract void draw(Canvas c);
      }

class Circle extends Shape {
      private int radius;
      public void draw(Canvas c) {
            // code to draw the circle
      }
}
```

The beauty of that is that you don't care which particular shape you are drawing – circle, square or rectangle! You just provide **draw**() method for each of them, and call it using superclass *Shape* reference (but the actual class is *Square*, *Circle*, etc.).

Note that the **move**() don't need to be specifically implemented for all subclasses of Shape – for all of them moving means just the change of anchor point. That is why we implement **move**() in a superclass *Shape*. So, out abstract class Shape has abstract and non-abstract methods as well.

The method Circle.draw() overrides the method Shape.draw(). In case shapes[i] happens to be an instance of Circle, then

```
shapes.get(i).draw(theCanvas);
```

will execute the code in Circle.**draw**(). Similarly, if i-th *Shape* happens to be an instance of *Square*, then the Square.**draw**() code is executed. As we already know, such run-time determination of which code to execute is called *dynamic binding*.

Polymorhism and dynamic binding allow us to avoid horrible code like the one below:

```
if (shapes.get(i) instanceof Circle) {
        Circle c = (Circle)(shapes.get(i));
        // code to draw the circle
} else if (shapes.get(i) instanceof Square) {
        Square s = (Square)(shapes.get(i));
        // code to draw the square
} else // etc., etc.
```

This is useful example of how inheritance, abstract classes and dynamic binding enable you to write polymorphic code - code working in a generic way for a number of classes.

KEY CONCEPTS YOU NEED TO GRASP FROM THE SECTION

Inheritance chain *Polymorphism* *Dynamic Binding* *super* *Superclass* *Subclass* *Overriding* *Widening Conversion* *Narrowing Conversion* *Declared Class* *Actual Class* *Abstract class and method* *toString()* *equals()*

TESTS

1. Which of the following classes might extend a class Order?

a) Letter b) Transaction c) OnlineOrder d) Chaos

2. Polymorphism allows you to:

a) Hide information

b) Use objects of subclasses in any code designed to work with the objects of their superclass

c) Implement only one idea to eliminate duplication of data

d) Use objects of subclasses in any code

3. When an object has many forms, it can be reffered to as:

a) Polymorphic b) Encapsulated

c) Scalable d) Reused

4. Suppose you want subclasses in any package to have access to members of a superclass. Which is the most restrictive access that accomplishes this objective

a) Public b) Protected

c) Package d) Private

5. What is the principle of the OOP should be used to replace the structure if-then-else in this code:

```
if (animal.IsCat()) { /* code*/ }
else if (animal.IsDog()) { /* code */ }
else if (animal.IsKoala()) { /* code */ }
. . .
else if (animal.isMouse()) { /* code */ }
```

a) Polymorphism b) Inheritance

c) Encapsulation d) Use *instanceof* operator

6. Although we did not discuss this, think why Java refuses support for multiple inheritance?

a) Multiple inheritance practically is never used

b) Support of multiple inheritance results in big losses of productivity

c) Multiple inheritance requires much more difficult algorithms

d) Because of ambiguity of a choice of behavior in case super classes of some class contain methods with identical signatures

7. A class derived from another class in called:

a) Superclass b) Nested

c) Subclass d) Parent

8. Animal a = new Cat(). Animal is _____ class, while Cat is _____ class

a) Actual, Declared b) Declared, Actual

9. For what type of conversion it is not necessary to provide cast operator and it's a safe cast?

a) Widening conversion b) For both of them

c) For none of them d) Narrowing Conversion

10. Which of the classes below are most likely to be abstract?

1) Book
2) Shape
3) Transport
4) Teacher

a) 1, 2 b) 2, 3 c) 3, 4 d) 2, 4

PROBLEMS

1. Consider the following class definitions and think what would be output of the code segment:

```
class A {
        public void method ( ) {
                System.out.println("A") ;
        }}
class B extends A {
        public void method ( ) {
                System.out.println("B") ;
        }}
A a = new A( );
a.method( );
a = new B( );
a.method( );
B b = new B( );
b.method( );
```

2. For the child class *B* defined in the previous exercise, modify its *method*() so that it invokes A's version of *method*() before printing out "B".

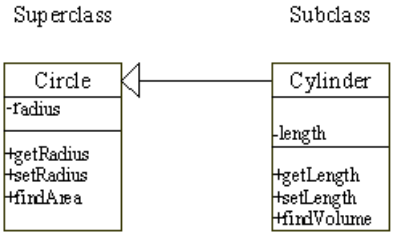
3. Suppose you have a class Person and Students extending it (as in the examples from this chapter).

```
Person p = new Person();
Student s = new Student();
Person ps = new Student();
```

Which of the following operations are legal and why? Which are not legal and why?

```
p = ps;
p = s;
s = p;
s = ps;
ps = s;
ps = p;
ps.getStudentNumber();
```

4. Create classes according to the following UML diagram. Note that '+' means *public*. '-' means *private*.



5. What is the output of the following code segment?

```
class T {
       T() {
              System.out.println("t");
       } }
class X extends T {
       X() {
              System.out.println("x");
       } }
class OrderOfConstruction {
       public static void main(String args[]) {
              X test = new X();
       } }
```

6. Create class *Animal* and a derived class of *Animal* at your choice (*Cat*, *Dog*, *Crocodile*, etc…). In a subclass (or derived class) demonstrate :

- The methods *overriding* and *overloading* of the base class methods.
- The use of *super*() keyword with and without parameters.

7. Create the abstract class for 3D shapes, e.g. *Shape3D*, having *volume*(), *surfaceArea*() (add other methods at your choice!). Then create data types *Cylinder*, *Sphere*, *Cube* extending this class.

8. Create a class called *Employee* whose objects are records for an employee. This class will be a derived class of the class Person (MUST contain *equals* and *toString* methods).

An employee record has an employee's name (inherited from the class *Person*), an annual salary represented as a single value of type double, a year the employee started work as a single value of type *int* and a national *insuranceNumber*, which is a value of type *String*. Inside this class you need to override *toString* and *equals* methods of the Person class.

Your class should have a reasonable number of constructors and accessor methods. Then create a class *Manager* extending *Employee*, each manager has a team of Employees and can get a bonus. You need to override *toString* and *equals* methods. Write

another class containing a main method to fully test your class definition.
Advice: Use super() keyword whenever possible.

9. Create a data type for chess pieces. Inherit from the base abstract class *Piece* and create subclasses *Rock*, *King* and so on. Include a method *isLegalMove(Position a, Position b)* that determines whether the given piece can move from a to b.

Then make a class *Board* and some test class in order to fully imitate chess game. Think of how you will store the current state of the game, take moves from user, drawing the board on a console, checking for illegal moves, etc.

10. When electricity moves through a wire, it is subject to electrical friction or resistance. When a resistor with resistance R is connected across a potential difference V, Ohm's law asserts that it draws current I = V / R and dissipates power V^2 / R. A network of resistors connected across a potential difference behaves as a single resistor, which we call the equivalent resistance.

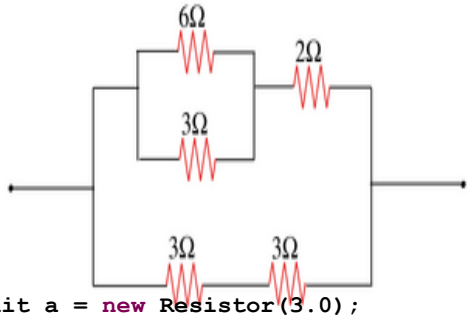
You should have: an abstract superclass ***Circuit*** that encapsulates the basic properties of a resistor network. For example, each network has a method *getResistance* that returns the equivalent resistance of the circuit.

```
public abstract class Circuit {
    public abstract double getResistance();
    public abstract double getPotentialDiff();
    public abstract void applyPotentialDiff(double V);

    public double getPower() {
                ///your code
        }

    public double getCurrent() {
                ///your code
        }
    }
```

A series-parallel resistor network is either **(i)** a single resistor or **(ii)** built up by connecting two resistor networks in series or parallel. So create three ***Circuit*** class subclasses ***Resistor***, ***Series***, and ***Parallel***. Your goal is to be able to compose circuits as in the following code fragment, which represents the circuit depicted below.




```
Circuit a = new Resistor(3.0);
Circuit b = new Resistor(3.0);
Circuit c = new Resistor(6.0);
Circuit d = new Resistor(3.0);
Circuit e = new Resistor(2.0);
Circuit f = new Series(a, b);
Circuit g = new Parallel(c, d);
Circuit h = new Series(g, e);
Circuit circuit = new Parallel(h, f);
double R = circuit.getResistance();
```

The class ***Resistor*** contains a constructor which sets the resistance in Ohms and an accessor method to return it. It also has a private field *pottentialDifference* and get/set mehods to it.

The class ***Series*** contains a constructor which takes two resistor circuit objects as inputs and represents a circuit with the two components in series.

The class ***Parallel*** is almost identical to ***Series*** except that it uses the reciprocal rule instead of the additive rule to compute the equivalent resistance.

Note: The potential difference across each section depends on whether the circuit is series or parallel. For a parallel circuit, the potential difference across each branch is equal to the potential difference across the whole parallel circuit. For a series circuit, first find the current (I) from Ohm's law I = V/R, where V is the potential difference across the series circuit and R is its total resistance. The potential difference across each resistor is equal to the current times the resistance of that resistor.

11. Suppose you are given the following class ***Account***:

```
class Account{
 private double balance;  //The current balance
 private int accNumber;  //The account number
 public Account(int a){
        balance=0.0;
        accNumber=a;
 }
 public void deposit(double sum) { ,,, }
 public void withdraw(double sum) { ,,,}
 public double getBalance(){,,,}
 public double getAccountNumber(){,,,}
 public void transfer(double amount, Account other){
 }

 public String toString() {
        ,,,
 }
 public final void print()
 {
        //Don't override this,override the
  // toString method
        System.out.println( toString() );
 } }
```

Finalize the ***Account*** class and using it as a base class, write two derived classes called ***SavingsAccount*** and ***CheckingAccount***.

A ***SavingsAccount*** object, in addition to the attributes of an ***Account*** object, should have an interest rate variable and a method which adds interest to the account. A ***CheckingAccount*** object (here there is a charge for each transaction), in addition to the attributes of an ***Account*** object, should have a counter variable, that will store the number of transactions done by user, and variable FREE_TRANSACTIONS – number of free transactions. Here you also will have a method *deductFee*(), that withdraws money for made transactions from account (suppose there is $0.02 for each transaction - withdraw or deposit). Ensure that you have overridden methods of the Account class as necessary in both derived classes.

- After that create a ***Bank*** class, an object of which contains a Vector of ***Account*** objects. Accounts in the Vector could be instances of the Account class, the **SavingsAccount** class, or the **CheckingAccount** class. Create some test accounts (some of each type).
- Write an *update* method in the bank class. It iterates through each account and deposits/withdraws money from accounts. After that Savings accounts get interest added (via the method you already wrote) and CheckingAccounts get fees deducted .
- The ***Bank*** class requires methods for opening and closing accounts.

Think, what kind of modifications we can make on a class Account?

12. Create an abstract class *Shape* with fields *color*, *locationX*, *locationY* and abstract methods for drawing a shape a) on a certain location b) on a certain location using certain color. Then create *Circle* class and realize those methods.
Hint : b should be an overloaded method of a.

5 Interfaces

"Walking on water and developing software from a specification are easy if both are frozen."
Edward Berard

Interfaces represent an extremely useful tool in object-oriented programming that many inexperienced programmers underestimate. In order to increase programming productivity, developers want to be able to reuse software components in multiple systems. Interfaces provide us with the way to separate the reusable part of a computation from the implementation parts that vary in each reuse scenario. The reusable part simply invokes methods of an interface, implemented by a certain class. In order to produce a different application, you simply plug in another class that implements the same methods. In such a way, the program's behavior varies according to the class that was plugged in – this refers to *polymorphism*, or *dynamic binding* (already discussed in Chapter 4).

Main functionality provided by interfaces includes:

- Reveal an object's programming interface (i.e. functionality of the object) without revealing its implementation. This represents the concept of encapsulation, as we already know.
- The ability to have unrelated classes implement similarmethods (behaviors). It is particularly very useful when one class is not a sub-class of another.
- To model multiple inheritance. As we know, a class can extend only one class, while it can implement multiple interfaces.

Interfaces exhibit polymorphism as well, since program may call an interface method, and the proper version of that method will be executed depending on the type of object instance passed to the interface method call. As an illustration, consider the following generic max method:

```
class Max{
       // Return the maximum between two objects
  public static Comparable max(Comparable o1,
                               Comparable o2){
              if (o1.compareTo(o2) > 0) return o1;
              else   return o2;
       }
}
```

In this example, *Comparable* is an interface, so instances of any class implementing this interface can be passed on to the method *max* in order to find the maximum among them.

5.1 Interfaces : Basic Concept

Interfaces are specifications for a number of various implementations. Interfaces are used to define a contract for how you interact with an object, which is not dependent on the underlying implementation. The objective of a professional object-oriented developer is to separate the interface from the hidden and very often (in particular situations) not needed details of the implementation.

As we know, an *abstract class* might have data in the form of instance variables, non-abstract and abstract methods. **Interface** can be referred to as an abstract class with only static final instance variables and all methods being abstract. So, an **interface** is a kind of *specification*, for a list of methods that a class implementing the interface must provide the implementation to. In this case, it is obligatory to implement all methods, or make class abstract otherwise. In short, you can think of an interface as of an abstract class having just abstract methods.

It is important to note that all methods in an interface are abstract and public by default. As a rule, it is not required, and in fact redundant to declare a method in an interface to be abstract or public, since implicitly they are public and abstract, whatever you will do.

It is less common to define some data in the interface. If there are data fields defined in an interface, then they are by default defined to be public, static, and final (8).

In simpler words, an **interface** is a way to describe what classes should do, without specifying how they should do it. For example, in Java we have an interface *Comparable* with a single method *compareTo*(). This interface need to implemented by classes whose objects need to be

comparable, that is, for objects *a* and *b*, we need to be able to say whether a>b, a<b or a=b holds. This interface looks like this:

```
public interface Comparable
{
        int compareTo(Object otherObject);
}
```

Comparable interface requires that any class implementing it must provide an implementation for a **compareTo()** method, and this method must take an *Object* parameter and return an integer.

As you can see from the example above, the interface declaration consists of a keyword *interface*, its name, and the members.

Concerning the modifiers of interfaces, by default, they have a *package* modifier and are all implicitly abstract (it is omitted by convention). You might also assign a *public* modifier to it.

5.2 Interface members

As classes, , interfaces can have inside fields and methods. Interface fields can be only constant variables.

- **Constants**

An interface can define named constants, which are *public*, *static* and *final* (these modifiers are omitted by convention) automatically. Keep in mind that interfaces never contain instant fields. Another rule is that all the named interface constants *MUST* be initialized. Consider an example interface, *Educated*:

```
interface Educated {
        int PUPIL = 0;
        int BACHELOR = 1;
        int MASTER = 2;
        int PHD = 3;
        void setEducationLevel (int level);
        int getEducationLevel();
}
```

Using this interface you can assign a level of education to any object, which class implements it, and be sure of consistency.

- **Methods**

Interface methods are implicitly *abstract* (omitted by convention). So every method declaration consists of the method header and a semicolon. In addition, they are implicitly *public*. No other types of access modifiers are allowed for methods. Also, members of the interface can't be neither *final*, nor *static*. A simple example is provided below:

```
void setEducationLevel (int level);
```

5.3 Implementation Procedure

There are two steps to make a class implement an interface:

1) Declare that the class intends to implement the given interface by using the **implements** keyword, for example:

```
class Employee implements Comparable {
        . . .
        }
```

You can encounter all interfaces class implements using a comma.

2) Provide definitions for all methods in the interface:

```
public int compareTo(Object otherObject) {
        Employee other = (Employee) otherObject;
        if (salary < other.salary) return -1;
        if (salary > other.salary) return 1;
        return 0;
}
```

Later on, you can use Java's API to sort the array, list, vector (any Collection) of objects, whose class implements this interface, as simple, as that:

```
Vector<Employee> staff = new Vector<Employee>();
staff.add(new Employee("Vasya", "Pupkin", 3000));
staff.add(new Employee("Ivan", "Ivanov", 2000));
Collections.sort(staff);
```

Collections is a class providing basic and advanced utility methods for all collections implementing the *Collection* interface. We will talk more about collections in the subsequent chapters.

Remember that in case a class leaves any method of the interface undefined, the class becomes abstract class and must be declared *abstract*.Another important thing to highlight is that a single class can implement multiple interfaces. Just separate the interface names by comma, like this:

```
class Employee implements Comparable, Cloneable {
        . . .
        }
```

Keep in mind that interfaces are not classes. Hence you can never use the *new* operator in order to instantiate an interface:

```
public interface Comparable {
        . . .
}
Comparable x = new Comparable( );//error!
```

Nevertheless, you can still declare interface variables:

```
Comparable x ;
```

These variables must refer to an object of a class that implements the interface:

```
class Employee implements Comparable {
        . . .
}
x = new Employee( );
```

5.4 Extending interfaces

As we already mentioned , the peculiar characteristics of interfaces that distinguishes them from abstract classes is support for multiple

interfaces implementation by 1 class. Besides that, an interface can extend more than one interface, so there can be multiples superinterfaces for one subinterface. For example, the interface below combines the methods defined in *Serializable*(later you will see that *Serializable* interface has no methods) and *Runnable* interfaces (one of the core interfaces in Java):

```
public interface SerializableRunnable
             extends Serializable, Runnable {
      . . .
}
```

An extended interface inherits all the constants from its superinterfaces. Pay attention to cases when the subinterface inherits more than one constants with the same name, or the subinterface and superinterface contain constants with the same name — always use sufficient enough information to refer to the target constants. Lets' consider these cases separately:

- *When an interface inherits two or more constants with the same name*

In this case, in the subinterface, you need to explicitly use the superinterface name to refer to the constant of that superinterface :

```
interface A {
      int val = 1;
}
interface B {
      int val = 2;
}
interface C extends A, B {
      System.out.println("A.val = "+ A.val);
      System.out.println("B.val = "+ B.val);
}
```

- *If a superinterface and a subinterface contain two constants with the same name, then the one belonging to the superinterface is* ***hidden.***

The subinterface-version constants can be accessed directly using its name. However, in order to access the superinterface-version constants you

need to provide the superinterface name followed by a dot and then the constant name, for example, X.val:

```
interface X {
        int val = 1;
}
interface Y extends X{
        int val = 2;
        int sum = val + X.val;
}
```

- *If a superinterface and a subinterface contain two constants with the same name, and a class implements the subinterface*

In this case, the class inherits the subinterface-version constants as its static fields.

```
class Z implements Y { }
//inside the class
System.out.println("Z.val:"+val);    //Z.val = 2
//outside the class
System.out.println("Z.val:"+Z.val); //Z.val = 2
```

You can use object reference in order to access the constants:

- subinterface-version constants are accessed by using the object reference followed by a dot followed by the constant name
- superinterface-version constants are accessed by explicit casting

Example below demonstrates this:

```
Z v = new Z( );
System.out.print( "v.val = " + v.val
              +", ((Y)v).val = " + ((Y)v).val
              +", ((X)v).val = " + ((X)v).val );

output: v.val = 2, ((Y)v).val = 2, ((X)v).val = 1
```

Concerning methods, there is a number of issues to pay attention to:

- If a declared method in a subinterface has the same signature as an inherited method and the same return type, then the new

declaration *overrides* the inherited method in its superinterface (3). As we know, this works similarly among superclass and subclass.

- In case there is difference only in the return type, then there will be a compile-time error
- An interface can inherit more than one methods with the same signature and return type. A class can implement different interfaces containing methods with the same signature and return type.
- Methods with same name but different parameter lists are referred to as **overloaded**, as you know.

5.5 Why do we use interfaces?

The usefulness of interfaces goes far beyond simply publishing protocols for other programmers:

- Any function can have parameters that are of interface type.
- Any object of a class that implements the interface may be passed as an argument.

One of the main reasons for interfaces being paid special attention in Java is that you often want to define some operation for objects that all conform to the same contract. So, by virtue of interface, you can define the specification in a very general way, with a guarantee that all objects conforming to this interface will have defined implementations for all the methods.

Again, the main purpose of interfaces is to define a contract. For example, in Java we have *Iterable<E>* interface, that defines the contract between *foreach* operator and ANY thing, that can be iterable, that is, we can traverse it (Vector, HashSet, Array, etc.). So, *Iterable<E>* says: "Whatever you are, as long as you conform to contract (implement *Iterable<E>*), I promise you will be able to iterate over the elements".

Another great example of how interfaces are used in Java is the *Collections* framework. Suppose you create a function that takes a *List* (interface) as a parameter, then it absolutely doesn't matter what to pass in to a function, a *Vector*, *ArrayList* or *LinkedList*. What is more, you can pass that *List* (Vector, ArrayList or LinkedList) to any function requiring a *Collection* or *Iterable* interface instance as a parameter. This significant benefit of interfaces makes functions like *Collections.sort(List list)* possible, regardless of which implementation of *List* is passed.

As another simple example, suppose we define a *Shape* interface with an *area*() method, and so can be sure that any class that implements the *Shape* interface, would define an *area()* method. Practically, it is very useful, since in case of multiple references to objects that implement the *Shape* interface, you can invoke the *area()* method on each of these objects and expect to obtain as a result a value that represents the area of some shape.

For better understanding let's abstract from the programming world and see how interfaces are used in the real world. Look at the picture below. What is common to all objects represented in the picture? After thinking for a while, you might guess that all of them can be used with power socket objects to get the electricity. For that, they need to follow certain standard rules, say, they need to implement *IPowerPlug* interface.

The beauty of that is that the *PowerSocket* doesn't need to know anything about the other objects, except for the fact that they implement IPowerPlug. All objects need Power provided by the *PowerSocket*, so they just implement *IPowerPlug*, and it allows objects to connect to it.

This definitely makes interfaces extremely useful since they provide contracts that objects can use to "talk" to each other without the need to know anything else about each other.

Figure 14. Real-world example of interface usage

To sum up, let's highlight the main benefits interfaces provide:

- Enable different objects to communicate easily
- Hide implementation details of classes from each other
- Support design by specification
- Enable the use of pluggable components (7)
- Allow software reuse

All in all, interfaces aid in bringing not only standardization to your system, but also extensibility, flexibility, maintainability scalability and reusability.

5.6 Marker Interfaces

A *marker* interface (also called sometimes *tag* interface) has neither methods nor constants. In simpler words, empty interface in java is called marker interface. It's main purpose is to allow the use of *instanceof* in a type inquiry. Cloneable interface is such an example.

Examples of marker interfaces include *Serializable* and *Clonnable.* .A marker interface does not contain constants or methods, but it has a special meaning to the Java system. For example, the Java system requires a class to implement the Cloneable interface to become cloneable. *Cloneable* interface has neither methods nor constants, but marks a class as partaking in the cloning mechanism. So, the classes implementing these interfaces don't have to override any of the methods.

The reasonable question that arises is that if such kind of interface doesn't have any fields or methods then why we need it? The answer is pretty simple -they are used to indicate, or signal something to compiler or JVM. In particular, if JVM sees a class is *Serializable* it allows to do some special operation on it. In much the same way, if JVM sees one class implements *Cloneable* interface it performs some operation to support cloning. So, marker interface is used to indicate, or signal something to JVM. So, for example, before an object is serialized and sent send over the wire, Java checks if the class implements *Serializable* interface. If not, an exception is thrown.

With this object in view, another question appears - "Why this signal can't be implemented using a simple flag inside a class?" . Of course, this make sense. But making use of tag interfaces makes it more readable and, what is more important, it also allows to take advantage of *polymorphism* in Java.

We will pay more attention to *Cloneable* and *Serializable* interfaces in subsequent chapters.

5.7 Object Cloning

Before going deep to cloning technique let's clarify, what is the difference between an object copy and an object clone. For this, you need to revise how assignment operator works for reference and primitive types (Chapter 2). When you use assignment operator to create a copy of the object, then, actually, you still have one object and two pointers referencing to it. Therefore, any changes done via one reference affect the state of the object for all references. In contrast, an object clone guarantees that subsequent changes to the new clone object do not affect the state of the original object.

Cloning can be performed via a **clone** method. This method returns a new object whose initial state is a copy of the current state of the object on which *clone* was invoked.

There are three factors in writing a clone method:

- The empty *Cloneable* interface. You must implement it to provide a **clone** method that can be used to clone an object. *Cloneable* interface has neither methods nor constants, but marks a class as partaking in the cloning mechanism. Actually, it looks like this:

```
public interface Cloneable {
        // no code there
}
```

  We don't need an abstract clone() method here, since we already have inside of the Object with partial implementation.

- The **clone** method implemented by the *Object* class. What it does it performs a simple clone by copying all fields of the original object to the new object.
- The *CloneNotSupportedException*, which can be used to signal that a class's **clone** method shouldn't have been invoked

So, now you know that the *Object* class provides a method named **clone**, which performs a simple cloning by copying all fields of the original object to the new object. This type of cloning is usually called *shallow cloning*. Although it works fine for many cases, sometimes you may need to override it for special purpose, when you want to perform *deep cloning*.

Let's compare shallow and deep cloning. Shallow cloning, as it was already said, is a simple field by field copy. For example, suppose you implement an Employee class and you want to provide a method for cloning:

```
class Employee extends Person implements Cloneable{
        double salary;
        Date hireDate;
        ...
        public Employee clone() throws
                                        CloneNotSupportedException{
                return (Employee)super.clone();
        }
}
```

This is an example of conventional shallow cloning. You just call the clone() method defined in an Object root class. The reason why we need to override it in order just to simply call it, is that it is not visible to Object's sublacces. Otherwise we even did not define the *Cloneable* interface and all objects were cloneable by default. However, sometimes we want to forbid object cloning, in this case we just do not implement *Cloneable* interface.

The problem with shallow cloning is that this might be wrong if it duplicates a reference to an object that shouldn't be shared. Consider the following example:

```
class IntegerStack implements Cloneable {
        private int[] buffer; // a stack of integers
        private int top;    // max index in the stack
        // (starting from 0)
        . . .
}
```

Below you can see how the original and copied objects are stored in the memory.

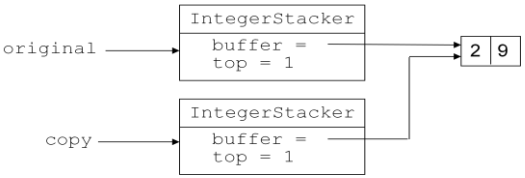


Figure 15. Shallow cloning

From the Figure 15 it can be easily seen that both original and cloned objects have its own copies of primitive type variables (top), but they have just one shared instance of reference type variable - array *buffer*. Although they have separate pointers to this array, it is common for both stacks, so in case one stack changes the array, it will be changed for the other one (remember an example with TV and a couple of remote controls form chapter 2 ?). This reveals the unsuitability of shallow cloning for certain cases – cases when object we want to clone has objects inside.

Is shallow cloning is suitable for Employee class defined above? Of course no, since *Employee* has a field of rederence type, *hireDate* of type *Date*.

For such composite objects containing objects inside you need to implement *deep cloning*: cloning all of the objects from the object on which clone is invoked. With deep cloning implemented, the original and cloned objects will be stored in this way:

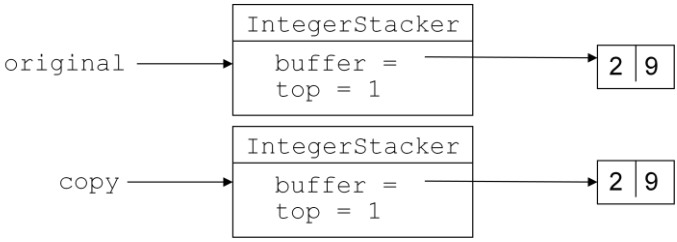


Figure 16. Deep cloning

Let's modify the clone method for Employee to perform deep cloning:

```
public Employee clone() throws
                            CloneNotSupportedException{
            Employee cloned =(Employee)super.clone();
            cloned.hireDate = (Date)hireDate.clone();
            return cloned;
    }
```

With such implementation, both clone and original will have it's own copy of the hire data.

To sum up, let's highlight once more, what needs to be done in order to make object cloneable:

1. Class must implement the *Cloneable* interface. The Java system requires a class to implement the Cloneable interface to become cloneable.
2. Class must redefine the clone method with the *public* access modifier (1).
3. Steps 1,2 are enough for *shallow* cloning. In case you need *deep* cloning, inside of the **clone**() method you need to call **clone**() for each object inside of the object you want to clone.

5.8 Interfaces vs Abstract Classes

Probably, you can ask, why we bother introducing two concepts: abstract class and interface? Why not just using abstract classes, for example, like in the code below:

```
abstract class Comparable  {
       public abstract int compareTo(Object other);
}
class Employee extends Comparable  {
       pulibc int compareTo(Object other) { . . . }
}
public interface Comparable {
       int compareTo (Object other)
}
class Employee implements Comparable  {
       public int compareTo (Object other) { . . . }
}
```

To answer this question, you need to remember the key differences between abstract classes and interfaces:

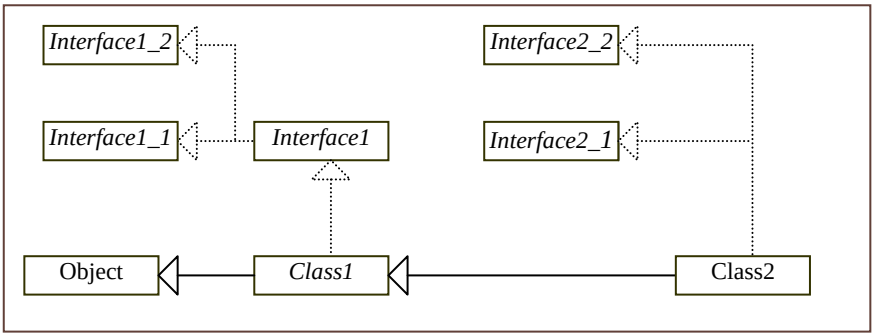
- A class can only extend a single abstract class, but it can implement an unlimited number of interfaces
- An abstract class, besides having abstract methods, can have methods with default implementation (non-abstract methods). In contrast, interfaces are limited to public abstract methods having no implementation.
- In an interface, the data **must** be constants, whereas an abstract class can have **all types** of data - static methods, static data, private and protected methods, etc.

- Abstract class must have at least one abstract method, whereas interfaces can be empty at all (such interfaces are referred to as tag, or marker interfaces, as we studied before).

Since you can inherit multiple interfaces, they often serve as a very useful mechanism to allow a class to have different behaviors in different situations. It is usually a good idea to implement an interface when you need to define methods that are to be explicitly overridden by some subclass. On the other hand, if you want some of the methods implemented with default implementations to be inherited and used by a subclass use an abstract base class instead of an interface.

Again, since all the methods defined in an interface are abstract methods, Java does not require you to put the abstract modifier in the methods in an interface, but you must put the abstract modifier before an abstract method in an abstract class.

Keep in mind the key difference between abstract class and interface - each method in an interface has **only a signature** without implementation, but an abstract class can have **concrete methods**.



From the figure above, think of the following:

Can a class extend multiple classes?
Can a class implement multiple interfaces?
Can an interface extend multiple interfaces?

5.9 Example : Stack

Let's consider the specification of a general stack based on LIFO[10] policy:

```
public interface StackInterface {
        boolean empty();
        void push(Object x);
        Object pop() throws EmptyStackException;
        Object peek() throws EmptyStackException;
}
```

Notice that methods above work with parameter of type *Object*. It is not surprisingly, since a stack is a "container type", it is very common to use the base class for all objects. Since all objects in Java extend the class *Object*, we can always refer to any object in the system using an *Object* reference. Nevertheless, remember that when you pop from a stack, it is required to explicitly type case from the *Object* type to an actual type, so that we can manipulate it respectively.

```
class Stack implements StackInterface {
      private Vector v = new Vector();
      public boolean IsEmpty() {
            return v.size() == 0;
      }
      public void push(Object item) {
            v.addElement(item);
      }
      public Object pop() {
            Object obj = peek();
            v.removeElementAt(v.size() - 1);
            return obj;
      }
      public Object peek() throws EmptyStackException{
            if (v.size() == 0)
                  throw new EmptyStackException();
            return v.elementAt(v.size() - 1);
      }
}
```

[10] LIFO – last in, first out, policy for storing the elements in a data structure

Above you can see the realization of a sample class implementing Stack interface. Keep in mind, that you need to provide implementation to all methods in the interface and conform to their signatures.

5.10 Inner classes

Sometimes you have a class that serves a very limited purpose. In this case, you can declare the class inside the method or class that needs it. Such a class refers to Inner class (sometimes also called as *Nested* class). So, an inner class is any class defined inside another class. Be careful when you use an inner class – in case you might need to use it beyond the scope of a class, it is better to make it publicly accessible.

As it was mentioned, you can declare an inner class inside the method and inside the class itself. The example for both these cases are provided below:

- **Declaring an inner class inside a method**

  Template:

```
class Outer{
        method signature{
                class Inner {
                        methods
                        fields
                }
        }}
```

  Example:

```
public static void main(String[] args) {
                class Pair {
                        public Pair(int x, int y){
                                this.x = x;
                                this.y = y;
                        }
                        int x;
                        int y;
                }
                Pair p = new Pair(2,4);
        }
```

- **Declaring an inner class inside the class**

  Template:

```
class Outer{
        methods
        fields
        assesModifier class Inner {
                methods
                fields
        }
        ...
}
```

  Example:

```
class LinkedList {
        //...
        Node [] nodes;
        class Node{
                Node next;
                int key;
                //...
        }
}
```

Later one, you can use them in the following way:

```
LinkedList list = new LinkedList();
Node n =  list.new Node();
```

It is important to note that methods of the inner class can directly access the members of the outer class.

All in all, try to use inner classes for small tactical classes that should not be visible elsewhere in a program.

5.11 Interfaces & Inheritance. When to use what?

From this chapter you got the idea of how interfaces can be useful to add a level of abstraction to the system. Usually, this is done at design

stage. However, it must be admitted that interfaces are useful for big and middle-sized projects, for small projects is most likely "overkill".

Another challenge is to be able to carefully determine when to use inheritance and when to use interfaces. Stick to the following guidelines, the correct choice will definitely increase scalability and reusability of your code.

- **A *Is a* B – use inheritance**

If it suits to say A is B (where A and B can be some logical elements or concepts), when obviously use inheritance. In this case, your class **is a** subclass of a more generalized class, for example, *HouseCat* (A) inherits from *Feline* (B), since house cat is definitely a feline.

- **A *Has* a B – use member fields**

For example, a *LittleGirl* **has a** car, so clearly, she shouldn't be a subclass of a *HouseCat*, since she **is not a** cat. Usually, in such situations it is best for *LittleGirl* to have a as an instance field:

```
class LittleGirl
{
    int age;
    String name;
    HouseCat pet;
}
```

- **A *performs* B – use interface**

Interfaces should be used in case we have a class or set of classes exposing similar *functionality*, but when there is no clear line of inheritance. So, interface is just a certificate saying "A objects performs this functionality (B)". Going back to our cat and girl, although a *HouseCat* inherits from *Feline*, it might implement *ICanHavePizza* interface. Obviously, a *LittleGirl* also might implement *ICanHavePizza* interface. So, we can have a method serve to which we can pass *LittleGirl*, *HouseCat*, and many other objects, implementing *ICanHavePizza* interface in order to feed them a pizza.

```
public Pizza serve(ICanHavePizza client) {
       ...
}
```

This is useful since *HouseCat* and *LittleGirl* do not have a common

subclass (e.g. Person) to pass in. However, they both perform acts involving pizza.

5.12 Principles of Object-oriented Design

Last several chapters acquainted us with main concepts underpinning Object-oriented programming. It would be great to conclude this chapter briefly focusing on various object-oriented design principles. Most of them should be familiar to you:

- **Divide-and-Conquer Principle**.
  It means that problems are solved by the way of dividing them into several classes, with each of the classes divided into separate methods. The idea of a class hierarchy itself is an application of this principle. Another name of this principle is *modularization*: breaking into pieces. A *module* can be defined in various ways, but generally must be a component or subsystem of a larger system, and operate within that system independently from the operations of the other components (1).
- **Encapsulation Principle.** This principle states that the superclasses should encapsulate those features of the class hierarchy that are shared by all objects in the hierarchy. The subclasses, in turn, encapsulate features that make them distinctive among the other classes in the hierarchy (4).
- **Interface Principle.** By interfaces programmers provide a specification of how various types of related objects interact with each other through the method signatures contained in the interfaces.
- **Information Hiding Principle.** This involved making consistent use of the *private*, *protected*, and *public* qualifiers.
- **Generality Principle.** This principle ensures that as you move down a well-designed class hierarchy, you go from the more general to the more specific features of the objects involved in the system (4).
- **Abstraction Principle.** Designing a class hierarchy is an exercise in abstraction, as the more general features of the objects involved are moved into the superclasses. Similarly, designing a Java interface or an abstract superclass method is a form of abstraction, whereby the signature of the method is distinguished from its various implementations (4). To conform to this principle, you

need to be able to identify key features and ignore the details. Programming bstraction is a mechanism and practice intended to reduce and factor out details so that one can focus on few concepts at a time (1).

- **Extensibility Principle.** This means to override inherited methods and implement abstract methods from either an abstract/non-abstract superclass or an interface. This can be used to extend the functionality of an existing class hierarchy.

KEY CONCEPTS YOU NEED TO GRASP FROM THE SECTION

Interface *Specification* *Comparable* *Serializable*
Cloneable *Marker Interface* *Object Cloning*
Shallow Cloning *Deep Cloning* *Modularization*
Divide-and-Conquer Principle *Generality* *Extensibility*

TESTS

1. All the interface methods are … ?

a) final b) abstract

c) static d) economic

2. Which of the following interfaces are marker interfaces?

a) Serializable, Comparable b) Cloneable, Equatable

c) Comparable, Comparable<T> d) Serializable, Cloneable

3. You can't construct interface objects, but you can still declare interface variables. Based on that, choose the correct answer:

a) x = new Comparable(. . .); x = new Employee(. . .);

b) Comparable x = new Employee(. . .);

c) Comparable x = new Comparable(. . .);

4. What term is used to describe the internal representation of an object that is hidden from view outside of the object's definition?

a) Encapsulation b) Polymorphism

c) Abstraction d) Inheritance

5. What are the main principles of Object-oriented design?

a) Modularity, Encapsulation and Abstraction

b) Hierarchy, Concurrency and Abstraction

c) Usability, Encapsulation and Hierarchy

d) Typing, Concurrency and Abstraction

6. What is true about Marker interface?

a) It is an empty interface b) It has just abstract methods

c) It has a boolean flag d) It has just final methods

7. What is incorrect?

a) Interfaces support multiple inheritance

b) Methods of interface can be static

c) Interface can have public and package modifiers

d) We can declare interface variable

8. Which method is used to compare two objects and how many types of logical output it can produce ?

a) compare(), 3 answers b) instanceof(), 2 answers

c) equals(), 2 answers d) compareTo(), 3 answers

9. What is the default modifier of interface?

a) public b) package

c) abstract d) final

PROBLEMS

1. Consider an interface that you have developed called *DoIt*:

```
public interface DoIt {
       void doSomething(int i, double x);
       int doSomethingElse(String s);
       }
```

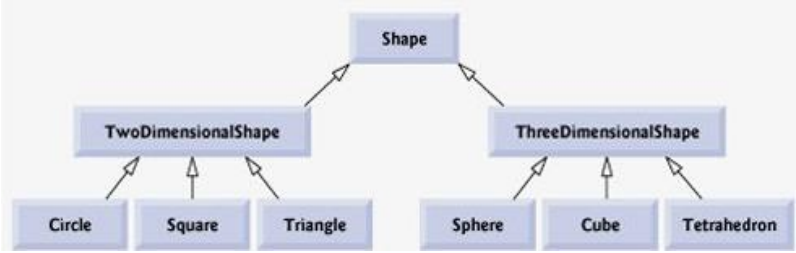
Let's imagine that later on you decided to add one more method to your interface, so that it looks like this:

```
public interface DoIt {
       void doSomething(int i, double x);
       int doSomethingElse(String s);
       boolean isOk(int i, double x, String s);
       }
```

Now, in case you make this change, all classes implementing the old *DoIt* interface having 2 methods will break, since they don't implement all methods of the interface anymore – specifically, *isOk* is not implemented by them. What solution to this problem you can offer?

2. Create an interface *Shape* and then, Implement the Shape hierarchy as shown in figure below.

- Each TwoDimensionalShape should contain:Method getArea in order to calculate the area of the two-dimensional shape.
- Each ThreeDimensionalShape should have methods getArea and getVolume to calculate the surface area and volume, respectively, of the three-dimensional shape.



After that write a program that uses an array of *Shape* references to objects of each concrete class in the hierarchy. The program should

print a text description of the object to which each array element refers. Also, in the loop that processes all the shapes in the array, determine whether each shape is a *TwoDimensionalShape* or a *ThreeDimensionalShape*. If a shape is a *TwoDimensionalShape*, display its area. If a shape is a *ThreeDimensionalShape*, display its area and volume in a meaningful way.

3. When to use an Interface vs when to use an abstract class. For each "when" provide extended example(s) (with class/interface codes).

4. Extend Employee and Manager classes created in problem # 8 of previous chapter.

- Replace field *year* by the field *hireDate* of type *java.util.Date*
- Your classes should implement *Comparable* interface. (Employee1 > Employee2 if its salary is more than the salary of Employee2, the same for managers, but if their salaries are equal, compare by bonus).
- Implement *Cloneable* interface so to be able to clone your objects. Use shallow or deep cloning, as you want.

5. Suppose you have an interface *Moveable*. Think of some interface that can extend it. Implement both interfaces.

6. You need to write a class *MinMax* with a method *minmax* that takes an array of integers as a parameter and returns min and max simultaneously (using one method and one call).
Hint: use inner class

```
public class MinMax {
       static class ??? {
       }
       static ??? minmax(int values[]){
              return ???;
       }
// Test class:
int a[] = {0, 8 , -3, 20};
MinMax m = new MinMax();
// Do something to find min and max
//using instance m of class MinMax.
```

7. Suppose you have two interfaces, *I* and *J*, and a class *C*, implementing *I* interface:

```
interface I{
        ...
}
interface J{
        ...
}
class C implements I{
        ...
}
```

Assume that *i* is declared as follows:

```
I i = new C();
```

Which of the following statements below will throw an exception and why?

```
C c = (C)i;
J j = (J)i;
i = (I)null;
```

What can you propose to determine such kind of errors at compile time?

8. Suppose you have the following interface:

```
public interface Filter {
        boolean accept(Object x);
}
```

Think of a couple of classes that might implement it and provide the corresponding code.

9. Create a class *Date* having fields *year*, *month*, *day*. Implement *Comparable* interface and provide *toString*() method . Then create several objects of *Date* , store them in a *Vector* and sort.

10. Create a class *Mark* having a field *points* (e.g. 95) and a method *getLetter*() (e.g. A-). Implement *Comparable* interface and provide

toString() and *equals*() methods . Then create several objects of *Mark*, store them in a *Vector* and sort.

11. Methods declared in an interface cannot be declared final. Why?

6 UML diagrams

"Perfection [in design] is achieved, not when there is nothing more to add, but when there is nothing left to take away."
Antoine de Saint-Exupéry

"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies."
C.A.R. Hoare

Before writing code for a complex application, you need to design your solution. Although there exist plenty of techniques for that, in most cases, design stage consists of the following tasks:

- **Identifying classes in the system**
  Highlight the nouns in the problem description and prepare a list of nouns. Exclude those ones that do not seem to be reasonable for classes.
- **Identifying the responsibilities of each class**
  First, you can make a list of major functions your system must have. Then, for each task you need to find a class (from the list you obtained at the previous step) that is responsible for carrying out this task.
- **Identifying the relationships among classes**
  At this stage, you need to prepare a class diagram reflecting the relationships between all the classes that you defined previously.

Generally, there are three major parts of a system's model, namely:

• *Functional Model*, which presents the functionality of the system from the user's perspective. Example: *Use Case Diagrams.*

• *Object Model*, which is used to represent the system structure using classes, attributes, operations, and various types of associations among them. Example: *Class Diagrams*.

• *Dynamic Model*, that describes the internal behavior of the system. Example: *Sequence Diagrams.*

In this chapter you are going to learn the types of relationships existing among classes and ways you can use to reflect the design of your system. We specifically concentrate on UML *Class* diagram, just briefly discussing

Sequence and *Use Case* Diagrams. We hope that after reading this chapter you will appreciate the usefulness of UML modeling approach and will be able to easily determine and implement various class diagram relationships that are used in object-oriented modeling.

6.1 Use Case Diagrams

A **Use Case Diagram** is a type of functional diagram that is used to present a graphical overview of the functionality system affords in terms of actors (usually users of the system), their actions - use cases, and any dependencies between those use cases. So, the main aim of a *Use Case Diagram* is to show which system functions are performed by each actor.

It needs to be emphasized that *Use Case Diagrams* are not suited for representing the detailed system design, and can't describe the internal details of operation of a system (4). Instead, they can be useful in facilitating the communication with the future business users of the system, and are particularly helpful to determine the required features the system must have. Hence, *Use Case Diagrams* specify, what the system should do, without specifying how this is to be achieved.

Use Case Diagrams have the following elements: *Actors* and *Roles*. These elements can be connected by various associations. The example of simplest possible Use case diagram is presented below:

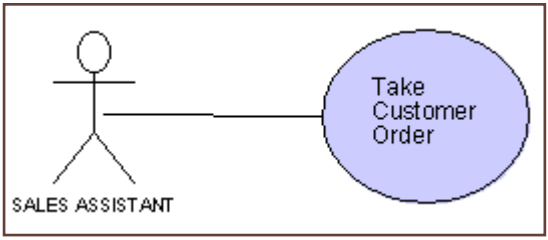


Figure 17. Example on Use Class Diagram

In addition, you can use such connections, as *include* and *extend*. *Extend* corresponds to usual inheritance (transition from general to more specific). *Include* relationship allows us to include one use case into the other. Consider an example below:

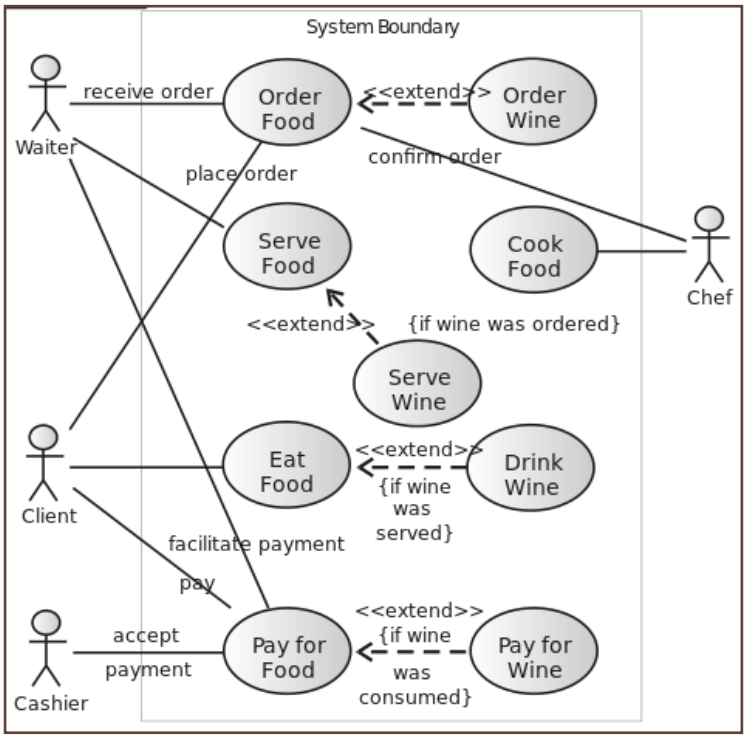


Figure 18. Use Class Diagram – Restaurant System[11]

From the figure above we can see that, for example, "Order Food" use case involves two actors – *Waiter* (receives the order) and *Client* (places the order). In addition, there is an extended use case from "Order Food", which is called "Order Wine".

6.2 Sequence Diagrams

A **Sequence Diagram** is a kind of dynamic diagram illustrating how processes operate with one another. It contains the objects involved in the system scenario and the *sequence* of activations, containing messages which are exchanged between the objects involved in the functionality of the

[11] Source: Wikipedia.org

scenario. The main convenience of a *Sequence Diagram* is that it allows to visually specify simple dynamic scenarios.

Sequence Diagrams are composed of objects, activations and messages (7). You already know about objects. *Activations* (or lifelines – parallel vertical boxes) are different processes that live simultaneously. *Messages* element - (shown as horizontal arrows) – are functional messages exchanged between activations (with the name written above them), in the order in which they occur.

For example, the Figure 19 below depicts the sequence of events that take place in an online shopping system.

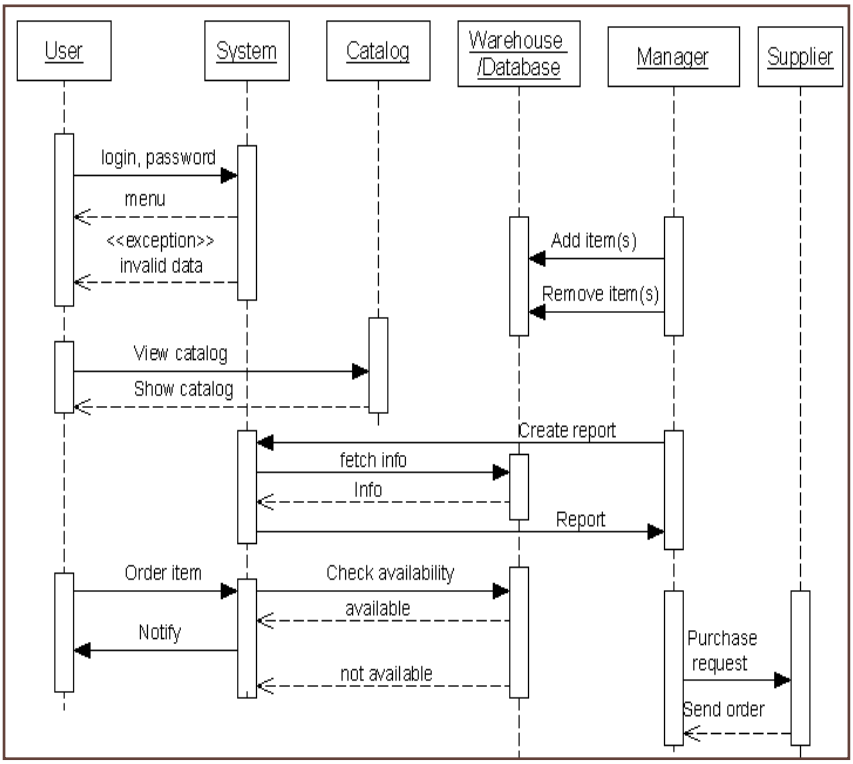


Figure 19. Sequence Diagram – Restaurant System

In the diagram presented in Figure X we have 6 objects: *User, Manager, System, Supplier, Database* and *Catalog*. First process is user validation and it involves 2 objects – *User* and *System*. Particularly, *User* commits the authorization activation, by submitting it's login and password.

System, in turn, processes this information and in the case of correct login and password, user is offered an option menu. Otherwise, exception arises, due to invalid data.

The next process involves *User* and *Catalog* objects. It is simple one: when *User* desires to view the *Catalog*, the list of items for sale is presented to him.

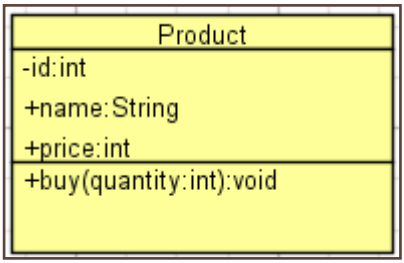
The procedure of ordering the item has a following scenario: user requests a particular item through the system, after that the availability of item chosen is checked in the database. Here there are 2 possible cases: item is either available or absent. In case when there is no such item in the warehouse, possibly, *Manager* will make a purchase request to a *Supplier*. In any case, the *User* is informed about the availability of the item.

Furthermore, *Manager* can generate reports (on order frequency), that process involves 3 objects: *System* (standing between *Database* and *Manager*), database – in order to fetch the information, and the manager himself. In addition, *Manager* can make purchase request to a *Supplier* (for example, after analyzing the report generated). *Supplier* responds by sending the items ordered. The last operations are pretty simple: manager is able to manage items, so he can add/delete them.

6.3 Class Diagrams

Class Diagrams are the most frequently used modeling technique and represents the richest notation in UML.The main designation of a class diagram is pretty simple – to describe the types of objects in the system and the various kinds of relationships that exist among them. This is done through a graphical representation. Obviously, the elements and relationships in the diagram must be static.

As you know, a class is the description of a set of objects with similar attributes (fields) and operations (methods). Visually, a class icon looks like a rectangle divided into three parts: with the class name at the top, attributes in the middle, and operations being in the bottom, as in Figure 20 below:



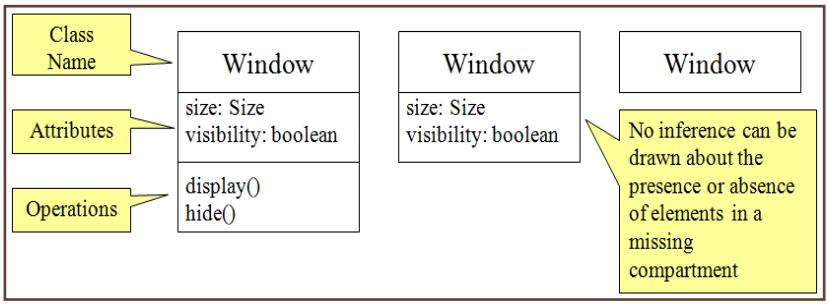


Figure 20. Simple Class Icons

An important thing you need to keep in mind is that you don't need to include all the attributes and methods of a class to a particular diagram (Figure X above, *Window* example). Instead, you just need to list the ones that are helpful in understanding the concept you want to explain by your diagram.

Now let's look deeper at features of attributes and operations.
In simple words, an attribute is property that object of a class have. They are used to describe the state of the object. For example, id, name and price can be typical attributes of the Product class. In most cases, attributes correspond to instance variables, so they are atomic entities having no responsibilities. Attribute syntax on the class icon is the following:

[visibility] name [: type] [= defaultValue]

By the way, class scope attributes (i.e. having *static* modifier) need to be underlined. Let's take for example a class *Note*, having the attributes for author, text and a static type attribute reflecting the total number of notes. The corresponding class icon and code in Java are provided below:

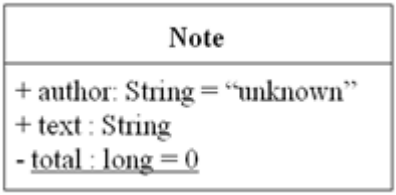


Figure 21. Class Icon for *Note* (Example on Attributes)

```
public class Note
{
  public String author = "unknown";
  public String text;
  private static long total = 0;
  ...
}
```

Now let's turn to operations. UML class diagram operations represent the processes that objects of a class performs. As a rule, operations correspond to class methods. Operations have the following syntax:

[visibility] name ([parameter-list]) [: return-type]

In the syntax above, *parameter-list* is a comma separated list of formal parameters that class has, each of them must be specified using the syntax: *name : type [= defaultValue]*. Class scope operations , as attributes, must be underlined.

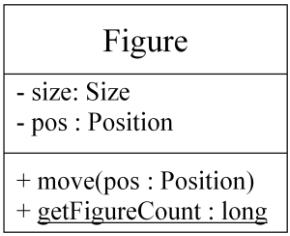


Figure 22. Class Icon for *Figure* (Example on Attributes)

```
public class Figure{
       private Size size;
       private Position pos;
       private static long figureCount = 0;

       public void move(Position pos) { ... }
       public static long
       getFigureCount() { return figureCount; }
       ...
}
```

- **Visibility**

Visibility notation is used to indicate whether an attribute or operation is visible and allowed to be referenced from other classes . UML provides four visibility types, corresponding to four access modifiers (Table X):

Visibility type	Public	Private	Protected	Package
Notation	+	-	#	~

Table 8. Multiplicity Notations

- **Multiplicity**

Sometimes it might be useful to add a logical denotation of how many objects can be aggregated inside of the other object. This is called *multiplicity* and it has the following notations:

Exactly one	**1**
Zero or more (unlimited)	*(0..*)
One or more	1..*
Zero or one (optional association)	0..1
Specified range	2..4
Multiple, disjoint ranges	2,4..6,8

Table 9. Multiplicity Notations

Note that *multiplicity* provides a lower and upper bound on the number of instances. As a simple example, one fleet can include multiple airplanes, while one commercial airplane may contain zero to many passengers.

Apparently, you can't have classes just floating around independently – they must be somehow interrelated. Besides the class icons, another essential element of UML Class Diagrams are relationships. According to UML standard, classes can be related to each other by association, generalization, realization, dependency, aggregation and composition. The table presented below provides a brief description of types of relationships in UML class diagrams:

All in all, class diagrams are very useful at design stage to describe all the classes, packages, and interfaces that make up a system and how these components are connected to each other. In addition, well-defined detailed class diagrams can be used as a source to translate the designed system into

a programming code (2). Almost all modern UML Tools provide this option.

In the following subsections we will consider in details each one of these relationships:

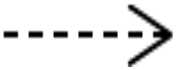
Relationship	Symbol	Line Style	Description
Association		Solid	Very general notation that indicates that two classes have some logical connection
Generalization		Solid	Represents *"is a"* relationship (between the child class and parent class). Similar to Inheritance.
Dependency		Dotted	Depicts *"uses"* relationship. Relation between two classes in which a change in one affects the other class object
Realization		Dotted	Relationship between an *interface* that defines a set of functionalities and a *class*, that realizes this functionality
Aggregation		Solid	Denoted *"has a"* relationship. Objects of one class contain objects of the other class.
Composition		Solid	Strong form of *Aggregation*. *Part* entity can't live independently of its *Owner*

Table 10. Relationships among classes in UML

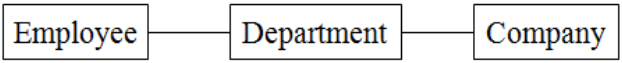
6.4 Association

Early in the design stage it is useful to use a more general type of relationship between classes called association. One class is associated with the other if you can navigate from objects of one class to objects of the

other class (i.e. by accessing an instance field, or performing a database lookup).

Association is a semantic relationship between classes that specifies connections among their instances. It's main intention id to specify that objects of one class are connected to objects of a second (it can be the same class) class.

A typical example is a relationship: "An Employee works for a Company" [12]:



The straight line between classes indicates that object at one end can "recognize" objects at the other end and may send messages to them.

Sometimes it might be useful to clarify the meaning of association by giving it a name and by assigning the corresponding roles played by the class at both ends of the association path. The name is represented as a label placed at the centre along the association line. The name is usually either a verb or verb phrase, while role is usually represented by a noun or noun phrase. It is important to note that specification of a role is mandatory for reflexive associations, in order to avoid confusion. As an illustration for that, consider the following diagram depicting the relationship between university staff, students and courses..

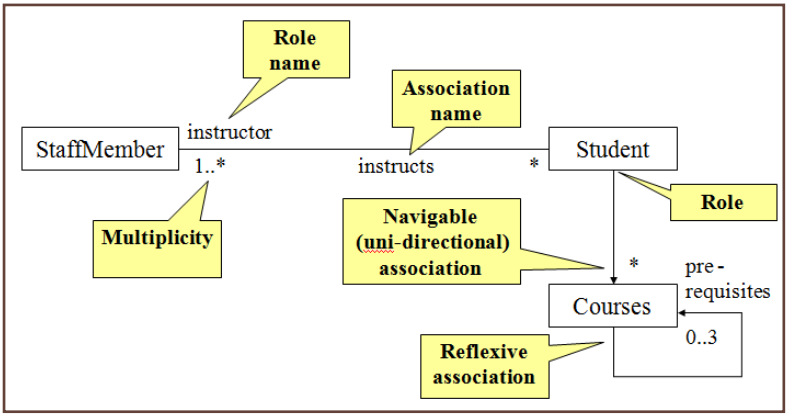


Figure 23. Association relationship example

[12] Note that here and in some of the subsequent diagrams, for the sake of simplicity we do not include the fields and methods inside of a class icon, when not needed

Reflexive association occurs when a class may have multiple functions or responsibilities.

We already mentioned that the UML notation for association is a solid line. In addition, you can optionally add arrows showing in which direction you can navigate the relationship. For example, the diagram below shows that Bank object can navigate to its customer, but it doesn't work the other way round. So, in this particular design, the *Customer* class has no mechanism to identify in which *Bank* it keeps its money.

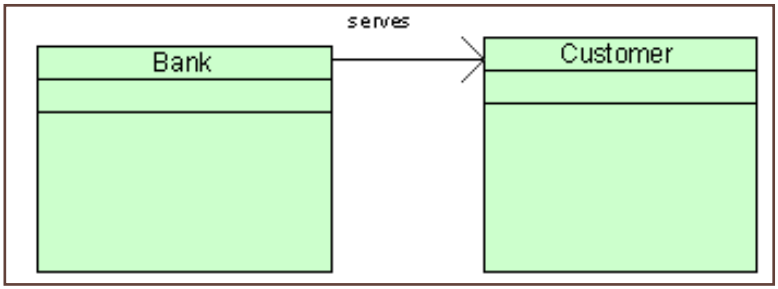


Figure 24. Association between Bank and Employee

6.5 Generalization

Generalization relationship is used to represent inheritance. So, it indicates objects of the specialized class (subclass) are objects of the generalized class (super-class).

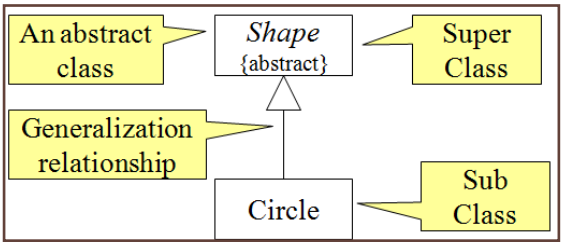


Figure 25. Generalization relationship example

Note that if the class in abstract, you must provide {abstract} tag value and make the name of the abstract class italic.

Furthermore, let's add some details to our classes. It seems reasonable to add abstract draw() method to Shape class and non-abstract draw() to circle:

To depict inheritance in a UML diagram, a solid line from the child class to the parent class is drawn using an unfilled arrowhead.

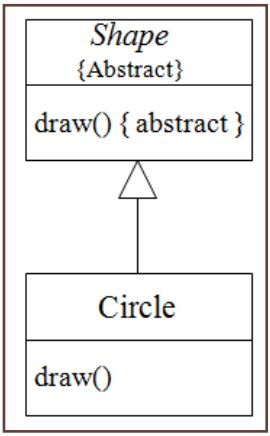


Figure 26. Generalization relationship example (with methods)

The sample code for that diagram is:

```
public abstract class Shape
{
        public abstract void draw();
        ...
}
public class Circle extends Shape
{
        public void draw() { ... }
        ...
}
```

6.6 Dependency

A dependency is a relation between two classes in which a change in one may impact the other class and cause changes , although there is no explicit association between them. A stereotype may be used to denote the

type of the dependency. Example - a class calls a class scope operation of another class.

As the following figure illustrates, a dependency is displayed in the diagram editor as a dashed line with an open arrow that points from the client model element to the supplier model element. By convention, Dependency relationships do not have names.

As an example, consider an e-commerce application, where a *Basket* class depends on a *Product* class, since the Basket uses the *Product* as a parameter for an *addProduct* operation (Figure 27). So, in a class diagram, a dependency relationship must point from the *Basket* class to the *Product* class. This relationship indicates that a change to the *Product* class might require a change to the *Basket* class.

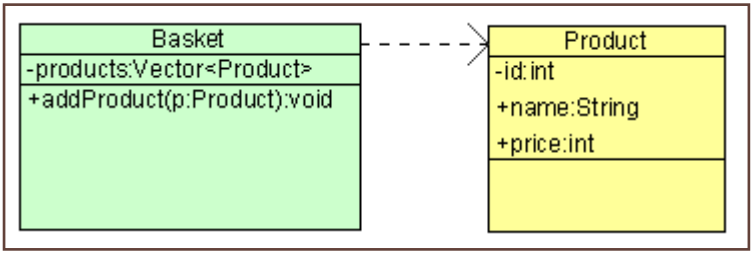


Figure 27. Dependency relationship example

Owing to the fact that a *Dependency* relationship can indicate several different types of relationships, stereotypes (a kind of keyword) are used in order to depict the precise meaning of the dependency. They are usually written at the centre along the relationship line. Most frequently used stereotypes are provided below (7):

- **<abstraction>** - used when elements represent the same concept at either different levels of abstraction, or from different viewpoints.
- **<substitute>** - indicates that one element can take the place of another element
- **<use>, <call>** - indicates that one element requires another element for full its operations
- **<instantiate>** - one element can be a creator of another element

6.7 Realization

A realization relationship indicates that one class implements a behavior specified by another class or interface. For a consistent use of *Realization* relationship, let's recover two main principles in the relationships among classes and interfaces:

- An interface can be realized by many classes.
- A class may realize many interfaces.

For example, *LinkedList* class from java.util package implements the *List* interface from the same package. They together represent the realization relationship (Figure X below):

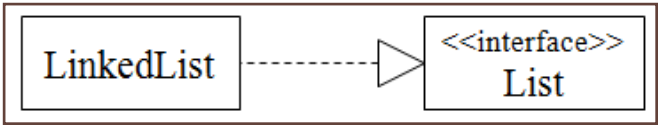


Figure 28. Realization relationship example

The source code corresponding to this UML is:

```
public interface List
{
  boolean add(Object o);
  ...
}
public class LinkedList implements List
{
  public boolean add(Object o) { ... }
  ...
}
```

6.8 Aggregation

An aggregation relationship is a special form of association that models a whole-part relationship between an aggregate (the whole) and its parts. So, it corresponds to "has a", or "is part of" relationships. For example, a Department class can have an aggregation relationship with a Company class, which indicates that the department *is part of* the company.

In this type of relationship, data flows from the whole aggregate to its part. A *part* can belong to more than one *aggregate* . Although an aggregation is closely related to composition, there is an important difference between them - in aggregation relationship *part* entity can exist independently of the aggregate, in contrast to *Composition* (discussed further in this chapter).

It must be admitted that aggregation relationships do not have to be unidirectional. They can also be bidirectional (no arrows at the end).

An aggregation association looks like a solid line with an unfilled diamond at the relationship end, which is connected to the entity representing the aggregate. An example is provided below:

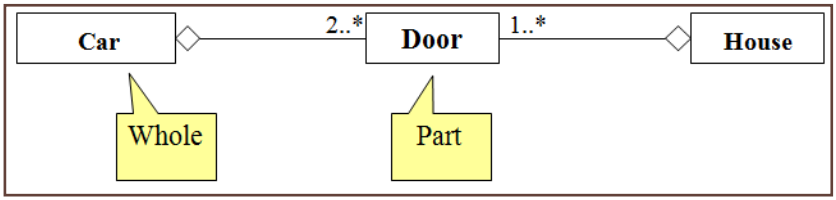


Figure 29. Aggregation relationship example

There is a number of questions you can use to test that you chose the correct type of relationship. For aggregation, they are the following (we use the example from Figure 29 to clarify the questions):

- *Is the phrase "part of" is suitable to describe the relationship?*
  Yes, since a door is *part of* a car

- *Are some operations on the whole are automatically reflected to its parts?*
  Yes. For example, when the car is moving, the door is also moving.

- *Are some attribute values from the whole disseminate to all or some of its parts?*
  Yes. For instance, if the car is red, the door is also red.

- *Is there an asymmetry to the relationship?.*
  For that, you need to check whether *is part of* relationship holds after exchanging the whole and part. It shouldn't hold for aggregation. For example: a door *is part of* a car, but a car *is not a part of* a door.

The Java code for that aggregation relationship is provided below:

```
public class Car {
   private Vector<Door> doors = new Vector<Door>();
   public void addDoor(Door door) { ... }
       ...
}
public static void main(String[] args)
{
       Door door = new Door();
       House house = new House(door);
       Car car = new Car();
       car.addDoor(door);
       ...
}
```

Another example of aggregation is Company – Department relationship:

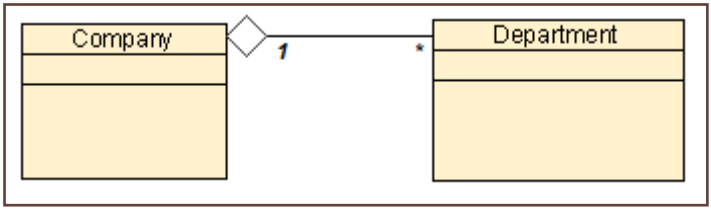


Figure 30. Realization relationship example

Again, of the confusing things about aggregation is that, as almost any relationship, it can be directed. Remember that a directed association indicates that control flows from one classifier to another. In other words, you can navigate from one of the association end to the other (i.e. if you know Company, you can access departments), but not vice versa.

6.9 Composition

Composition relationship is strongly related and very similar to aggregation. Simply speaking, composition is a strong form of aggregation, in which the whole is the sole owner of its part. So, the part object can only

belong to only one whole. It means that the multiplicity at the whole side must be zero or one.

The composite (owner) is responsible for managing the formation and destruction of its parts. Therefore, the lifetime of the part object is dependent upon the whole.

As a rule, in a composition relationship, data flows only in one direction - from the whole classifier to the part. For example, in a credit system of education, a composition relationship might connect a *Student* class with a *Schedule* class, which can be proved by the fact that in case you remove the student, the schedule is also removed.

A composition relationship is illustrated as a solid line connecting two classes with a filled diamond placed at the association end, which is connected to the composite (whole) object.

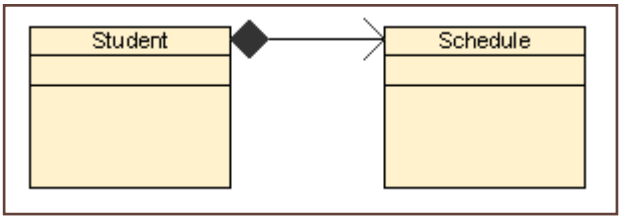


Figure 31. Composition relationship example

It is important to understand that in case a class-container (owner) is destroyed, the part class will also be killed. For example, in case a bag is destroyed, it's side pocket will also do that.

Several more examples of Composition are represented below: between *Circle* class and its center - object of a class *Point* (Figure 32), and between standard computer *Window* and its parts – *Slider*, *Header* and *Panel (Figure 33)*.

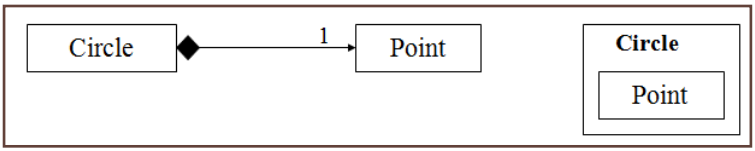


Figure 32. Composition relationship example – Circle and Point

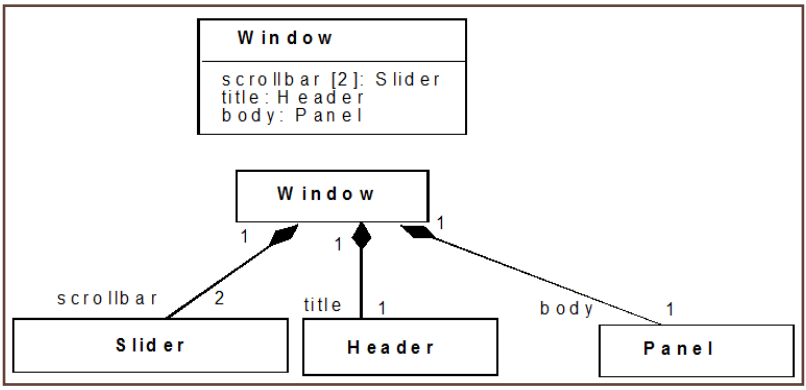


Figure 33. Composition relationship example – Window and its parts

6.10 Important issues – multiplicity, constraints and notes

Constraints and notes are important tools in UML modeling. They can be used to annotate associations, attributes, operations and classes. It must be admitted that constraints are semantic restrictions depicted as expressions.

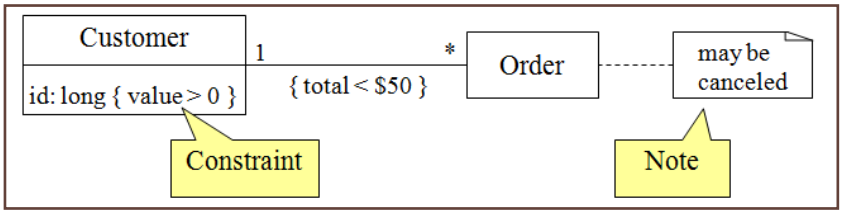


Figure 34. Constraints and notes example

The constraint shown in Figure 34 (id value>0) can be implemented, for example, in the constructor or the setter method for the id field. The note on Figure X represents a note which is connected to a model element. It can also stand independently, and comment on a certain part of a diagram.

6.11 UML Packages

As you already know from previous chapters, a *Package* represents a general purpose grouping mechanism. Its functions in UML modeling are pretty similar to the one you use during system coding – it is commonly used for specifying the logical distribution of classes and other elements in the system. So, you use *Package* when you want to group any UML element (e.g. use case, actors, classes, components and other packages) together.

In UML, packages can be represented as simple as that:

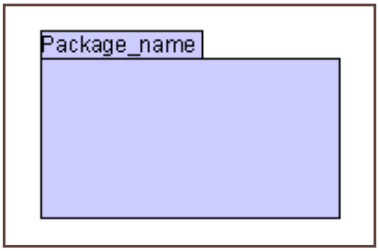


Figure 35. UML Package Representation

Developers usually accentuate the logical structure of the system by providing various relationships (e.g. dependency, association) among packages. This can be considered as a high level view of the system (Figure 36):

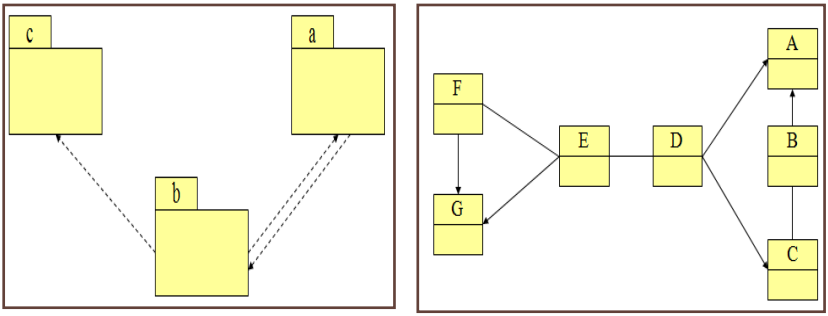


Figure 36. Specifying relationships among packages

It is obvious that you can create relations and dependencies between public classes from different packages, in a similar manner (Figure 37). This is useful since helps to emphasize the interface between packages.

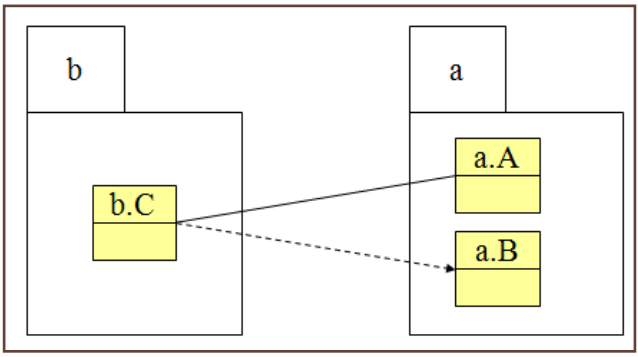


Figure 37. Specifying relationships among classes from different packages

6.11 Tips

The useful tips provided below will help you not to get confused with various techniques in UML modeling and use it in a consistent manner:

- Don't ever try to use all the various relationships and notations.
- Don't draw models for every single detail, it is better to concentrate on the crucial aspects of the system, which are important for its correct functioning.
- Try to draw implementation models only in case you want to illustrate a particular implementation technique.

Finally, let's review the steps you need to follow in UML modeling:

- Create class symbol in the editor you use (e.g. *TopCoder UML Tool*) and name it
- Specify the class attributes
- Specify the class operations
- Indicate the associations between classes
- Provide comments, notations and constraints, if needed.

6.12 Summary on Class Relationships

You studied various types of class relationships and their use cases. However, you need to remember that this strongly depends on the context, so for the same concepts there might be different relations under different contexts. Hence all examples should be domain-specific, since from another angle of view the association may become more specific.

For a final fixation of such an important topic, as class relationships, let's look at several relationships examples :

- Association – *"uses a"*
  Means that two classes have some kind of relationship, could be any relationship.
  Example: a class *Human* uses a class *Pen*
- Aggregation: *"has a"*
  Example: a class *Human* has a class *Car* . Note that in case *Human* dies *Car* will continue to exist.
- Composition: *"contains a, consists of B"*
  Example: a class *Human* owns a class *Heart*. *Heart* can't exist separately without a *Human*.
- Generalization: "*is a*"
  Example :a class *Student* is a class *Human*.

Very often beginners are confused about *Composition* and *Aggregation*. The simplest way to distinguish them is to think about how strong the relationship is. Particularly, think about what will happen in case you delete the owner object. In case of *Aggregation*, the part object continues to live. For instance , after cancelling the *Order* object, the *Product* object continues to exist. Concerning the *Composition*, the part object dies with its owner. For example, the *Paragraph* object dies together with the corresponding *Document* object.

It must be admitted that composition, aggregation, and association are semantic, not programming concepts. In Java, you may implement all of them in the same way. It's a conceptual difference.

As a rule of thumb, try to keep class diagrams as simple as possible. That will make it much easier to understand the diagrams. In addition, remember to label your classes and relationships by assigning descriptive names.

Although the class relationships seem to be very simple, be sure, it is just at first glance. When you work on a complex project, like in Figure X, it is very easy to get lost in such an abundance of classes.

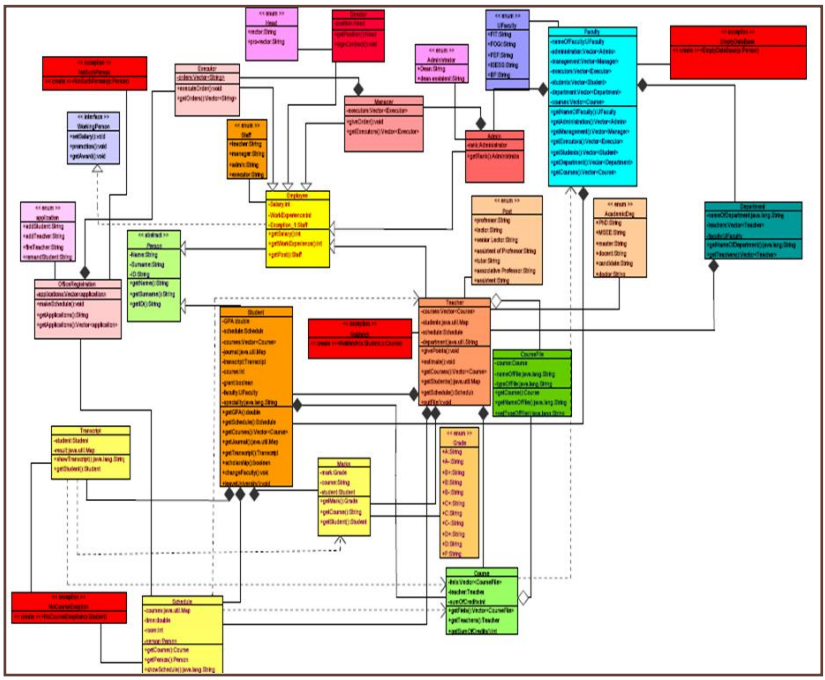


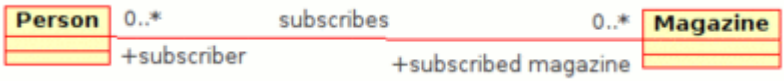
Figure 38. Complex UML diagram

KEY CONCEPTS YOU NEED TO GRASP FROM THE SECTION

Use Case Diagram *Sequence Diagram* *Actor* *Role*
Activation *Message* *Class Diagram* *Constraints*
Generalization *Association* *Realization* *Dependency*
Aggregation *Composition* *Multiplicity* *Visibility*

TESTS

1. What is the connection in the diagram … ?



a) Association
b) Dependency
c) Aggregation
d) Composition

2. What is the differences between composition and aggregation??

a) The whole of a composition must have a multiplicity of 0..1 or 1

b) A part must belong to only one whole

c) The part may have any multiplicity.

d) *a* and *b* are correct

3. Which one of the relationships below corresponds to relations *Person-Student, Person-Professor*?

a) Association
b) Generalization
c) Realization
d) Composition

4. Strong form of aggregation?

a) Association
b) Directed Association
c) Generalization
d) Composition

5. Relationship that indicates that one class implements a behavior specified by interface:

a) Realization
b) Directed Association
c) Generalization
d) Composition

6. The *Aggregation* relationship most precisely describes the relation between...

a) You and your hands

b) You and your friends

c) Your room and room of your neighbors

7. A special form of association that models a whole-part relationship between an aggregate and its parts?

a) Aggregation b) Association

c) Generalization d) Composition

8. Choose the relationships in UML class diagram that indicates inheritance?

a) Generalization b) Association

c) Realization d) Composition

9. What is the relationship between Department and Employee according to the code below?

```
public class Department {
       private int id;
       private Set<Employee> employees;
}

public class Employee {
       private int id;
       private String name;
}
```

a) Association b) Generalization

c) Realization d) Aggregation

10. What is the difference between Use Case and Sequence Diagrams?

a) Use Case Diagram represents a functional model, whereas Sequence Diagram – dynamic model

b) Use Case Diagram is more detailed

c) Sequence Diagram is more understandable by business users

d) Use Case Diagram represents an object model, whereas Sequence Diagram – dynamic model

PROBLEMS

1. In order to represent a triangle we need to have coordinates for each of its three sides (so, you need a class *Point*) and methods for creation of the triangle, area and perimeter calculation. Draw a UML *Class Diagram* to represent this triangle and any other classes involved.

2. You need to write a program that prints out an invoice. An invoice is used to describe the charges for a set of products in certain quantities. For the sake of simplicity we omit some complexities such as dates, taxes, customer numbers, etc. The program simply needs to print out the billing address, all items, and the amount due. Each item has the description (name of the item) and unit price of a product, the quantity that was ordered and the total price. Example is presented below:

INVOICE

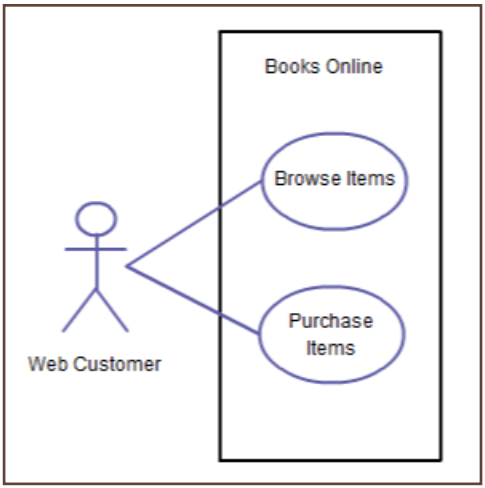
Tom's Small Appliances
543 Main Streat
New York, CS 32423

Description	Price, $	Q-ty	Total, $
Toaster	29.95	3	89.95
Hair dryer	24.95	1	24.95
Microwave oven	19.99	2	39.98

AMOUNT DUE : $ 154.78

Create a UML Class Diagram diagram and then implement and test the system.

3. Analyze the simple *Use Case Diagram* presented below and create analogous *Sequence* and *Class Diagrams*



4. Analyze the Use Case Diagram presented in Figure 39 and implement the corresponding system.

5. Analyze the Use Case Diagram presented in Figure 40 and implement the corresponding system.

6. Design the system for an enterprise healthcare company. Particularly, you need to design an internal application that doctors and other employees will use in order to manage information about the clients, their current plans, schedules, medication, assigned doctors, etc. Take into account the following details:

1. Doctor may retire
2. In case some doctor retires, system needs to assist user in finding a new doctor
3. Doctors need to be contacted annually in order to renew their contracts. How can you store a group of clients or doctors in your application?
4. Is there any way that you can write a single method that will handle addresses for both clients and doctors? How will you implement this?

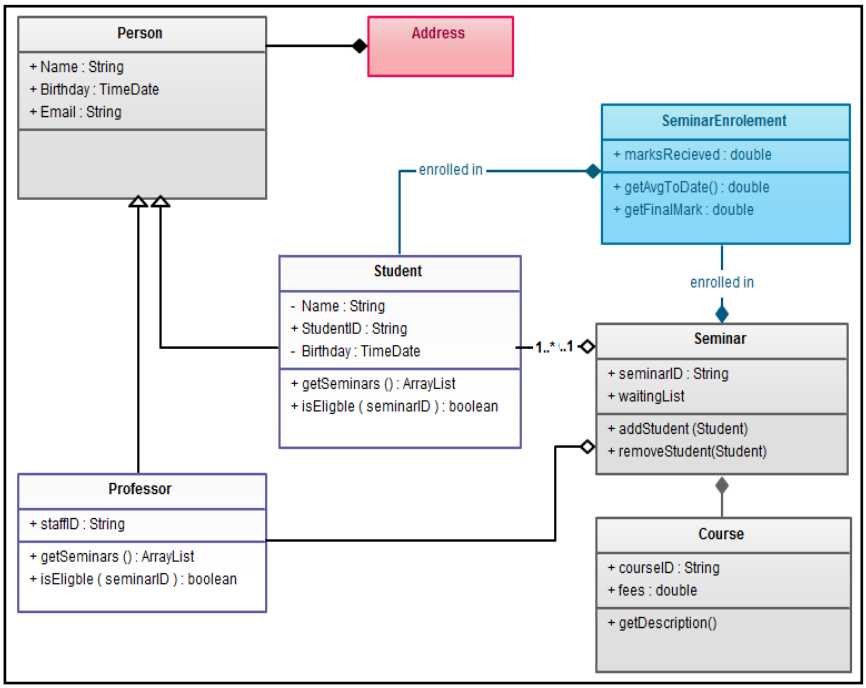


Figure 39. Class Diagram for Problem 3[13]

7. As you know, there exist two types of languages: natural language and programming language. The programming languages can be divided into object-oriented, imperative, logic and functional. Natural languages can also be divided into Turkic, European, Farsi, etc. Think how can you represent the language hierarchy and draw the UML *Class Diagram* for this hierarchy.

8. Create the UML *Class Diagram* that can be used to implement an electronic address book. Think carefully, which classes such application requires and what kind of functionality it needs to provide. Then implement this application.

[13] Diagrams from figures X, XX are taken from www.creately.com

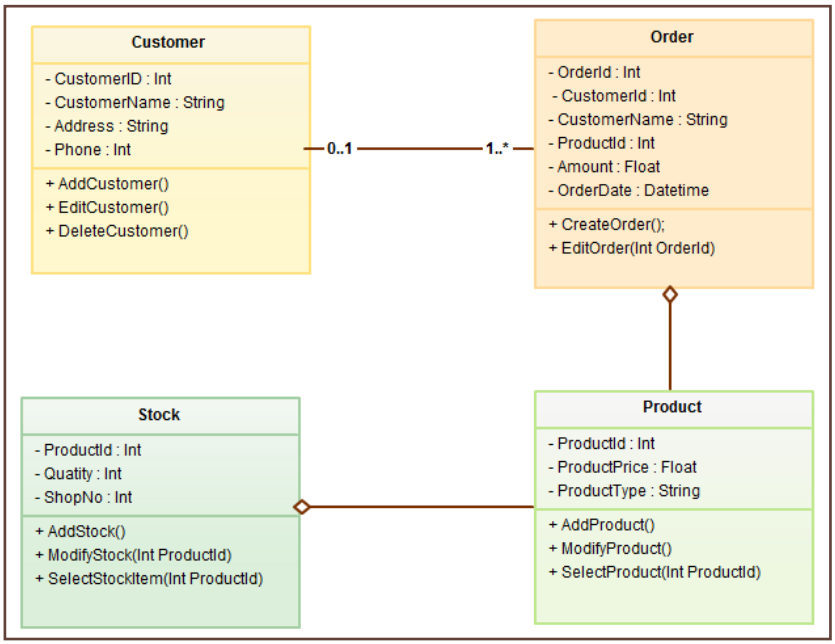


Figure 40. Complex UML diagram

9. A queue is an abstract data type for adding and removing elements. The first element added to a queue is the first element that is removed (first-in-first-out, FIFO).
Design an interface *AbstractQueue*, with methods to add and remove elements (integers). You also need to have a method to check whether the queue is empty. Then, create the class Queue implementing *AbstractQueue*. Before writing the code, create the corresponding UML *Class Diagram* . Then test your implementation with small and large queues
Note: Implement the queue with an array or Vector. In case you chose array: if the array becomes too small to hold all added elements, create a new larger (double the size of instance) array and copy all elements of the small array to the new one.

10. Write an abstract class *MathExpression* that represents some mathematical expression of undefined type. Define abstract method *findDerivative*() to return derivative of the expression.

Then create a class *Polynomial* that extends *MathExpression* and represents a polynomial with real coefficients. The coefficients of the polynomial should be passed as an array parameter with array type double in the constructor of your class. In addition, provide *equals*() and *compareTo*() methods that compare the polynomials by degree.

Hint: the resulting Polynomial is defined by **`coefficients[0] + coefficients[1] * x + coefficients[2] * x2 + ... + coefficients[n] * xn`**

7 Collections and Data Structures

Java Collection Framework is a unified architecture used to represent and manipulate various collections. This chapter introduces the fundamentals of Java Collection Framework, by providing an overview of the interfaces and concrete classes in this framework. Besides simple arrays, computer scientists discovered a variety of data structures having different performance tradeoffs.

In general, a collection is a data structure (object itself) which is used to hold other objects, to store, manipulate and organize objects in useful ways for efficient access (8). As a rule, elements in the collection represent data items forming natural group, like a collection of cards, telephone directory, mail folder, etc. If you check out the java.util package, you will find lots of interfaces and classes providing a general collection framework. They allow you to group multiple elements into a single unit in an efficient way.

Below you can see the classes and interfaces constituting the framework. As it can be seen, there is one root interface, *Collection*. It has two child interfaces. *Set* (unordered collection of unique elements) and *List* (ordered collection that might have duplicates). Map is another root interfaces for all collections storing key-value pairs. We will consider them in details later on in this chapter.

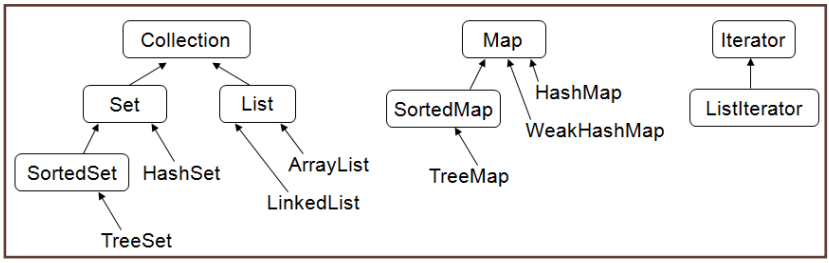


Figure 41. *Java Collections Hierarchy*

7.1 Root Interface - Collection

Actually, *Collection* is the least common denominator that all collection implement. Hence, every collection object (except maps) is a type of *Collection* interface. This is extremely useful, since we can use *Collection*

interface as a type and pass collection objects as parameters and manipulate them when maximum generality is required (remember *Polymorphism*?).

Java does not provide any built-in direct implementations of *Collection* interface, but it provides a number of implementations for more specific interfaces, extended from *Collection*, such as *Set* and *List*. Table 11 below presents main abstract methods in the *Collection* interface:

Signature	Description
`public int size()`	Returns the size of the collection
`public boolean isEmpty()`	Returns a boolean indicating whether element is in the collection or not
`public boolean contains(Object elem)`	Returns a boolean indicating whether element is in the collection or not
`public boolean add(Object elem)`	Depends on whether the collection allows duplicates
`public boolean remove(Object elem)`	Removes the specified element
`public Iterator iterator()`	Returns the Iterator for this collection
`public void clear()`	Remove all elements from collection
`public Object[] toArray()`	Returns a new array containing references to all collection elements

Table 11. *Methods of the Collection Interface*

Apart from methods in Table 11, *Collection* Interface also contains methods operating together with another collection object (Table 12):

Signature	Description
`public boolean containsAll(Collection coll)`	Returns the size of the collection
`public boolean addAll(Collection coll)`	Returns true if any addition succeeds. *Which logical operation is it?*
`public boolean removeAll(Collection coll)`	Returns true if any removal succeeds. *Which logical operation is it?*
`public boolean retainAll(Collection coll)`	Removes from the collection all elements that are not elements of coll. Which logical operation is it?

Table 12. *Methods of the Collection Interface*

7.2 *Iterator* and *ListIterator*

As you already know, the *Collection* interface defines an *iterator()* method to return an object implementing the *Iterator* interface. Iterators are able to access the elements in a collection without exposing its internal structure (9). However, it doesn't provide any guarantee regarding the order in which the elements are returned.

There are three defined methods in *Iterator* interface:

- public boolean **hasNext()** – returns true if the iteration has more elements
- public Object **next()** – returns the next element in the iteration
  In case the is no next element, an exception will be throws. Pay special attention to the fact that *next()* returns an Object, hence you may need special type casting.
- public void **remove()** – remove from the collection the element last returned by the iteration
  - can be called only once per call of next, otherwise an exception is thrown.

Code listing below demonstrates the classical usage example of the iterator. Note that this methods accepts any object, implementing *Collection* Interface.

```
public void removeBigStrings(Collection col,int max){
            Iterator it = col.iterator();
            while ( it.hasNext() ) {
                  String str = (String)it.next();
                  if (str.length() > max)
                        it.remove()
            }
      }
```

ListIterator is a subinterface of the *Iterator* interface. It extends its parent by adding methods to manipulate an ordered List object (e.g. LinkedList or Vector) during iteration. Specifically, you can remove the last returned element (***remove***() method), insert the object into the list after the last returned element (***add***() method), set the last returned element with another object (***set()*** method).

In addition, besides having inherited from *Iterator hasNext(), next()* methods *ListIterator* also has *hasPrevious(), previous(), nextIndex(), previousIndex()* methods. It is important to know that when iterator is at the

end of the list, *nextIndex()* will return list.size(), while when it's at the beginning of the list, *previousIndex*() will return -1.

There exist one problem with *Iterator/ListIterator* - they do not guarantee the *snapshot*[14] of the collection. It means that in case the content of the collection is modified while the iterator is in use, it will affect the values returned by the methods. Code listing below demonstrates such situation:

```
ArrayList a = new ArrayList();
        a.add("1");
        a.add("2");
        a.add("3");
Iterator it = a.iterator();
        while(it.hasNext()) {
                String s = (String)(it.next());
                if(s.equals("1")) {
                        a.set(2,"changed");
                        }
                System.out.print(s+ " ");
                }
```

The output will be : "1 2 Changes", although in case of snapshot we should have "1 2 3 ". Therefore, if you really need a collection snapshot, you need to make a copy of the collection.

It must be admitted that after the return of the Iterator object can't modify the collection before you completely traverse the collection. If you do this, *ConcurrentModificationException* will be thrown, as in example below:

```
ArrayList a = new ArrayList();
a.add("1");
a.add("2");
a.add("3");
Iterator it = a.iterator();
a.add("4"); // can't modify now!
while(it.hasNext()) {
        String s = (String)(it.next());
        System.out.println(s);
        }
```

Exception in thread "main" java.util.ConcurrentModificationException

[14] A shapshot returns the collection elements in the state they were when the Iterator object was created.

As regards the traversing of the collection, there exist two general schemes for that:

- Iterator (already shown)
- for-each
  The for-each construct allows you to concisely traverse a collection or array using a for loop. The general syntax for that is the following:

```
for (Object o: collection)
                System.out.println(o);
```

For instance, if you have a *HashSet* of *Student* objects *(HashSet<Student>)*, you can traverse it in the following way:

```
for (Student s: myHashSet)
                System.out.println(s);
```

Use iterators instead of *foreach* in case you need to Remove the current element.

7.3 *List* Interface

We already mentioned *Vector* in previous chapters, so the basics of *List* interface must be familiar for you . A *List* is an ordered *Collection* which allows duplicate elements to be stored together. The list's indices range from *0* to *list.size()-1*. List allows the precise control over where each element is inserted and enables to access elements by their position.

Due to its specific features, *List* has several new methods for an ordered collection, that enable:

- **Positional access**
  get(int index), set(int index, E element), add(int index, E element).
- **Search** (returns the index)
  indexOf(Object o), lastIndexOf(Object o).

The List interface is implemented by the following classes:

- **ArrayList** and **Vector**

Most commonly used implementation in the practice. In simple words, it is a resizable array implementation of the List interface. Prefer *Vector* when you need fast constant-time positional access. However, if you need to frequently add or remove elements from the middle, prefer *LinkedList*, since *Vector* requires *O(n-i)* for this. Another convenience of *Vector* (or *ArrayList*)

is that it can be efficiently scanned without the need of creating an *Iterator* object.
By the way, *ArrayList* and *Vector* are identical to each other with the only difference - methods of *ArrayList* are not synchronized, in contrast to *Vector*.
Going deeper in a hierarchy, there is a class *Stack*, extending *Vector*. It has the following typical methods:

- *peek()* - returns the object at the top of this stack without removing it.
- *pop()* - removes the object at the top of this stack and returns it
- *push(E item)* - pushes an item onto the top of this stack.

- **LinkedList**
  Prefer *LinkedList* when you know you will frequently add elements to the beginning of the List or iterate over the List to delete elements from its interior. But getting an element at specific index is more expensive for *LinkedList* - *O(i).* By the way, *LinkedList* in java represents a doubly-linked list. *LinkedList* also implements the *Queue* interface, providing first in, first out (*FIFO*) queue operations for *add*, *poll*, *peek*, remove[15] and so on. Finally, keep in mind that *LinkedList* have to allocate a node object for each its element, so you pay a big price in performance. So, use it when needed. Remember that positional access requires linear-time in a LinkedList and constant-time in a Vector or ArrayList.

7.4 *Set* and *SortedSet* Interfaces

The Set interface provides the same methods as Collection interfaces, but it provides a more specific contract for its methods (9). Particularly, you can't add duplicate element to a set, since it contains *UNIQUE* elements.

The *Set* interface has a child interface, *SortedSet*, which extends *Set* and provides an additional contract – iterators returned from that set return the set elements in a specified order. By default it will be the elements' order specified by the implementation of Comparable interface (for predefined types, like *String* and *Date*, the elements are stored in a typical alphabetical or chronological order, since *String* and *Date* both implement the Comparable

[15] The difference between *poll* and *remove* is that in case the stack is empty, *poll* will throw an exception, while *remove* will return null.

interface, which enables objects of that class to be sorted automatically). Apart from that, you can specify a Comparator object with the purpose of ordering the elements instead of the natural order specified by *compareTo()* method.

There are two implementations of *Set* interface which are widely used in everyday programming:

- *HashSet* – implements *Set*, implemented using a hashtable
- *TreeSet* – implements *SortedSet* , uses a balanced tree structure
- *LinkedHashSet* - Implemented as a hash table with a linked list running through it.

Generally, *HashSet* works faster than *TreeSet* , providing constant-time vs log-time for most operations. Nevertheless, HasSet offers no ordering guarantees, while *TreeSet* keeps elements sorted. So, the handicap of *HashSet* is chaotic ordering, of *TreeSet* – worse performance.

LinkedHashSet is considered to be intermediate between *HashSet* and *TreeSet*, since it offers insertion-ordered iteration and is almost as fast as *HashSet*.

Below is the simple example on *HashSet*. We have the bag of fruits (array), where fruits can repeat. What if we want to know just types of fruits in the bag? To get that, we just need to iterate over the array and attempt to add each fruit to a set.

```
String [] bagOfFruits = {"apple", "orange",
                         "plum", "plum", "pear", "apple"};
HashSet<String> fruits = new HashSet<String>();
for(String fruit: bagOfFruits)
       System.out.println(fruit+ " " +
              (fruits.add(fruit)? "added! " : "not added"));
System.out.println("Fruit types in the bag: "+ fruits);
```

Add () method returns a boolean value indicating the success of addition. So, the output of the program is going to be:

```
apple added!
orange added!
plum added!
plum not added
pear added!
apple not added
Fruit types in the bag: [orange, apple, pear, plum]
```

Note that in case we used TreeSet instead of HashSet for this example, there wouldn't be any difference, except the fact that fruit would be printed in alphabetic order.

In this example we used values of predefined type String as keys, that is why HashSet declined adding duplicates (String class has *hashcode()* and *equals()* methods). But what about the user-defined types? Lets' wrap the fruit to a class Fruit:

```
class Fruit {
        String name;
        public Fruit(String name) {
                this.name = name;
        }
        public String toString(){
                return name+"";
        }
}
```

Now let's see what happens if we put duplicate fruits to a set:

```
HashSet<Fruit> fruits = new HashSet<Fruit>();
fruits.add(new Fruit("apple"));
fruits.add(new Fruit("banana"));
fruits.add(new Fruit("apple"));
System.out.println(fruits);
```

The Output is: **`[apple, apple, banana].`**

Why this happens, why do we have duplicate fruits in the set? The answer is simple – our set stores objects of type *Fruit*, that we defined on our own. Since we defined what is a fruit, we also need to define when two fruits are equal. Remember equals() method of object class? We need to use it in order to specify the conditions for equality. Moreover, we need to define hashcode() method. Hashcode is an integer number that is formed for each object by a certain rule. It is used in searching operation in hashtables and can even shorten searching time. It is important to understand that hashcode is not always unique for different keys. Keys are equal, if their hashcodes are equal and method equals() returns true.

So, let's specify these methods (code is written inside of the *Fruit* class):

```
public boolean equals(Object o){
                Fruit other = (Fruit)o;
                if(other.name.equals(this.name))
                        return true;
                return false;
        }
public int hashCode(){
                return name.hashCode();
        }
```

After addition of this code we have just two fruits in the set, apple and banana.

7.5 *Map* and *SortedMap* Interfaces

The *Map* interface does not extend *Collection* interface since a *Map* contains *key-value pairs*, not just keys. As you know, maps can't contain duplicate keys and each key must map to just one value.

Interface *SortedMap* extends *Map* and keeps its keys in a sorted order. Sorted maps are widely used for ordered collections of key-value pairs, like dictionaries, address books, and telephone directories.

There are three *Map* implementations: *HashMap*, *TreeMap* and *LinkedHashMap*. If you need to keep pairs in ascending key order, then use *TreeMap*, which is the implementation of *SortedMap* interface.. In case you want to get maximum performance and don't care about iteration order, then make use of *HashMap*. Finally, if you want something intermediate, the compromising alternative is *LinkedHashMap*, which provides good performance and insertion-order iteration. But still, *HashMap* implementation of the map is considered to be the most efficient, providing constant-time performance for the basic *get* and *put* operations. This state of affairs is similar to the one we have with 3 *Set* implementations.

Let's consider a simple example on using maps to store mappings of the programs to the extension of their files:

```
HashMap<String, String> extensions =
            new HashMap<String,String>();
extensions.put("Word", "docx");
extensions.put("Excel", "xlsx");
extensions.put("Power Point", "pptx");
System.out.println(extensions.containsKey("Word"));
System.out.println(extensions.get("Word"));
```

There are methods that return the set of keys or collection of values only, *keySet()* and *values()*, respectively. In order to fetch value mapped to a particular key, use *get* method:

```
for(String program: extensions.keySet())
    System.out.println(program+ " has "+
    extensions.get(program)+ " extension");
```

It must be admitted that the collections returned by these methods are referenced by the *Map*, hence in case of removing an element from these collections the respective key-value pair will also be removed from the map.

7.6 *Collections* Class

Collections is an utility class that provides exclusively static methods operating with collections. Table 13 presents the frequently used methods provided by *Collections* class. Note that the first argument is the collection on which the operation is to be performed - *c*, the second one (if present) the parameter of type Object – *obj*..

Method	Description
`Collections.sort(c)`	Sorts the list *c* in ascending order. If you try to sort a list, the elements of which do not implement *Comparable* interface, a *ClassCastException* will be thrown
`Collections.shuffle(c)`	Randomly permutes the elements in list *c*
`Collections.reverse(c)`	Reverses the order of elements in *c*
`Collections.max(c)`	Returns the maximum element in *c*. There is analogous *min* method.
`Collections.fill(c,obj)`	Replaces all of the elements in *c* with the specified element.

`Collections.binarySearch(c, obj)`	Searches the list *c* for the object *obj* using the binary search algorithm.
`Collections.frequency(c, obj);`	Counts the number of times the specified element occurs in collection *c*
`Collections.disjoint(c1, c2);`	Returns true if *c1* and *c2* contain no elements in common, and false otherwise

Table 13. *Main Static Methods in the Collections class*

Remember that all the methods above are static, so you need to call them using the class reference, as in table 13.

KEY CONCEPTS YOU NEED TO GRASP FROM THE SECTION

Collection Set List SortedSet HashSet TreeSet LinkedHashSet Map HashMap LinkedHashMap Vector Stack Queue LinkedList ArrayList Collections class Iterator ListIterator Comparable equals hashcode

TESTS

1. What can be used to create stacks, queues, trees and deques (double-ended queues)… ?

a) ArrayList b) LinkedList

c) Collection d) HashSet

2. *Collection* is the root interface in the collection framework from which interfaces _____ and _____ are derived.

a) Set, Map b) List, Iterator

c) Map, Set, List d) Set, List

3. Which collection lets you associate its elements with key-value pairs, and objects according to FIFO (first-in, first-out) policy?

a) Vector b) HashMap

c) LinkedHashMap d) TreeMap

4. If you want to store elements in a collection that guarantees that there are no duplicates and all elements can be accessed in natural order, what collection you will choose?

a) Any Set b) TreeSet

c) HashSet d) TreeMap

5. An ordered collection that allows presence of duplicate elements?

a) Any Set b) ListIterator

c) Any List d) Vector

6. Choose the interface that doesn't extend *Collection* interface?

a) Map b) Set

c) SortedSet d) List

7. Which of the following will sort the elements in a list *items*?

a) items.sort() b) new LinkedList(items, sort)

c) Collections.sort(items) d) Arrays.sort(items)

8. Which statement is true for the class HashSet?

a) The elements in the collection are ordered

b) The elements in the collection are guaranteed to be unique

c) The elements in the collection are accessed using a unique key

9. Which of the methods are not defined in *Iterator* interface?

a) public void size() b) public void remove()

c) public Object next() d) public boolean hasNext()

10. Choose Collection which remembers the order in which you add elements?

a) LinkedList b) HashSet

c) LinkedHashMap d) TreeMap

11. Choose method that adds element to any Collection?

a) add() b) put()

c) insert() d) set()

12. Choose method that can be used to add key-value pair to a map?

a) add() b) put()

c) insert() d) set()

13. Which of these methods determines union of Sets?

a) retainAll() b) retain()

c) addAll() d) containsAll()

14. What is the parent interface for TreeMap?

a) SortedMap b) Map

c) HashMap d) Comparable

15. What is incorrect?

a) Set, List, Map are interfaces of Collection interface

b) Map contains elements with unique keys

c) The interface List is implemented by Vector

PROBLEMS

1. Check out the studied classes and interfaces at java API, and make examples using some common methods.

2. Prove that the utility methods in Collections class behave correspondingly (make examples).

3. Create enumeration *Extension*. Then create a class *Document* having an instance of *Extension* enumeration, title(String), size (long). Override *equals()* method. Then create a class *Folder* having a *HashSet* of documents and a name (String). So that we can write:

```
Document d = new Document("lab7", 200, Extension.DOC);
Folder f = new Folder("labs");
f.add(d); // adds to HashSet
```

Add several documents to Folder and print them using *foreach* operator.

4. Prove that the utility methods in Collections class behave correspondingly (make examples).

5. Create a class *Mark* having a field *points* (e.g. 95) and a method *getLetter()* (e.g. A-). Implement *Comparable* and *Cloneable* interfaces and provide *toString()* and *equals()* methods . Then create a class *Student* with fields *name*, *id*, *mark* (of type *Mark*). Then create several objects of *Student*, store them in a Vector and sort by mark they have.

6. Write a class *Student* with the field *grade* of type *int*. Then create several instances of your class and add them to *TreeSet*. Then iterate through your set (using foreach or iterator) and show what will be the output.

7. Create abstract class *MyCollection* with all the methods below. Then, implement *MyVector* (implements *MyCollection* interface) class to represent a growable array of objects. Like an array, it contains components that can be accessed using an integer index. Your class will store integers.

Constructors:

MyVector() – constructs an empty vector.
MyVector(int [] a) – constructs a vector with all integers from array a.

Methods:

add(int element) - appends the specified element to the end of this Vector.

add(int index, int element) - inserts the specified element at the specified position in this Vector. Do not forget to throw Exception if index > size.

clear () - removes all of the elements from this Vector.

contains (int o) - returns true if this vector contains the specified element.

get(int index) - returns the element at the specified position in this Vector.

indexOf(int o) - returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.

insertElementAt(int element, int index) - inserts the specified integer vector at the specified index.

isEmpty() - tests if this vector has no components.

removeAt(int index) - removes the element at the specified position in this Vector.

remove(int element) - removes the first occurrence of the element in this Vector.

removeAll(int element) - removes all occurrences of element in this Vector.

reverse() - reverses the elements in the array

set(int index, int element) - replaces the element at the specified position in this Vector with the specified element.

size() - returns the number of components in this vector.

sort() - performs sorting of the elements in the array.

toArray() - returns an array containing all of the elements in this Vector
in the correct order.

toString() - returns a string representation of this Vector

8. Write a console application representing a simple telephone book, containing mappings from names to telephone numbers. Think carefully, which data structure can be used for this type of application. In the main menu user has options to add new contact, to view all contacts, to update information on particular contact, and search the contacts by name.
Hint: make use of *Scanner*, while(true), labels.

8 Files and Streams

In general, a *stream* is an abstraction of the continuous one-way flow of some data. It might be useful to think of a stream as of the somehow ordered sequence of data that have a ***source*** (input streams) or a ***destination*** (output streams), see Figure 42 below.

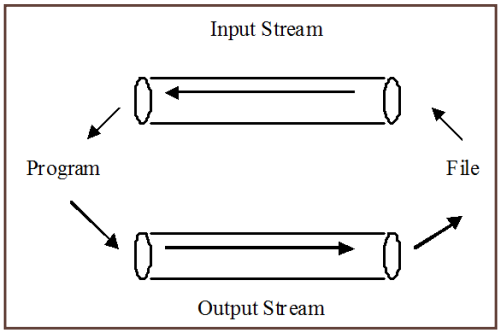


Figure 42. *Streams Illustration*

8.1 Hierarchy of Streams in Java

Stream classes are generally categorized into two types: *byte streams* and *character streams*. Character streams represent a higher level concept as compared to Byte Streams. Actually, a Character Stream is a Byte Stream which is wrapped with some kind of logic enabling it to display specific encoding characters , rather than reading bytes and then decoding the chars they represent (5).

All byte stream classes in Java have either InputStream or OutputStream parent root class, and all character stream classes in Java have either Reader or Writer root class. It is important to note that the subclasses of these two groups of root classes are analogous to each other.

Figure 43 below represents the hierarchy for Byte Streams in Java.

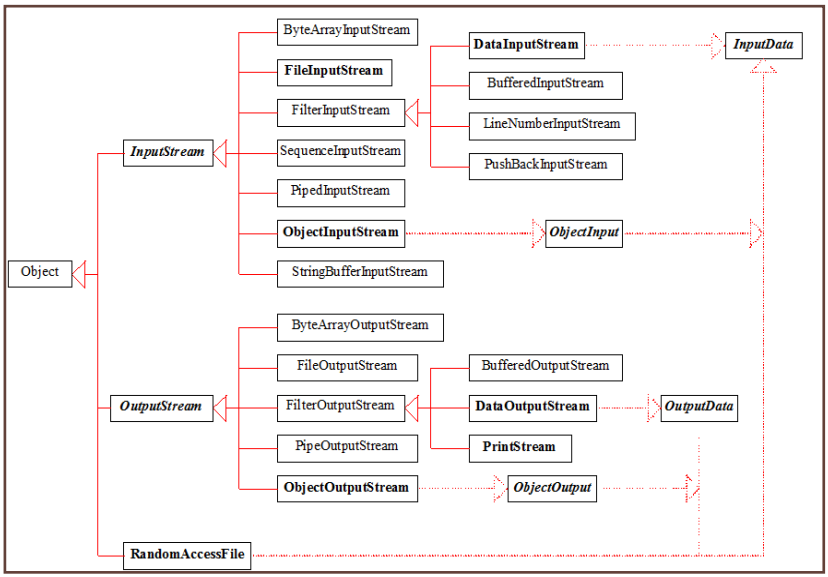


Figure 43. *Byte Streams in Java*

Figure 44 represents the hierarchy for Character Streams in Java.

In order to read form or write to a disk file you need to use file streams. For byte streams, make use of FileInputStream and FileOutputStream, and for character streams, use FileReader and FileWriter for reading and writing, respectively. To construct the objects of these classes, you need to provide the file name to a corresponding constructor, for example:

```
//byte input stream
FileInputStream infile = new FileInputStream("in.txt");
//character output stream
FileWriter outfile = new FileWriter("out.txt");
```

8.2 Data Streams

Data Streams are used to read and write Java primitive types, like int, long, Boolean, etc., in a machine-independent fashion. Hence, you can write a data file in one machine and read it on another machine that has a different operating system or file system, for example. There are two respective data stream classes in Java, for reading – *DataInputStream*, for writing – *DataOutputStream*.

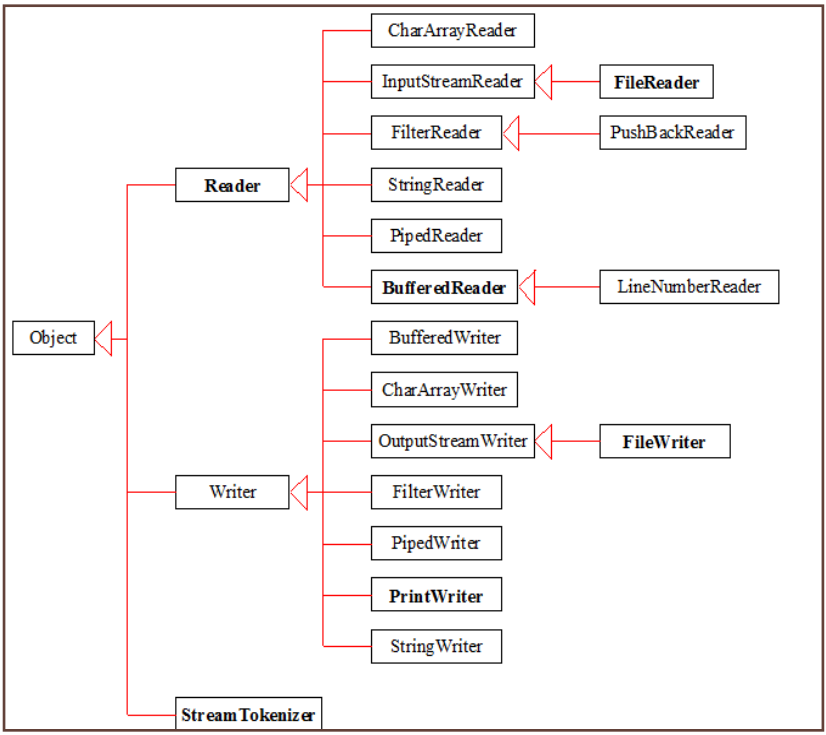


Figure 44. *Character Streams in Java*

To construct the instance of this classes, you need to pass to a corresponding constructor the FileInputStream (for DataInputStream) or FileOutputStream (for DataOutputStream) instance bound to a file you plan to read from or write to:

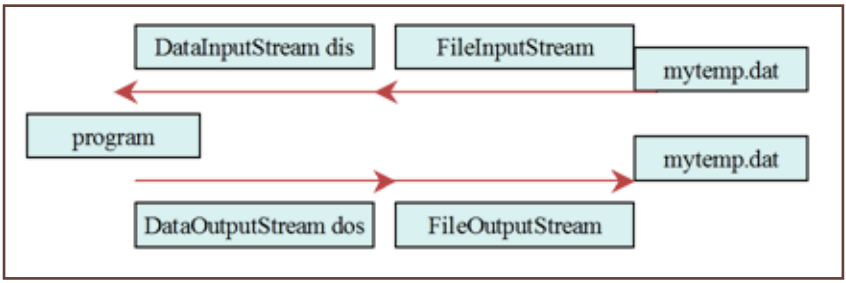


Figure 45. Wrapping File Steams inside of the Data Streams

```
DataOutputStream outfile = new
DataOutputStream(new FileOutputStream("out.txt"));
```

Once you have such instance, you can write operations (or read operations, if you create DataInputStream object). Actually, using data streams you can read / write any primitive type variable, the method name for that is the following: *[read/write]Type,* for example, for *DataInputStream* :

```
int readShort() throws IOException
int readInt() throws IOException
char readChar() throws IOException
```

And for *DataOutputStream*:

```
void writeByte(byte b) throws IOException
void writeInt(int i) throws IOException
void writeBoolean(boolean b) throws IOException
```

8.3 Print Streams

The limitation of the data output streams is that they output data in a binary format. Hence, if you open the output file, you won't be able to view its contents as text – it is not readable. If you need such option, you can use print streams (e. g. *PrintWriter* class provides this functionality) to output data into files. In this case, you will be able to see the file as a readable text.

To use PrintWriter, you need, as usually, create its instance. As a parameter you can pass either stream for writing to a console, or to a file:

```
PrintWriter(Writer out)
PrintWriter(OutputStream out)
```

For example:

```
//to read from console
PrintWriter pw = new
             PrintWriter(new OutputStreamWriter(System.out));
//to read from file
PrintWriter pwFile = new
             PrintWriter(new FileWriter("a.out"));
```

The main methods this class provides are *print* and *println* methods. Remember overloading? Due to overloading, you can pass variable of any

primitive type to these methods, including *int*, *double*, *long*, *boolean*, etc. These methods also might accept second optional parameter of type boolean, indicating *autoflush*. By default, it has a *false* value. *Autoflush* means to automatically flush the content of the buffer will be flushed upon receiving some data (9). If autoflush is false, then, data will be printed just upon closing the stream. So, the rule of thumb is to always close streams, since you may end up with the empty file.

8.4 Buffered Streams

Buffered Streams are extremely useful, since they considerably speed up input and output operations by reducing the number of reads and writes. Particularly, in case of input, *a bunch of data is read all together at once instead of just one byte at a time*. Similarly, in case of output, *data are first cached into a buffer*, then written all together to the specified file. It is highly recommended to use buffered streams. You can wrap streams to BufferedReader/BufferedWriter for reading/writing from/to console and file. For example:

```
BufferedReader in
        = new BufferedReader(new InputStreamReader(System.in));
BufferedReader in
        = new BufferedReader(new FileReader("fileName"));
```

Straight after that you can invoke *in.readLine()* to read from the file line by line. Now let's consider a simple example on using both BufferedReader to read information from console and PrintWriter to write write it to a file:

```
try {
      BufferedReader br = new
         BufferedReader(new InputStreamReader(System.in));
      PrintWriter pwFile = new
         PrintWriter(new FileWriter("a.out"));
      String line = br.readLine();
      while (!line.equals("q")) {
              pwFile.println(line);
              line = br.readLine();
        }
        br.close();
        pwFile.close();
      } catch(IOException ioe) {
              System.out.println("Can't read!");}
```

As it is seen, we read the information, line by line, until user enters "q", and print it to a file *a.out*. Note than in case you want to read form file, not from a console, all you need to change is to put *FileReader* object instead of *InputStreamReader* and provide the name of the file in its constructor.

Keep in mind that using buffered streams is very useful, since they provide the buffering of characters in order to enable the efficient reading of characters and lines.

8.5 Serialization

Generally, Serialization is the process of converting the object to a sequence of bits. Actually, by *object* here we mean any object, really. It can be the one which is predefined in Java (Date, HashMap, String, array of TreeSets, etc.) or the one you defined on your own. Deserialization is the reverse process of unpacking the sequence of zeros and ones into an original object. Hence, Serialization and Deserialization mechanisms allow you to transfer objects to and from a stream.

To allow an object to be read or write, the object's defining class has to implement the *Serializable* interface (java.io package). This interface is a marker interface, so, it has no methods, that is why you don't need to add some additional code in your class to enable it to be serializable. Implementing this interface let's Java serialization mechanism to automate the processes of storing and retrieving the objects from a file.

In order to perform input and output at the object level you need to use Object streams. In particular, you need to make use of these classes:

- *ObjectOutputStream* – for storing objects (writing them to file)
- *ObjectInputStream* – for restoring objects (reading them)

It must be admitted that Serialization is only concerned with writing objects and the fields they contain to a stream, hence this excludes static members of a class. (static type variables will have their default values).

Let's highlight the requirements for a class that needs to be serializable:

1) Class must be public
2) Class must implement Serializable interface
3) All base classes of the class are desired to implement Serializable interface too. However, if not, it is enough for such base class to have a default constructor (no-argument constructor).

So, let's suppose we want to create some objects of Book and then retrieve them. First of all, let's create the class itself:

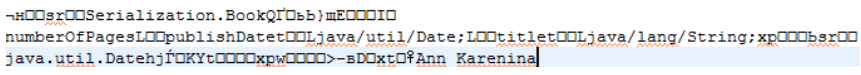
```
import java.io.Serializable;
import java.util.*;
public class Book implements Serializable{
	int numberOfPages=0;
	private String title= "No title";
	Date publishDate = new Date();
	public Book(int num, String title){
		numberOfPages = num;
		this.title = title;
	}
	public void setTitle(String title){
		this.title = title;
	}
	public String getTitle(){
		return title;
	}
	public String toString(){
		return title+" ,"+numberOfPages+
		" pages, published: "+publishDate;
	}
}
```

The code sample below demonstrates that it is lot less difficult to serialize the object than it sounds :

```
FileOutputStream fos = new FileOutputStream("book.out");
ObjectOutputStream oos = new ObjectOutputStream(fos);
Book b = new Book(220,"Ann Karenina");
oos.writeObject(b);
oos.flush();
oos.close();
```

Again, we can witness the wrapping here – we wrap fos object inside of *oos*. The serialization procedure is fully implemented by *ObjectOutputStream*, *FileOutputStream* just represents the file in this context.Note that you need to place this code to a try-catch block (for catching exceptions of type *IOException*). We will study exception later on in the following chapter. For now, just think, there can be various exceptions involved in Serialization – file is not found, class does not implement *Serializable* interface or it is not public, etc.

If you open book.out you will see something like this:



Why? Because in order to save space, the objects are stored in such a format. This file is not intended to be read by a programmer. It is intended to be read by ObjectInputStream, that performs the deserialzation, after which we can get the information:

Now let's see how to recover the serialized file from book.out.

```
FileInputStream fis = new FileInputStream("book.out");
ObjectInputStream oin = new ObjectInputStream(fis);
Book b = (Book) oin.readObject();
System.out.println(b);
```

Note that *readObject()* method returns the object, that is why we need to apply type casting to convert back to Book. The output of this program is the following:

```
Ann Karenina ,220 pages, published: Sun May 12 09:56:51
BDT 2013
```

As it was already mentioned, you can apply serialization to even more complex objects (e.g. HashMap that has objects as key and value). Just do not forget to correctly type-cast your object.

8.6 Sequential and Random Access Files

All of the streams that we have covered up to now (FileInputStream and FileOutputStream, FileReader and FileWriter, BefferedReader and BufferedWriter) are based on sequential access. In other words, they enable you to treat a file as a stream to input or output *sequentially*. In contrast, *RandomAccessFile* class lets you read and write data beginning at the a specified location, all you need to do for that is to set a ***file pointer*** to indicate the starting position , by using *seek()* method.

Moreover, **RandomAccessFile** allows a file to be read and written at the same time. It includes typical methods, that are already familiar to you, like readInt(), readLong(), writeDouble(), readLine(), writeInt(), and writeLong(). Other useful methods are presented in Table 14 below:

Method Signature	Method Description
`void seek(long pos)`	Sets the pointer to where the next read or write need to happen
`long getFilePointer()`	Returns the current pointer offset, in bytes, from the beginning of the file
`long length()`	Returns the length of the file.
`final void writeBytes(String s)`	Writes a string to the file

Table 14. *Methods of RandomAccessFile class*

In the example below we use *RandomAccessFile* to open the file for both reading and writing. Then, we set the pointer to the beginning of the file and run till the length of the file, reading and printing the current character. Upon finishing the reading, we write the corresponding message to the end of the file.

```
try{
        RandomAccessFile rand =
                new RandomAccessFile(file,"rw");
        int i= (int)rand.length();
        rand.seek(0);//Seek to start point of file
        for(int ct = 0; ct < i; ct++){
               byte b = rand.readByte();
               System.out.print((char)b)
               }
        rand.writeBytes("Finished scanning");
        rand.close();
}
catch(IOException e){
        System.out.println(e.getMessage());
}
```

8.7 The File Class

The **File** class is very useful utility class that is used to retrieve the information about a file or a directory from a disk.

It must be admitted, that an object of this class represents merely a path, but not the underlying file. Hence, it can't be used to open files, read or write it, or provide any other processing capabilities (5).

You can create an instance of this file by either specifying the full path in the constructor, or by providing two arguments, directory (of type *File*) and file name:

```
public File( String name)
public File( File directory, String name)
```

Using methods of this class, you can check whether file exists or not (exists() method), whether it is file or directory (isFile(), isDirectory() methods), get parent directory (getParent()), determine length of the file (length()), and many other methods. Check Java Api for more information about the methods of File class.

By the way, every file stream type has two types of constructors, the one that takes a single string, representing the file name, and the other one, which accepts an object of type File which refers to the file.

An example code is presented below. This program reads the name of the file from the console and then, in case file does not exist, exits the program.

```
BufferedReader in = new
        BufferedReader(new InputStreamReader(System.in));
System.out.print("Enter File name : ");
String str = in.readLine();
File file = new File(str);
if(!file.exists())
       {
       System.out.println("File does not exist.");
       System.exit(0);
       } else {. . .}
```

8.8 *StringTokenizer*

The *StringTokenizer* class allows you to break a string into tokens, by a certain delimiter or a set of delimiters (9). It is frequently used for parsing text files whicr are in specific formats. An object of *StringTokenizer* can behave in two ways, depending on the boolean value (third parameter in the constructor called returnDelims). If this flag is false, the delimiter indicated is used to separate the string into tokens. If it is true, delimiter is considered to be token itself. The small example below will help you too clarify this issues:

```
StringTokenizer st = new StringTokenizer("A *B* Cccc", " ");
while (st.hasMoreTokens()) {
       System.out.println(st.nextToken());
       }
```

First parameter in the StringTokenizer constructor indicates the string itself, the second one – the delimiter. The output of this code is the following:

```
A
*B*
Cccc
```

Now let's look at the next example in which we use delimiter as a token, by setting returnDelims to true. By the way, in case you do not specify the 3rd parameter it will be false by default, as in the previous example.

```
StringTokenizer st2 = new
              StringTokenizer("A*B*Cc|cc", "*|", true);
while (st2.hasMoreTokens()) {
       System.out.println(st2.nextToken());
       }
```

Notice the delimiter in this example, "*|". This complex delimiter indicates that string will be splitted by both * and | characters. The sample code above has the following output:

```
A
*
B
*
Cc
|
cc
```

KEY CONCEPTS YOU NEED TO GRASP FROM THE SECTION

Stream *Byte Streams* *Character Streams* *StringTokenizer*
BufferedReader *BufferedWriter* *Serialization* *File*
Deserialization *PrintWriter* *ObjectInputStream*
ObjectOutputStream *RandomAccessFile* *Sequential Access*

TESTS

1. How Stream classes can be categorized?

a) Byte Streams and Character Streams

b) Writer Streams and Reader Streams

c) Data Streams and Object Streams

2. BufferedReader is used to read :

a) text from a character-output stream

b) text from a character-input stream

c) an array of bytes

d) one byte from a file

3. Which of these classes allow a file to be read and updated at the same time?

a) Reader/Writer b) ObjectStream

c) RandomAccessFile d) StringTokenizer

4. How to deserialize a HashMap<String, Integer> ?

a) hm = (HashMap<String, Integer>) oin.read();

b) hm = (HashMap) oin.readObject();

c) hm = (HashMap<String, Integer>) oin.readObject();

d) hm = (HashMap<String, Integer>) oin.writeObject();

5. How to create BufferedWriter object to write to a file a.out?

a) BufferedWriter bw = new BufferedWriter(new FileWriter("a.out"));

b) BufferedWriter bw = new BufferedWriter("a.out");

c) BufferedWriter bw = new BufferedWriter(new FileOutputWriter("a.out"));

d) BufferedWriter bw = new BufferedWriter(new File("a.out"));

PROBLEMS

1. Create a class *Contact* with fields *name*, *phoneNumber*. Provide *equals()* method. Show how to serialize a Vector of *Contact* objects.

2. Create a class *Mark* having a field *points* (e.g. 95) and a method *getLetter*() (e.g. A-). Implement *Comparable* interface and provide *toString*() and *equals*() methods . Then create several objects of *Mark* , store them in a Vector and serialize.

3. This problem consists of 2 parts.

a) Write a program that reads student scores from the file "scores.txt", stores the score in some appropriate collection (guess which one), finds the best score, and then assigns grades for all the students according to these plan:

- Grade is A if score is >= best–10;
- Grade is B if score is >= best–20;
- Grade is C if score is >= best–30;
- Grade is D if score is >= best–40;
- Grade is F otherwise.

The result must be stored in a file "grades.txt" in exactly the following format format:

1) Ivanov Ivan – "A"

Format of scores.txt:

Ivanov Ivan 100
Aliyev Ali 22
Menshikov Sergey 65
…

b) Now write the second program. Suppose you don't need to store students' names, all you need is just the maximum, minimum and average mark. Think carefully, which collection needs to be used with new conditions.
Print the answer to the same "grades.txt" file. Take into account the fact that you need to ALTER (update) the file, not override it, so the previously printed information must not be arisen. So, the "grades.txt" may look like:
1) Ivanov Ivan – "A"
…
…
Average – 60
Maximum – 100
Minimum – 25

4. Develop an application for a university course offering. There are to be classes for *Textbook*, *Instructor*, and *Course*. This

application should also have a Driver class to test the classes constructed.

Classes:

1. ***Textbook***: this class is to have variables (fields) for isbn, title, and author(s).
And, it should include a constructor, accessor and mutator methods for the fields, as well as toString and equals methods.
2. ***Instructor*:** variables to include are firstName, lastName, department, and email. Define a constructor and methods as done for Textbook.
3. ***Course*:** this class is an aggregation of data which includes a variable for *courseTitle*, and variables for *Textbook* and *Instructor*. The *Course* constructor would initialize the *courseTitle* and reference the *Textbook* and instructor constructors to initialize fields for those classes. Include accessors and mutators and a *toString*(), *equals*() methods.

Driver or test program:

The Driver application should be run in 2 modes: *user mode* and *admin node*.
The user mode should offer the user choices to :
a) View the list of available courses
b) Display information about the course
For the second choice (b) , the program should be capable of retrieving values associated with course objects then printing those details.
Textbooks, courses and instructors need to be inputted to a system by admin (admin mode). Admin's username and hashed password need to be stored in a file "admin.text" . Each time admin enters the system, it should be logged to "admin.txt" (you need to alter the file, not to override). So, format of the "admin.txt" is the following:

```
Username: root
Password:  z53h /// it is a hash!

27.10.12 13:44 admin logged in to a system
27.10.12 13:47 admin added new course "Natural language processing"
27.10.12 14:05 admin added new textbook "Data Mining – tools and applications"
```

…

For console input, include user prompts. Include descriptive headings or labels to identify output data. Data for several textbook, instructor and course objects inputted by the admin need to be saved by using serialization. So, for displaying the information to users you need to deserialize it.

9 Exceptions

"The primary duty of an exception handler is to get the error out of the lap of the programmer and into the surprised face of the user. Provided you keep this cardinal rule in mind, you can't go far wrong."
Verity Stob

As you might already noticed, almost everything in Java represents object. And *Exceptions* are not an exception. Because exceptions are objects, just as other objects, different types of exceptions can be subclasses of one another. For instance, *FileNotFoundException* is a class derived from *IOException*. You need to clearly understand that if we catch *IOException*, then we also catch all its child classes, including *FileNotFoundException*.

9.1 Exceptions Hierarchy

Figure 46 depicts the simplified version of the exceptions hierarchy in Java. As it can be seen, all exceptions in Java extend the class *Throwable*, which directly splits into two branches, *Error* and *Exception* (9). Error and its subclasses represent internal errors and resource exhaustion inside of the Java runtime system (e.g. *OutOfMemoryError*). According to Java Documentation, Error might occur due to some abnormal conditions that should never occur (9). It is little you can do to handle it. Another child class of *Throwable*, in which we are more interested is and need to focus on is *Exception*. Subclasses of *Exception* correspond to errors that a program can recover from.

Exception class splits into two branches. Namely, *IOException*[16] and *RuntimeException*. All exceptions derived from *RuntimeException* mostly happen because errors exist in your code, hence it is your fault. For example, an incorrect cast (*ClassCastException*), out-of-bounds access to array (*IndexOutOfBoundsException*), access to null object (*NullPointerException*), etc. The second branch of classes includes exceptions which are, actually, happened not by your fault – your program

[16] This is the main subclass of Exception not derived from *RuntimeException*. There are also other ones, besides *RuntimeException* and *IOException, which are direct childs of Exception, like SQLException, MalformedURLException, among others.*

is good, but some other bad things happened. For instance, when you try to open a malformed URL (*MalformedURLException*).

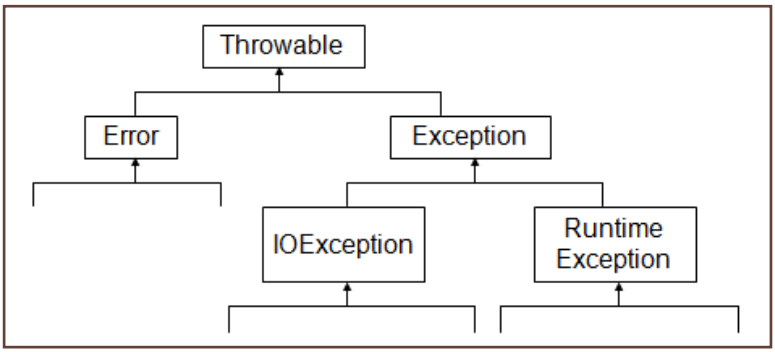


Figure 46. *Exceptions Hierarchy in Java*

9.2 Checked and Unchecked Exceptions

Exceptions which are derived from the class *Error* or from the *RuntimeExcepiton* are considered to be ***unchecked***. As a rule, they represent programming errors, not errors happened due to invalid user input data, for example. So, unchecked exceptions can be thrown "at any time", and they don't' have to be explicitly handled, methods don't have to declare that they can throw such exception (3). Since any method can *throw* them – actually, certain instructions (e.g. array access, integer division, etc.) can throw it. Examples of such exceptions are *NullPointerException*, *ClassCastException*, *OutOfMemoryError*.

When an unchecked exception happens, Java handles it automatically. The example below demonstrate this:

```
String name = null;
char c = name.charAt(8);
```

After launching of this program, at runtime, you will get the following error:

```
Exception in thread "main"
java.lang.NullPointerException
at Tutorial.teststts.main(teststts.java:51)
```

All the remaining exceptions represent ***checked*** exceptions**.** Basically, they are called "checked" since the compiler checks whether programmers provide exception handlers for the exceptions which may happen. This is done in order to ensure that in case the exception occurs, it is handled in some way. Hence, the flow of these exceptions is explicitly controlled.

9.3 Try – Catch Block

The standard syntax of try/catch block is the following:

```
try {
        //statements
} catch(exceptionType1 identifier1) {
        //handler for type1
} catch(exceptionType2 identifier1) {
        //handler for type2
} . . .
```

You can provide catch blocks for more than one exception, but keep in mind that the order *must* be from the most specific to the most general, since we want to handle the exception in a most specific way possible. There is a plenty of examples from real life based on the same idea. For example, you fill headache, and in case you have general reliever, or analgesic (e.g. "Noshpa"), and some medicine which specifically helps to tackle headache (e.g. "Tempalgin"), you will certainly choose "Tempalgin".

As a rule, a potentially dangerous code that might generate exceptions is placed into a try *block*. In turn, the handling of this exception (actions, that need to be done in case exception happens) are put into a *catch* block. There are two possible situations using execution of the statements in a try block: exception either happens or no

In case an exception is thrown, the program flow is interrupted and the remaining code in the try block is skipped. After that the provided catch clauses are checked for the compatibility with the type of the thrown Exception. In case an appropriate *catch* block is identified, the code inside of its body is executed. Needless to say, all the remaining *catch* blocks are ignored after that. If there were no appropriate *catch* clauses found, then the thrown exception is further thrown into an outer *try* that may have a *catch* clause to handle it.

If no exception occurs during the execution of the code in the *try* block, all the *catch* clauses are simply ignored (not executed) .

Let's look at a specific example of catching exceptions. Consider the code below:

```
public void read(String fileName) {
       try {
          InputStream in = new FileInputStream(fileName);
          int b;
          while ((b = in.read()) != -1) {
              //process input
                     }
            } catch (IOException e) {
                           e.printStackTrace();
            }
       }
```

The in. read() method in the code above is the one which will throw an IOException. In this case, the stack trace of the exception will be printed.

Sometimes you may want to do some actions whether or not an exception is thrown, anyway. For that purpose, the finally clause is used: The rule is that the code inside of the finally will always be executed, even if an exception is thrown from within the try or catch block. What is more, even if in your try-catch block there is a return statement, then the code inside the finally will run before returning from the method. Let's see an example :

```
Graphics g = image.getGraphics();
try {...}
catch (IOException e) {...}
finally {
   g.dispose();
     }
```

In the example above, we dispose the *Graphics* object in a *finally* block, since anyway (whether exception happened or not) you need to do it to release the memory. Frequently, *finally* block is used to close streams, connections to databases, dispose some objects, etc.

9.4 Claiming Exceptions

When we discussed try-catch block we considered the read() method. In the catch block we put the code which we needed to execute in case exception

occurs. Another choice for this situation is to do nothing but simply pass the exception on to the caller of the method. This refers to exception *claiming* .

In Java method can claim an exception via the keyword *throws*. This needs to be put at the end of the method's prototype (before the definition), as in the example below:

```
public void myMethod() throws IOException
```

You can also claim multiple exceptions, just separate them with a comma:

```
public void myMethod() throws IOException,
                              otherException
```

So, let's redo the example mentioned above:

```
public void read(String fileName) throws IOException {
       InputStream in = new FileInputStream(fileName);
       int b;
       while ((b = in.read()) != -1) {

                              ///process input
       }
}
```

9.5 Throwing Exception

Programmers usually throw an exception under some bad situations. Consider an example below for the clarification.

Suppose you have a method named *readData* , which reads file whose header says it contains 550 characters, but it encounters the end of the file after 500 characters. You decide to throw an exception when this bad situation happens by using the throw statement. EOFException is used to signals that an end of file has been reached unexpectedly during input. By the way, you can throw the exception in two ways:

```
throw (new EOFException()); // 1st way
EOFException e = new EOFException(); // 2nd way
throw e;
```

Usage (case is discussed above) :

```
String readData(Scanner in) throws EOFException {
       String s = "";
       while(. . .) {
              if (!in.hasNext()) {
              //EndOfFile encountered {
                     if(n < len)
                            throw (new EOFException());
                      . . .
                                   }
                     }
                     return s;
              }
```

Do not get confused with claiming, throwing, catching exceptions. Figure X below will help you to finally fix these concepts:

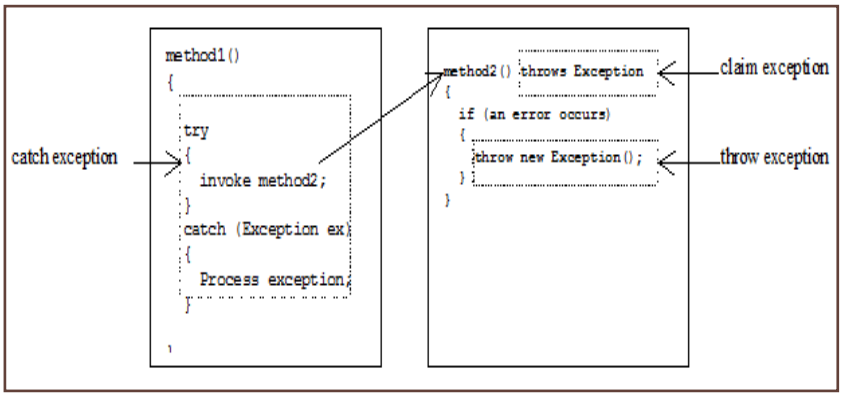


Figure 47. *Catching, Throwing, Claiming Excepitons*

Weighing everything up, we can conclude that exception handling is very useful in the sense that it separates error-handling code from normal programming tasks, making programs much easier to read and to modify. However, in most cases, exception handling requires the allocation of time and resources, since it needs to instantiate a new exception object, roll back the call stack, etc.

KEY CONCEPTS YOU NEED TO GRASP FROM THE SECTION

Stepwise refinement *Robustness* *Portability* *JVM*
Naming conventions *bytecode* *JIT* *Interpretation*
API *Eclipse IDE* *Primitive type* *Reference type*
Overloading *Methods* *Variables* *Scanner*

TESTS

1. Which of the statements below demonstrates how to *throw* an exception?

a) method() throws Exception; b) try{…}(Exception e){…};

c) try{…}catch(Exception e){} d) Exception a = new Exception();throw e;

2. Which of the statements below demonstrates how to *catch* an exception?

a) method() throws Exception; b) try{…}(Exception e){…};

c) try{…}catch(Exception e){} d) Exception a = new Exception();throw e;

3. Which of the statements below demonstrates how to *claim* an exception?

a) void run() throws Exception; b) try{…}(Exception e){…};

c) try{…}catch(Exception e){} d) Exception a = new Exception();throw e;

4. Which class is the root class in Java exception hierarchy?

a) Error b) Exception

c) Error d) Object

5. Choose an *unchecked exception*?

a) NullPointerException
b) IOException
c) FileNotFoundException
d) CloneNotSupportedException

PROBLEMS

1. In this part of the lab you will consider a simple example of exception handling.

```
public class Example1{
	public static void main(String[] args){
		int	denominator, numerator, ratio;
		numerator   = 5;
		denominator = 2;
		ratio = numerator / denominator;
		System.out.println("The answer is: "+ratio);
		System.out.println("Done."); // Don't move it
	}
}
```

1. Create class Example1.java in your developing environment.

2. Compile and execute the application Example1.
   What was output by the application when you executed it?

3. Change the value of denominator to 0.

4. Re-compile and re-execute Example1.
   What "error" was generated by the application when you executed it? Why was this "error" generated at run-time (rather than at compile-time)?

5. Add a try-catch statement. Specifically, put only the statement that generated the exception inside of the try block and put no statements in the catch block. (Hint: You should be able to determine what exception to catch and what line generated the exception from the error message that you received during the previous step.)Re-compile Example1.
   What error is generated and why?

6. Move the "output statement" into the try block (as well).

7. Add the statement System.out.println("Divide by 0."); to the catch block. Re-compile and re-execute Example1.
What output was generated?

8. Add a call to the printStackTrace() method of the *ArithmeticException* to the end of the catch block. Re-compile and re-execute Example1.
What output was generated?
Did the application execute properly or not?

2. This part of the lab considers an example of exception handling within and outside of block statements.

```
public class Example2{
       public static void main(String[] args){
              int         i, ratio;
              int[]       numbers = {100,10,0,5,2,8,0,30};
              try{
                 for (i=0; i < numbers.length-1; i++){
                     ratio = numbers[i] / numbers[i+1];
                     System.out.println(numbers[i]+"/"+
                                 numbers[i+1]+"="+ratio);
                     }
              }
              catch (ArithmeticException ae){
                 System.out.println("Couldn't calculate "+
                           numbers[i]+"/"+numbers[i+1]);
              }
       }
}
```

1. Write the code for Example2.java and compile it.
What error was generated?

2. Initialize i to 0 inside of the try block (but before the for loop).

3. Compile Example2.
What error was generated?

4. It is not possible for i to be used before it is initialized. Why is this error generated anyway? (Hint: Think about block statements.)

5. Move the initialization of i before the try block.

6. Compile and execute Example2.
   What output is generated?
   Why aren't all of the divisions even attempted?

7. Fix Example2 so that it executes properly. (Hint: Move the try-catch block inside of the for block.) What did you change? What has happened?

3. This part of the lab considers an inappropriate use of exception handling and how to "fix" it.

```
public class Example3{
       public static void main(String[] args){
              int i;
              int[] data = {50, 320, 97, 12, 2000};
              try {
                     for (i=0; i < 10; i++){
                            System.out.println(data[i]);
                     }
              }
              catch (ArrayIndexOutOfBoundsException aibe){
                     System.out.println("Done");
              }
       }
}
```

1. Compile and execute Example3 and verify that it outputs all of the values followed by the word "Done".
2. Modify Example3 so that it loops "properly" and does not need to use a try-catch statement. (Note: The output should not change.) **What did you change?**

4. This part of the lab will give you some experience with some other exceptions, where they arise, and how they can be used.

```
public class Example4{
   public static void main(String[] args){
       double leftOperand, result, rightOperand;
       String leftString, operator, rightString;
       StringTokenizer tokenizer;
       Scanner in = new Scanner(System.in);
       tokenizer = new StringTokenizer(in.nextLine(),
                                       "+", true);
       try{
          leftString   = tokenizer.nextToken();
          operator     = tokenizer.nextToken();
          rightString  = tokenizer.nextToken();
          leftOperand  = Double.parseDouble(leftString);
          rightOperand = Double.parseDouble(rightString);
          if (operator.equals("+"))
              result = leftOperand + rightOperand;
          else result = 0.0;
               System.out.println("Result: " + result);
           }
        catch (NoSuchElementException nsee){
             System.out.println("Invalid syntax");
           }
        catch (NumberFormatException nfe){
           System.out.println("Operand(s) is not a number");
           }
       }
}
```

1. What functionality does a StringTokenizer object provide? Give example.

2. What are the three formal parameters of the explicit value constructor in the StringTokenizer class? Give example.

3. Create and run Example4.java

.

4. After running the program write the following to the command line: 5.3+9.2
   What output is generated?

5. Now run it again and enter the following: 5.3+
   What output is generated?

Why? In particular, what exception is thrown and why?

6. Run again and enter 5.3+a.
   What output is generated?
   Why? In particular, what exception is thrown and why?

5. Modify Example4.java so that it supports addition (+), subtraction (-), multiplication (*), and division (/). In addition, modify Example4.java so that it processes more than just one expression for evaluation. So, for example, it should be able to be execute the following input:

```
45.0+4.1 3.2*9.1.
```

Modify Example4.java so that it tells you which operand is not a number. (Hint: You may need to use nested try-catch blocks.)

10 Recent Advances in Component Software – Does Scala Beat Java?

"Pure functional languages will make your brain hurt. People that like them are the type of people that liked calculus in school."
James Gosling

Nowadays it is admitted that the lack of progress in component software results from shortcomings in the programming languages used to define and integrate components. Particularly, most existing languages provide only limited support for component abstraction and composition (10). This holds in particular for statically typed languages such as Java and C# in which much of today's component software is written. Nevertheless, the necessity of creating the large projects demanded programming abstractions that allow components of a project to be reused. The key enabler for that is encapsulation that allows to hide the implementation details, which can be exploited by abstracts classes and abstract data types. Second order existential quantification can be used to model the abstract data types. In turn, the Curry Howard correspondence (discussed further in 1st section) can be extended to cover encapsulation. Furthermore, Mitchell and Plotkin's discovery allowed the concept of abstraction to be adopted by functional programming languages. Polymorphism (subject of discussion of subsection 2) and abstract data types (subsection 3) together provide powerful data abstractions.

This chapter seeks to consider in details the issues discussed above on the example of Scala PL. Scala has been developed around 2004. It stems from a research effort to develop better language support for component software (11). It is designed to be SCALAble in the sense that the same concepts can describe small as well as large parts. Moreover, in order to ease its adoption, the new language needs to integrate well with existing platforms, since the majority of components that can be reused are based on those platforms. Scala fits this bill, since it has been designed to work well with Java and C#. It adopts a large part of the syntax and type systems of these languages. Finally, it is important to note that scalable support for components can be offered by language which fuses object-oriented and functional programming. For statically typed languages (e.g. Scala, Java) these two paradigms were *up to now* strongly separate.

10.1 From Deductive Systems to Programs

There exists a *Curry Howard Correspondence* that influences the majority of modern programming languages. The main idea of this principle lies in the existence of the precise correspondence between the deductive proofs in logical systems and well-typed programs. So, each formulae of minimal propositional logic can be mapped into a respective feature in functional programming language. This mapping is provided in the table below.

Minimal Propositional Logic	Functional Programming
Implication $(A \rightarrow B)$	The Function Type
Conjunction $(A \times B)$	The Tuple Type
Disjunction $A+B)$	The Tagged Union

Table 15. Specifying relationships among packages

Furthermore, for each of these formulae and for each type there is an *introduction* and an *elimination* rule. This correspondence serves as a basis of the Curry Howard Correspondence between minimal propositional logic and the simply typed λ-calculus. More commonly called a "function", a lambda abstraction is a value that abstracts over another value. In addition, for each of the rule presented below we can find the corresponding feature in functional language (Scala in our case):

Minimal Propositional Logic	Functional Programming	Scala
Implication Introduction	Lambda abstraction – $\lambda x: A.t$	Defining the function
Implication Elimination	Function application – $t\ u$	Call to the function
Conjunction Introduction	The pair constructor – (t, u)	The Tuple constructor
Conjunction Elimination	Left and Right projection – *left t, right t*	Getting the Tuple item
Disjunction Introduction	Left and Right injection – *inl t, inr t*	Either type
Disjunction Elimination	The case statement – case $(t, \lambda x: A.u, \lambda y:B.v)$	The case statement

Table 16. Mapping of rules in Propositional Logic to features in Scala

It is obvious that lambda abstraction corresponds to the definition of function in any programming language, and Scala is not the exception. As an illustration, let's create a function that increases the integer passed to it and returns it in turn:

```
//implication introduction
def inc(num: Int): Int = num+1
```

As regards the implication elimination, it corresponds to a function call, as we've already stated above:

```
//implication elimination
inc(8)
```

Turning to the conjunction, introduction of it corresponds to the *Tuple* constructor in Scala, while the elimination is reflected into accessing the left or right item of the *Tuple*. Note that a *Tuple* groups together simple logical collections of items without using a class:

```
//conjunction introduction
def hostPort = ("localhost", 80) // or "localhost" -> 80
def Point = (50,5)
//conjunction elimination
hostPort._1 // "localhost"
Point._2    // 5
```

Concerning the disjunction, it was a bit obscure which feature in Scala corresponds to it. Nevertheless, after a small research I discovered an *Either* type. *Either* represents a value of one of two possible types (a disjoint union). Instances of *Either* are either an instance of *Left* or *Right* . Obviously, *Either* represents the disjunction introduction:

```
//disjunction introduction
type myType = Either[String, Int]
val left: myType = Right(8)
```

For instance, you could use defined **myType** (**Either[String, Int])** to detect whether a received input is a *String* or an *Int*. This corresponds to the disjunction elimination:

```
//disjunction elimination
def PrintRightOrLeft(l:myType ):Unit = l match {
       case Right(x) => println("Right, Integer: " + x)
       case Left(x) => println("Left, String: " + x)
       PrintRightOrLeft(left)
}
```

By conducting this small research I eventually understood the connection between the minimal propositional logic and functional programming language. Moreover, I learned about some peculiar data types and features Scala provides. I feel this might be useful for me in the future.

10.2 Polymorphism and Every Day Programming

The vast majority of modern programming languages nowadays are polymorphic. What is more, polymorphism is a core feature for a number of these languages, like Java and C#. Polymorphism was revealed by using an extension of the λ-calculus called System F, which refines the λ -calculus with variables for types and quantifiers over type variables. It is notably that the expressiveness of System F allows to use this system for programming of any everyday function.

As it is well known, there are two general approaches to polymorphism - *type checking* and *type inference*. Actually, Scala uses a mix of both type checking and type inference, with the strong emphasis put on type checking and providing only local type inference. In essence, type checking means that an algorithms checks if the program is well typed using the type annotations that program has.

The polymorphic type system of Scala relies on an extension of *System F* that allows sub-typing called *System F<*. *System F* or λ-calculus with polymorphic types – is a striking real-life illustration of the Curry-Howard isomorphism, which relates logic and PLs (12). In a few words, it states that a type is a proposition, and a proof of that proposition corresponds to a term that is classified by that type. Thus, type checking and proof checking are strongly related. Fragments of system F have served as the basis for many polymorphic PLs. The most notable example, besides Scala is Haskell PL. It's extension *F<* combines parametric polymorphism with subtyping.

In *F<* subtyping is reflected in the syntax of types by a new type constant *Top* (the supertype of all types, maximal type), and by a subtype bound on second-order quantifiers : *∀(X<:A)A'* (*bounded quantifier*). Ordinary second-order quantifiers are recovered by setting the quantifier

bound to *Top*. We use $\forall(X)A$ for $\forall(X<:Top)A$. Clearly, there is a subtype bound on polymorphic functions, $\lambda\ (X<:A)a$. We use $\lambda\ (X)a$ for $\lambda\ (X<:Top)a$ (12). The main idea behind the additional terms is that we can change the type of any argument not used in the body of a term to *Top*, and still have a term of the same type .

Since polymorphic functions operate independently of their type parameter, they may be considered equivalent at all their type instances. In *F<:* we state that whenever two type instances have a common supertype, they will be equal when considered as elements of that supertype.

Some of the syntax rules of System F< together with their meaning and equivalent in Scala are provided in the table below.

Notation	Meaning	Example (in Scala)
A, B ::= ; a,b ::=	Types ; Values	`type IntRow = List[Int]`
X ; x	Types variables ; Value variabes	`def b : IntRow = List(1,2,3)`
Top	The supertype of all types	`Scala.Any class`
A→B	Function spaces	`Int=>Unit`
∀(X<:A)B	Bounded quantifications	`type T <: Ordered`
λ (X:A)b ; b(a)	Functions; Applications	`def sub(num: Int): Int = num-1; sub(7)`
λ (X<:A)b ; b(a)	Bounded type functions	`def do[T](t: T):String = t.toString()`

Table 17. Syntax Rules of System F<

Consequently, now we can define some basic judgments based on the syntax defined. Specifically, ***E |- A type*** means that *A* is a type in environment *E*, while ***E |- A <:B*** defines that *A* is a subtype of *B*. The Scala equivalent for this judgment is provided below (Frog is a subtype of Animal):

```
class Frog extends Animal {
       override def toString = "green"
                       def Numlegs: Int = 4
}
```

Additionaly, the piece of the code above that defines Numlegs variable of type Int represents the next judgment , ***E |- a:A,*** meaning that a has type A. In our case, Frog has type Int.

Another useful notations in System F< are ***B{X←C}*** for the substitution of C for X in B, and ***FV(-)*** for sets of free variables. These rules are used to define the following equality judgments:

$$\begin{array}{lll} \forall(X<:A)B & \equiv \forall(Y<:A)\ B\{X\leftarrow Y\} & \text{where } Y \notin FV(B) \\ \lambda(x:A)b & \equiv \lambda(y:A)\ b\{x\leftarrow y\} & \text{where } y \notin FV(b) \\ \lambda(X<:A)b & \equiv \lambda(Y<:A)\ b\{X\leftarrow Y\} & \text{where } Y \notin FV(b) \end{array}$$

So, we see that a subtyping judgment is added to *F* 's judgments. Moreover, the equality judgment on values is made relative to a type, and that is highly important since values in *F*< can have many types, and two values may or may not be equivalent depending on the type that those values possess. As it can be seen, the essence of the subtyping judgment is that a member of a type is also a member of any supertype of that type. The rules for the typing judgment, ***E |- a : A***, are the same as the corresponding rules in *F*, except for the extension to bounded quantifiers (12). However, additional typing power lies in the subsumption rule, which allows a function to take an argument of a subtype of its input type.

Most of the equivalence rules provide symmetry, transitivity, congruence on the syntax, and *Top-collapse* rule, which states that any two terms are equivalent when "seen" at type *Top*; since no operations are available on members of *Top*, all values are the same at that type. These rules can be seen below:

$$\text{(Sub refl)}\ \frac{E \vdash A\ type}{E \vdash A <: A} \qquad \text{(Sub trans)}\ \frac{E \vdash A<:B \quad E \vdash B<:C}{E \vdash A <: C} \qquad \text{(Sub Top)}\ \frac{E \vdash A\ type}{E \vdash A <: Top}$$

As we've already mentioned, 1st rule represented above is Reflexive rule (every type is a subtype of its own type), 2nd one is transitive rule, the last one – Top collapse rule.

The other important standard rules - about subsumptions, functions and applications are given below:

$$\text{(Subsumption)}\ \frac{E \vdash a:A \quad E \vdash A<:B}{E \vdash a : B} \qquad \text{(Val fun)}\ \frac{E,x:A \vdash b:B}{E \vdash \lambda(x:A)b : A\rightarrow B} \qquad \text{(Val appl)}\ \frac{E \vdash b : A\rightarrow B \quad E \vdash a:A}{E \vdash b(a) : B}$$

The standard encoding for pairs in *F* is shown below:

$$A \times B \triangleq \forall(C)(A \to B \to C) \to C$$

Furthermore, the usual operations on pair are defined as follows:

$$pair: \quad \forall(A)\ \forall(B)\ A \to B \to A \times B\ ,$$

$$fst: \quad \forall(A)\ \forall(B)\ A \times B \to A\ ,\quad snd: \quad \forall(A)\ \forall(B)\ A \times B \to B$$

As regards the tuple, it represents an iterated product type, where *n* is more or equal to 1 (*n* is the number of items in a tuple). The basic useful operation that tuple provides is ***a.i*** – selecting the i-th item of *a*. It is of interest to note that these conventions hold exactly in Scala. As an illustration, consider creating a tuple to hold x,y,z coordinated in 3D space.

```
def Point = (883, 8 , 3)
Point._1 // 883
Point._3 // 3
```

Concerning the decidability of type-checking in Scala, it is at present too hard to give a definite answer since full Scala is too complicated to admit a formalization of its type system which is complete yet still manageable enough to admit a proof of decidability.

One of the design goals for Scala lies in interoperability with languages like Java. Obviously, this requires Scala's type system compatibility. In particular, this means that Scala needs to support subtyping and overloading (12) such as:

```
def add (x : Int, y : Int) : Int = //
def add (x : String, y : String) : String = //
```

However, this makes type inference difficult. Nevertheless, Scala does support local type inference. Specifically, it is possible most of the time to infer the return type of a definition and the type of a lambda variable. For example, an alternative definition to power would be: **def power (x : Int) = twice (y => y *y, x)** in which both the return type and the type of the lambda variable *y* are inferred. Essentially, unknown types are replaced by type variables that record the constraints that must be met in order for type checking to succeed. The solution of these constraints determines the missing type (12). Local type inference needs to construct least upper

bounds and greatest lower bounds of sets of types. Here the problem of infinite approximations of lub's and glb's arises. The Scala compiler tackles this problem by imposing a limit size on the types computed by its lub and glb operations. It is currently set at 10 levels. If a type computed by lub or glb exceeds this limit the system will reply with an exception . As a result, Scala compiler turns the potential problem of undecidability of type inference into another problem: local type inference might now fail to give a solution even if a best type would exist. Nevertheless, it is always possible to add more type annotations, so that infinite approximations are avoided.

As it is well-known, in a statically typed language, such as Scala, types verification is performed before the program is run. This is in contrast to Clojure which is dynamically typed - the types are checked at runtime. Verification deals with all possible executions of the program and detects all violations against the specifications that it can express. The trade-off is that verification may dismiss valid programs.

Furthermore, as it can be easily guessed from the fact that Scala is based on *System F*, it supports parametric polymorphism, known as generics in the object-oriented world (11). First-order parametric polymorphism is now a standard feature of statically typed PLs (12). Starting with *System F* and functional PLs, the constructs have found their way into object-oriented languages such as Java, C#, and many more.

In fact, generics can be seen as a generalisation of the type of arrays. Similarly, a polymorphic class or method can be expressed in terms of its type parameters, which represent unknown types that must be made concrete by the clients of the class or method. This extension fits nicely with the Scala philosophy of fusing functional and object-oriented programming. One standard application area of generics is collections. As an illustration, the type **List[T]** represents lists of a given element type *T*, which can be chosen freely, the polymorphic class of lists is not sensitive to the exact type of its elements. Now, with parametric polymorphism this kind of consistency can be obtained without defining a new list for every specific type. A polymorphic list simply abstracts over the concrete type of its elements using a type parameter. To ensure consistency, a list of elements of type *T* only allows another element of type *T* to be added to the list. As a result, when the first element of a list of elements of type *T* is retrieved, the list may guarantee that the element has type *T*. Without parametric polymorphism, the list can only be expressed to hold any object (using subtype polymorphism), and, for every interaction, the user had to check (through casting) that the expected type of object was retrieved. No doubt, the integration of genericity in an object-oriented language makes the type system more powerful.

So, Scala, similar to Java, supports the generics, or parameterized types. In essence, it works more or less like in Java, with a slightly different sytax and a more concise means of use, thanks to Scala's type inference. Here is an example which demonstrates this:

```
class Stack[T] {
       var items: List[T] = Nil
       def push(x: T) { items = x :: items }
       def top: T = items.head
       def peek: T = items.tail
       def pop() { items = items.tail }
}
```

Class *Stack* models stacks of an arbitrary type *T*. Using type parameters allows to check that only legal elements (of type *T*) are pushed onto the stack. Here are some usage examples:

```
val stack = new Stack[Int]
stack.push(1)
stack.push('d')
println(stack.top)//100
stack.pop()
println(stack.top)//1
```

We need to emphasize that subtyping of generic types is *invariant*. This means that if we have a stack of characters of type Int then it cannot be used as an stack of type *Char*.

In addition, methods in Scala can also be genericized, as they can be in Java:

```
def doit[T](thing: T):String = thing.toString
```

A static type system is an important tool in efficiently developing correct software. The recent introduction of "genericity" in object-oriented programming languages has greatly enhanced the expressiveness of their type systems (12). Genericity, which is also called "parametric polymorphism", is extremely useful as it allows the definition of polymorphic lists, which use a type parameter to abstract over the type of their elements. The beauty of that is that the user of these abstractions need not worry about their inner workings. It is of interest to note that besides

having a common ancestor - *Top*, Scala also introduces *Nothing* type, which is the subtype of all types.

10.3 Mixing Approaches to Encapsulation

As it was stated in the introduction, encapsulation can be considered as a key enabler for components packaging and reuse. The crucial idea about encapsulation is that it allows to hide implementation details. Particularly, abstract data types provide a general type signature for a data type without providing exact implementation details. Subsequently, these abstract data types can be implemented in several ways and hidden from the user. Therefore, the expert programmer can encapsulate domain knowledge so that the mainstream programmer, as the user of these abstractions, may soundly instantiate them (10)**.**

Scala offers a new model for component systems. Essentially, components in this model are classes, which can be combined using nesting and mixin composition. In addition, classes can contain abstract types which may be later instantiated in subclasses. The advantage of this approach is that a relatively small set of language constructs is sufficient for core programming as well as the definition of components and their composition. Scala's component constructions provide a middle ground between the worlds of object-oriented programming and functional module systems (11)**.**

Going deeper, Scala has a notion of abstract types, which provide a flexible way to abstract over concrete types used inside a class or trait declaration. Abstract types are used to hide information about the implementation of the component. As with any other kind of class member, abstract types in a class must be given concrete definitions before the class can be instantiated. So, for instance, List could also be defined as a class with an abstract type member instead of as a type-parameterised class, like:

```
abstract class List { type Elem }
```

Then, a concrete instantiation of List can be defined as :

```
List {type Elem = String}
```

Furthermore, classes may contain *type* members. An abstract type member is similar to a type parameter. The main difference between

parameters and members is their scope and visibility. So, a type parameter is part of the type, meanwhile a type member is encapsulated. In addition, type members are inherited, whereas type parameters are local to their class. Type parameters are made concrete using type application. The complementary strengths of type parameters and abstract type members are a key ingredient of Scala's recipe for scalable component abstractions (10).

In contrast to languages that only support single inheritance, Scala has a more general notion of class reuse. Scala makes it possible to reuse the new member definitions of a class in the definition of a new class. This is done through a mixin-class composition (11). As an illustration, consider the following abstraction for iterators[17].

```
abstract class AbstractIterator {
        type T
        def hasNextElement: Boolean
        def next: T
}
```

Next, consider a mixin class which extends *AbstactIterator* with a method *foreach* which applies a given function to every element returned by the iterator. To define a class that can be used as a mixin we use the keyword *trait*.

```
trait IteratorWithForEach extends AbstractIterator {
        def foreach(f: T => Unit) {
                while (hasNextElement) f(next)
        }
}
```

Here is concrete iterator class, returning successive characters of a string:

[17] The presented example is not the best practice in functional programming, it is presented just for the sake of explaining mixin compositions. For further reading, refer to (10), (11), (12).

```
class SimpleStringIterator(cur: String)
                              extends AbstractIterator {
        type T = Char
        private var index = 0
        def hasNextElement = index < cur.length()
        def next = {
         val temp = cur charAt index; index += 1; temp
                    }
}
```

Now suppose we would like to combine the functionality provided by of *SimpleStringIterator* and *IteratorWithForEach* into a single class. With single inheritance and interfaces (like in java or c#) this is totally impossible, as both classes contain member impementations with code. Hovewer, in Scala, it is reality by virtue of its mixin-class composition, that allows to reuse the delta (all new definitions that are not inherited) of a class definition. This mechanism makes it possible to combine our classes - *SimpleStringIterator* with *IteratorWithForEach*, as is done in the following test program which prints all the characters of a given string.

```
class MyCoolIterator
                extends SimpleStringIterator("bla-bla")
                with     IteratorWithForEach
val iterator = new MyCoolIterator
iterator foreach println
```

The *MyCoolIterator* class is constructed from a mixin composition of the parents *SimpleStringIterator* and *IteratorWithForEach* with the keyword **with**. The first pancestor is called the superclass of *MyCoolIterator***,** whereas the second one – is a mixin. This example demonstrates that merging polymorphism and abstract data types provides powerful data abstractions.

So, as we have seen, in Scala, a class can inherit from another class and one or more traits. A trait is a class that can be composed with other traits using mixin composition - a restricted form of multiple inheritance (10). The main difference between an abstract class and a trait is that the latter can be composed using mixing inheritance. Another difference is that traits cannot define constructors. Also, an intersection type, such as A *with* B can be understood as the type that is a subtype of both A and B (12).

So, we learned that instead of interfaces, Scala has the more general concept of traits. Like interfaces, traits can be used to define abstract methods. Nevertheless, in contrast to interfaces, traits can also define

concrete methods (12). Let's look at some other examples illustrating that traits can be combined using mixin composition, making a safe form of multiple inheritance possible:

```
trait Hello {
       val hi = "Hello"
}
trait Curious {
       val question = "Do you like apples?"
       def ask() = println(question)
}
trait Angry {
       def shout(s: String):Unit
}

trait AngryPerson extends Hello with Curious with Angry{
      val greeting = hi+ ", "+question
      def shout(s: String) =
                println(s.toUpperCase()+" !!!")
}
```

In the example provided above, we use traits to declare both abstract methods like *shout()* and concrete methods like *ask()*. In Java or C#, for instance it would not be possible to define a subclass that combined the functionality of the four code blocks above. Nevertheless, mixin composition allows any number of traits to be combined: the trait *AngryPerson* inherits methods from *Hello*, *Curious*, implements the method *shout()* from *Angry*, and defines a value greeting combining the functionality of it's both ancestors.

Now let's look at another appealing example of mixins. Suppose we have the following Traits: *Student, Employee, Mother, Young , ChessPlayer*. How could we declare a class *BuzyPerson* with all these traits?

```
class BuzyPerson extends Student
                  with Employee
                  with Mother
                  with Young
                  with ChessPlayer
```

As it can be seen from the code above, it is easy to do, since when declaring a class you just use the "with" keyword as often as you want. Also

if you don't want to have a class which always uses the same traits you can use them later:

```
class YoungMother extends Young with Mother
val Pakita = new YoungMother with Employee with Cooker
```

Finally, let's create a trait *MyBool* that mimics the pre-defined booleans. It is of interest to note that such operators as "||" and "&&" can also be represented as methods because Scala allows to pass arguments by name. Table 18 below provides the trait *MyBool* itself as well as its two canonical instances.

As can be seen in these implementations, the right operand of a && operation is evaluated only if the left operand is the True object. Respectively, the right operand of || operation is evaluated if the left operand is False. So, it is possible in Scala to define every operator as a method and treat every operation as a call to a method.

MyBool trait	False	True

trait MyBool {
def && (x: MyBool): MyBool
def || (x: MyBool): MyBool
}
	object False extends MyBool {
	def && (x: MyBool): MyBool = thi
	def || (x: MyBool): MyBool = x;
	}
		object True extends MyBool {
		def && (x: MyBool): MyBool = x;
		def || (x: MyBool): MyBool = thi
		}

Table 18. MyBool Trate

Now let's discuss the interoperation between Scala and Java. As we mentioned earlier, Scala is aimed at industrial usage: a key design goal of Scala is that it should be easy to interoperate with mainstream languages like Java and C#, making their many libraries readily available to Scala programmers (10). That is why it is currently implemented on the Java and .NET platforms. It shares with these languages most of the basic operators, data types, and control structures.

In particular, Scala adopts most of Java's control structures and constructions like classes, abstract classes, subtyping and inheritance, but it lacks Java's traditional for-statement. Instead, there are for-comprehensions which allow one to iterate directly over the elements of the list without the need for indexing. Additionally, in Scala, constructor parameters follow the class name, so there is no separate class constructor definition within the body. Scala also incorporates some less common concepts. In particular, there is a concrete notion of object, and interfaces are replaced by the more

general notion of traits, which can be composed using a form of mixin composition, as we have already seen (11). Finally, certain types are spelled differently for Java and Scala: **Object** becomes **Any**, **Unit** is a regular type that corresponds to the **void** keyword in Java, and **Nothing** is the subtype of all types, which cannot be expressed in Java. However, in general, Scala syntax is relatively familiar for Java programmers. Consider the code below:

```
if (name startsWith "A") // syntactic sugar
       System.out.println("A"+name.substring(1));
```

This small example proves that even though their syntax is different, Scala programs can interoperate without problems with Java programs. In the example above, the Scala program invokes methods *startsWith* and *substring* of **String,** which is a class from Java library. It also accesses the static *out* field of the Java class System, and invokes its *println* method.

Moreover, Scala classes and objects can also inherit from Java classes and implement Java interfaces. This makes it possible to use Scala code in a Java framework (10). In general, you can use Scala classes from Java (as well as Java classes from Scala) without ever even knowing that they were defined within another language. The key enabler for that is that, despite the very different syntax, both Scala and Java programs produce almost identical bytecode when compiled. As a simple illustration, let's create a simple class in Scala:

```
class Person {
       var Name = "No name"
       def getDescription() = "Name is: "+ Name
}
```

Subsequently, after compilation of this class, the class file, Person.class, containing the byte code, will be created. Whenever you are a Java, Scala or c# programmer, you can be a user of this class. Let's look how to use it in Java:

```
Person p = new Person();
String desc = p.getDescription();
```

As a result we can conclude that Scala has uniform and powerful abstraction concepts for both types and values. Moreover, it has flexible symmetric mixin-composition constructs for composing classes and traits.

Finally, Scala programs resemble Java programs in many ways and it is highly useful that they can interact with code written in Java.

10.4 Have Object-Oriented and Functional Languages Converged?

In conclusion, we can say that having done this small research about capabilities of Scala and having compared it with Java PL, we can say that Scala aims to unify object-oriented and functional programming. This relatively recent language provides a smooth integration of the functional and object-oriented paradigms. Not only does Scala provides equivalents of all the necessary functional programming features, like folding, higher-order functions, parametric polymorphism and type- and constructor-classes, but it also provides the most useful features of object-oriented languages, such as overriding and overloading, subtyping, traditional single inheritance and multiple inheritance in the form of mixins. Therefore, some people claim that Scala is object-oriented language that has some functional features. Yet others put forward the view that Scala is a functional language that happens to have object-oriented features. Indeed, it offers the best of both worlds. So, it has a unique position as being both a functional and an object-oriented language.

As we know, Scala is aimed at the construction of components and component systems. Obviously, that was the main motivation for fusing object-oriented and functional programming in a statically typed programming language. So, answering the question put in this task - “Have Object-Oriented and Functional Languages Converged?”, we can say: “Yes, they have converged. In Scala programming language”.

Conclusion

Once you have finished reading this book the core concepts constituting object-oriented programming should be well understood. Nevertheless, there are still a number of issues where there is no consensus in object-oriented community. The current challenges in Object-oriented Programming which you may face applying theory in real-world situations are:

- *Common conceptual framework* which underlies object--oriented modeling should be formulated independently of the languages.
- *Abstraction mechanisms*. Despite of the fact that there exist some basic agreement about the core abstraction mechanisms, there are still considerable differences between the languages. For example, single versus multiple inheritance, types versus classes, encapsulation, dynamic typing versus static typing, etc.
- *Data storage* represents a common challenge, since the majority of organizations use relational databases to store data, and relational databases are generally not suited for storing objects. You, as a programmer, need to be able to tackle conflicts of such kind. The best option is to use Object-relational Mapping[18] (e.g. Hybernate), which creates a kind of virtual object database. Other programmers prefer to solve this issue by using object-oriented databases, but generally it is a rare case when company uses an object-oriented database. The simplest possible way which is frequently used in practice is to write functions that translate objects to a relational database model, and vice versa. Balance the need to adhere to object-oriented design philosophy and the need for an uncomplicated application. For example, if the sole purpose of defining an object is to accommodate a procedure, then consider defining the procedure as a stand-alone procedure rather than defining another object. A stand-alone procedure is acceptable in an object-oriented program if it makes sense to the application.
- *Identifying objects.* Although identifying objects in the real world seems very natural and intuitive, in the world of programming it might become challenging since some real-world concepts can be interpreted as two or more objects. We advice to use “keep it

[18] Object-relational Mapping – technique used to convert data between incompatible type systems in oo languages. Particularly, between objects and table records.

simple" principle – avoid creating too many objects in your system, unless they are all critical, do not create tens of subclasses. Instead of that, you can create one object and highlight the distinctions between different kind of objects through fields and methods. As a simple illustration, consider a *Student* object. Because there are several types of students (e.g. undergraduate, graduate, postgraduate, etc.) you might want to define the class for each student. However, it is much better to use attributes for this purpose, that can describe the kind of student (through enumeration, for instance).

- *Complex Hierarchy.* Sometimes novices in programming with the wild enthusiasm create a complex hierarchy for their systems, missing the application's goal. Of course, multilevel inheritance is technically strong, but in practice it represents a challenge to maintain it. So, avoid creating unnecessary complex inheritance structures, especially in case it involves more than three levels.

As we have learned, object-oriented approaches highlight the importance of objects, in contrast to procedural, or imperative programming, which emphasizes the execution of sequential commands. In turn, functional programming emphasizes the definition of functions.

As you learned in the last chapter, nowadays, the most popular approaches to programming consider fusion of object-oriented and functional programming, which together provide powerful data abstraction mechanisms. The outstanding example for such approaches is recently born Scala programming language that was designed to work well with Java and C#. It adopts a large part of the syntax and type systems of these languages. Scala provides a smooth integration of the functional and object-oriented paradigms. Not only does Scala provides equivalents of all the necessary functional programming features, like folding, higher-order functions, parametric polymorphism and type- and constructor-classes, but it also provides the most useful features of object-oriented languages, such as overriding and overloading, subtyping, traditional single inheritance and multiple inheritance, which is represented in the form of mixins in Scala programming language.

Acknowledgements

It is pleasure for me to acknowledge the efforts of people whose names do not appear on the cover, but whose cooperation, help and understanding were crucial in the creation of this textbook:

Academic reviewers:

L.B. Atymtayeva, doctor of physical and mathematical sciences
I. M. Ualiyeva, candidate of physical and mathematical sciences
R. Zh. Satybaldiyeva, candidate of technical sciences
R. Horne, PhD in Computer Science

Book cover designer :
Aitpaeva A.A. – 2nd year FIT student

In addition, I want to express my gratitude to 2nd and currently 3rd year FIT students (FIT-11, FIT-10) taking the course "Object-oriented Programming and Design" in 2012-2013 and 2011-2012 academic years respectively. This book would not have been possible without their kind suggestions, motivating comments, challenging questions, reasonable corrections I was obtaining during the course.

We would sincerely appreciate all your comments, criticisms, corrections and suggestions for improving the text and examples. Please, feel free to address all comments to the following email address: **pakita.shamoi@gmail.com**

References

1. **Eck, David J.** *Introduction to programming using Java.* s.l. : Online book: http://math.hws.edu/javanotes, 2006.

2. **Cay S. Horstmann, Gary Cornell.** *Core Java, Volume 1 - Fundamentals.* s.l. : Prentice Hall (Sun Microsystems Press), 2008.

3. **Perry, J. Steven.** *Introduction to Java programming, Part 1: Java.* s.l. : IBM Corporation, 2010.

4. **R. Morelli, R.Walde.** *Java, Java, Java Object-Oriented Problem Solving.* s.l. : Pearson Education, Inc., 2012.

5. **Horton, Ivor.** *Beginning Java 2, JDK.* s.l. : Wiley Publishing, 2005.

6. **Deitel, H. M.** *Java - How to Program.* s.l. : Prentice Hall, 2004.

7. **Horstmann, Cay.** *Big Java.* s.l. : Wiley Publishing, 2008.

8. **James Gosling, Bill Joy, Guy Steele.** *Java Language Specification.* s.l. : Addison-Wesley, 2011.

9. **java.sun.com/j2se/1.4.2/docs/api.** *Java API documentation.* s.l. : Offcial java site.

10. **Odersky, Martin.** *An overview of the Scala programming language.* EPFL Lausanne, Switzerland : s.n., 2004. IC/2004/64.

11. **Prof. Dr. ir. W. JOOSEN, Prof. Dr. ir. F. PIESSENS.** *Type Constructor Polymorphism for Scala: Theory and Practice.* s.l. : Katholieke universiteit Leuven, 2009.

12. **Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov.** *An extension of system f with subtyping. .* s.l. : Theoretical Aspects of Computer Software, volume 526 of Lecture Notes in Computer Science, pringer Berlin Heidelberg,. , 1991.

13. *Open Issues in Object-Oriented Programming.* **Madsen, Ole Lehrmann.** Computer Science Dept., Aarhus University, Denmark : s.n.