

OBJECT-ORIENTED PROGRAMMING Using JAVA

ABSTRACTION

QUAN THAI HA

HUS

FEBRUARY 18, 2024



1 Abstraction

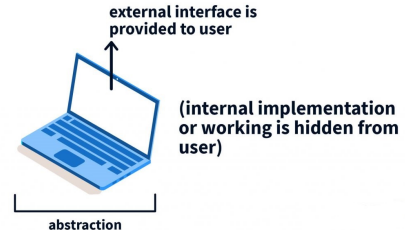
- Abstract Classes
- Interfaces
- Usage of abstract classes
- Usage of interfaces

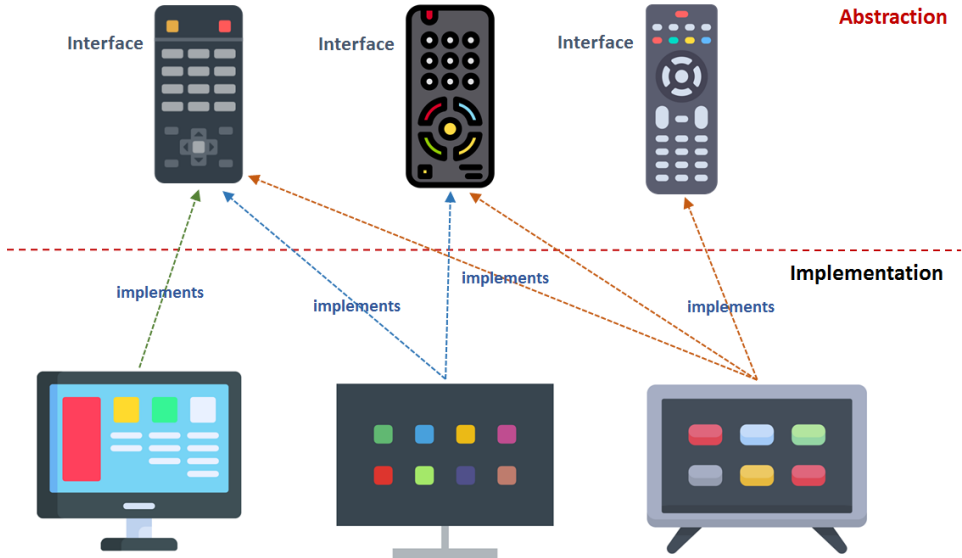
2 Abstract Data Types

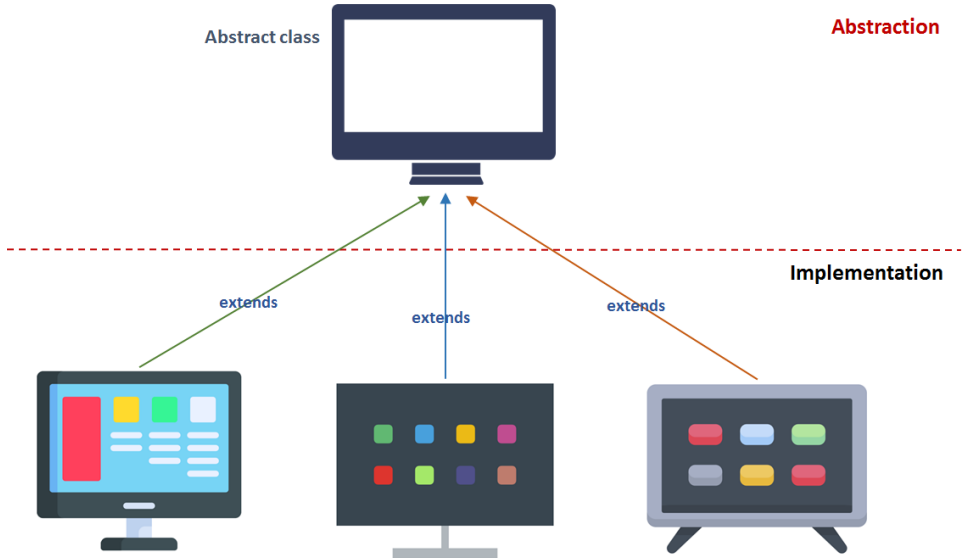
3 Modelling Tools and Languages

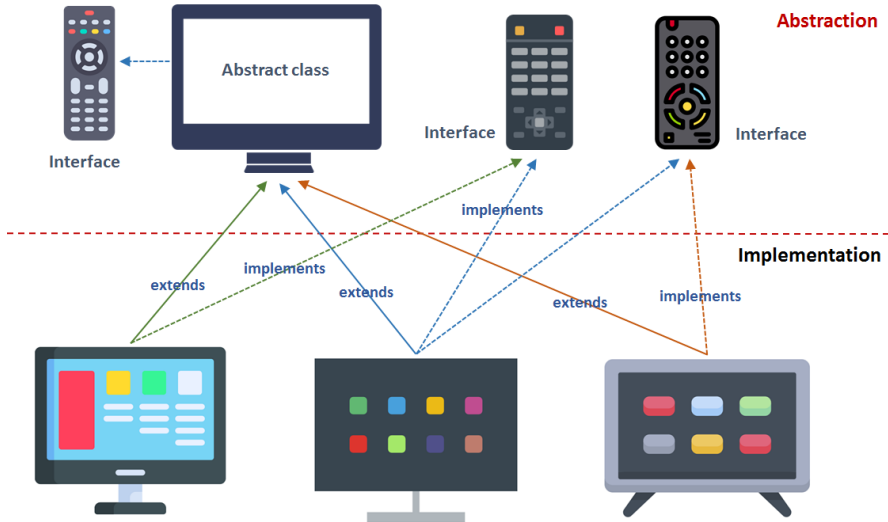
4 References

- Abstraction is the concept of object-oriented programming that **shows only essential attributes** and **hides unnecessary information**.
- The main purpose of abstraction is hiding the unnecessary details from the users (information hiding).
- Abstraction is selecting data from a larger pool to show only relevant details of the object to the user. It helps in **reducing programming complexity and efforts**.
- You can take a real-life example. On a laptop you can see the screen, use the keyboard and type something, you can use the mouse pad and can do all your work on the laptop. But do you know its internal details, how that keys are working and how the motherboard of the laptop is working, and how the picture is showing up on the screen? So here, all the internal implementation of the laptop is hidden from the user and it is separated from its external interface. You can use the laptop without knowing its internal implementation. So abstraction provides simple and easy-to-use interfaces for the user.

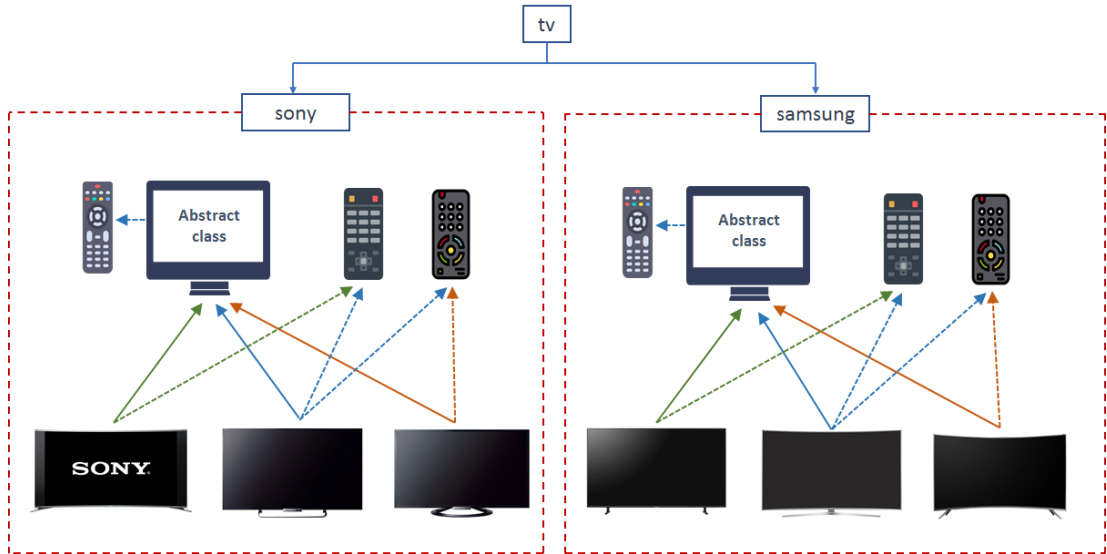




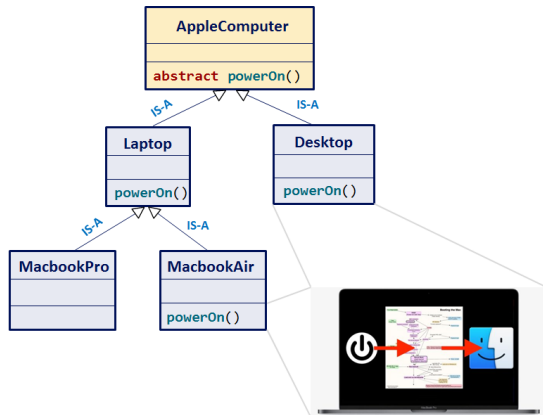




EXAMPLE OF DESIGNING A JAVA PROJECT

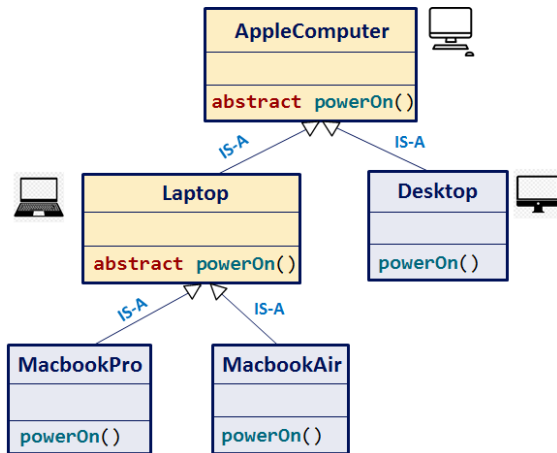


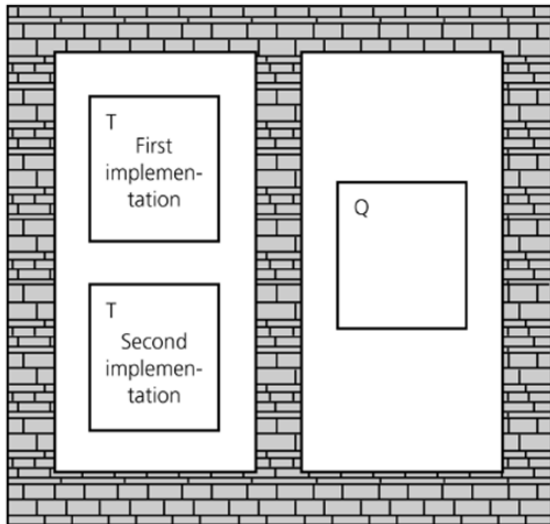
- Suppose that our program uses many kinds of computers, such as **MacbookPro**, **MacbookAir**, and so on. The method **powerOn()** has been written in all of the classes and has functionality to start computer.
- Implementation of the method **powerOn()** in the **AppleComputer** is NOT possible, as the actual computer is not yet known. It has been tagged as **abstract**. Implementation of these abstract method will be provided later once the actual computer is known. These abstract methods cannot be invoked because they have no implementation.
- A class containing one or more abstract methods is called an abstract class. An abstract class CANNOT be instantiated, as its definition is not complete.



The Abstraction hides detailed information on how to start the computer.

- If we required we could also tag the **powerOn()** method as abstract in a derived class, for example we could also have tagged the **powerOn()** as abstract in the **Laptop** class.
- This would mean that you could not create an object of the **Laptop** class.
- It would pass on responsibility for implementing the **powerOn()** method to its children.





- **Information hiding** is like walls building around the various classes of a program.
- The wall around each class **T** prevents the other classes from seeing how **T** works.
- Thus, if class **Q** uses (depends on) **T**, and if the approach for performing **T** changes, class **Q** will not be affected.
- **Makes it easy to substitute new, improved versions of how to do a task later.**

■ Abstraction can be achieved in two different way:

- ▶ Using abstract classes
- ▶ Using interfaces

Interface

I only know method names that I will require for my job to be done.
You have to provide body for those methods.

Sure, I will definitely provide body to all your methods but in my way.

Interface

Implementer

contract



Abstract class

Some methods I know.

Some methods I don't know and I will depend upon you to provide.

Implementer 2



abstract class

Some methods I know.

Some methods I don't know and I will depend upon you to provide.

Implementer 1



1 Abstraction

■ Abstract Classes

- Interfaces
- Usage of abstract classes
- Usage of interfaces

2 Abstract Data Types

3 Modelling Tools and Languages

4 References

- An **abstract method** is a method with only signature (i.e., the method name, the list of arguments and the return type) **without implementation** (i.e., the method's body). You use the keyword **abstract** to declare an abstract method.
- These abstract methods cannot be invoked because they have no implementation.
- Example



```
1 public abstract class Shape {  
    .....  
3    .....  
  
5    public abstract double getArea();  
    public abstract double getPerimeter();  
7    public abstract void draw();  
}
```

- Implementation of these methods is NOT possible in the **Shape** class, as the actual shape is not yet known. (How to compute the area if the shape is not known?) Implementation of these abstract methods will be provided later once the actual shape is known.

- A class containing one or more abstract methods is called an **abstract class**. An **abstract class** must be declared with a class-modifier **abstract**.
- An **abstract class** is **incomplete** in its definition, since the implementation of its abstract methods is missing. Therefore, an abstract class **cannot be instantiated**. In other words, you cannot create instances from an abstract class (otherwise, you will have an incomplete instance with missing method's body).
- Example



```
public abstract class Shape {  
2    .....  
    .....  
4    public abstract double getArea();  
    public abstract double getPerimeter();  
6    public abstract void draw();  
}
```

- You can create instances of the subclasses such as **Triangle** and **Rectangle**, and upcast them to **Shape** (so as to program and operate at the interface level), but you cannot create instance of **Shape**.



```
1  /**
   * This abstract superclass Shape contains an abstract method
   *   getArea(), to be implemented by its subclasses.
   */
5  public abstract class Shape {
    private String color;
7
    public Shape (String color) {
9      this.color = color;
    }
11
    @Override
13    public String toString() {
        return "Shape[color=" + color + "]";
15    }

17    // All Shape's concrete subclasses must implement a method called getArea()
    public abstract double getArea();
19 }
```

- To use an abstract class, you have to derive a subclass from the abstract class. In the derived subclass, you have to override the abstract methods and provide implementation to all the abstract methods. The subclass derived is now complete, and can be instantiated. (If a subclass does not provide implementation to all the abstract methods of the superclass, the subclass remains abstract.)



```
1 public class Rectangle extends Shape {  
    private int length;  
3    private int width;  
  
5    public Rectangle (...) { ... }  
  
7    @Override  
    public double getArea() {  
9        return length * width;  
    }  
11 }
```

- For example, in the abstract class **Shape**, you can define abstract methods such as **getArea()**. No implementation is possible because the actual shape is not known. However, by specifying the signature of the abstract methods, all the subclasses are forced to use these methods' signature. The subclasses **Rectangle** could provide the proper implementations.

- In summary, an abstract class provides a **template for further development**. The purpose of an abstract class is to provide a common interface (or protocol, or contract, or understanding, or naming convention) to all its subclasses.
- Coupled with polymorphism, you can upcast subclass instances to **Shape**, and program at the **Shape** level, i.e., program at the interface. The separation of interface and implementation enables better software design, and ease in expansion.
 - ▶ For example, **Shape** defines a method called **getArea()**, which all the subclasses must provide the correct implementation. You can ask for a **getArea()** from any subclasses of **Shape**, the correct area will be computed.
 - ▶ Furthermore, your application can be extended easily to accommodate new shapes (such as **Circle** or **Square**) by deriving more subclasses.
- **Notes**
 - ▶ An abstract method cannot be declared final, as final method cannot be overridden. An abstract method, on the other hand, must be overridden in a descendant before it can be used.
 - ▶ An abstract method cannot be private (which generates a compilation error). This is because private method are not visible to the subclass and thus cannot be overridden.

- **Rule of Thumb:** Program at the interface, not at the implementation. (That is, make references at the superclass; substitute with subclass instances; and invoke methods defined in the superclass only.)



```
1 public class TestShape {  
    public static void main(String[] args) {  
3         Shape s1 = new Rectangle("red", 4, 5);  
        System.out.println(s1);  
5         System.out.println("Area is " + s1.getArea());  
  
7         Shape s2 = new Triangle("blue", 4, 5);  
        System.out.println(s2);  
9         System.out.println("Area is " + s2.getArea());  
  
11        // Cannot create instance of an abstract class  
        Shape s3 = new Shape("green");  
13        // Compilation error: Shape is abstract; cannot be instantiated.  
    }  
15 }
```

1 Abstraction

- Abstract Classes

- **Interfaces**

- Usage of abstract classes

- Usage of interfaces

2 Abstract Data Types

3 Modelling Tools and Languages

4 References

- A Java **interface** is a **100% abstract superclass** which define a set of methods its subclasses must support.
- An interface contains only public abstract methods (methods with signature and no implementation) and possibly constants (public static final variables).
- You have to use the keyword "interface" to define an interface (instead of keyword "class" for normal classes).
- The keyword public and abstract are not needed for its abstract methods as they are mandatory.
- Unlike a normal class, where you use the keyword "extends" to derive a subclass. For interface, we use the keyword "implements" to derive a subclass.
- **Interface Naming Convention:** Use an adjective (typically ends with "able") consisting of one or more words. Each word shall be initial capitalized (camel-case).
 - ▶ For example, **Serializable**, **Extensible**, **Movable**, **Cloneable**, **Runnable**, etc.

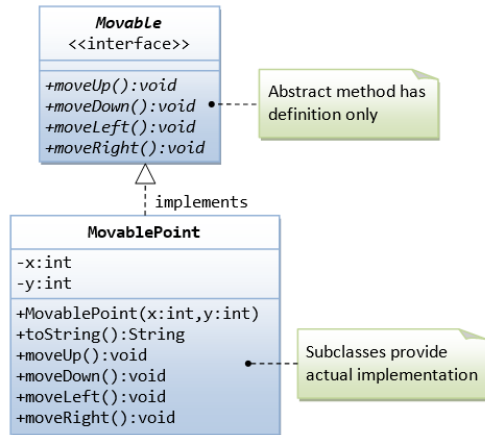


```
1  /**
   * The Movable interface defines a list of public abstract methods
3  *   to be implemented by its subclasses
   */
5  public interface Movable { // Use keyword "interface" (instead of "class") to
                              //   define an interface
7      // An interface defines a list of public abstract methods to be implemented
      //   by the subclasses
9      [public abstract] void moveUp(); // "public" and "abstract" optional
      void moveDown();
11     void moveLeft();
      void moveRight();
13 }
```



```

1  /**
2   * The subclass MovablePoint needs to
3   * implement all the abstract methods
4   * defined in the interface Movable.
5   */
6   // Using keyword implements instead of extends
7   public class MovablePoint implements Movable {
8       private int x;
9       private int y;
10
11      public MovablePoint(int x, int y) { ... }
12
13      @Override
14      public void moveUp() {
15          y--;
16      }
17
18      @Override
19      public void moveDown() {
20          y++;
21      }
22      ...
23  }
    
```





```

1 public class TestMovable {
2     public static void main(String[] args) {
3         MovablePoint p1 = new MovablePoint(1, 2);
4         System.out.println(p1);    // (1,2)
5         p1.moveDown();
6         System.out.println(p1);    // (1,3)
7         p1.moveRight();
8         System.out.println(p1);    // (2,3)
9
10        // Usage of interface.
11        Movable p2 = new MovablePoint(3, 4);    // Upcast. Interface as a datatype.
12        p2.moveUp();                            // Test polymorphism.
13        System.out.println(p2);                // (3,3)
14
15        MovablePoint p3 = (MovablePoint)p2;    // Downcast
16        System.out.println(p3);                // (3,3)
17    }
18 }
    
```

- An interface is a **contract** for what the classes can do. It, however, does not specify how the classes should do it.
- An interface provides a **form**, a **protocol**, a **standard**, a **contract**, a **specification**, a **set of rules**, an **interface**, for all objects that implement it. It is a specification and rules that any object implementing it agrees to follow.
- Similar to an abstract superclass, an interface cannot be instantiated. You have to create a "subclass" that implements an interface, and provide the actual implementation of all the abstract methods.
- In Java, abstract class and interface are used to **separate the public interface of a class from its implementation** so as to allow the programmer to **program at the interface instead of the various implementation**.

- One of the main usage of interface is provide a communication contract between two objects. If you know a class implements an interface, then you know that class contains concrete implementations of the methods declared in that interface, and you are guaranteed to be able to invoke these methods safely. In other words, two objects can communicate based on the contract defined in the interface, instead of their specific implementation.
- Secondly, Java does not support multiple inheritance (whereas C++ does). Multiple inheritance permits you to derive a subclass from more than one direct superclass. This poses a problem if two direct superclasses have conflicting implementations. (Which one to follow in the subclass?). However, multiple inheritance does have its place. Java does this by permitting you to "implements" more than one interfaces (but you can only "extends" from a single superclass). Since interfaces contain only abstract methods without actual implementation, no conflict can arise among the multiple interfaces. (Interface can hold constants but is not recommended. If a subclass implements two interfaces with conflicting constants, the compiler will flag out a compilation error.)

- **An abstract class is a class that is incomplete in the sense it has missing method bodies.** In that it describes a set of operations, but is missing the actual implementation of these operations.
- A class is like a set of plans from which you can create objects. In relation to this analogy, an abstract class is like a set of plans with some part of the plans missing. E.g. it could be a car with no engine - you would not be able to make complete car objects without the missing parts of the plan.

- Any class containing one or more abstract methods is an **abstract class**.
- You **must** declare the class with the keyword **abstract**:



```
abstract class MyClass { ... }
```

- You **cannot instantiate** (create a new instance of) an abstract class.
 - ▶ So, can only be used through inheritance.
- You **can extend (subclass) an abstract class**.
 - ▶ If the subclass defines all the inherited abstract methods, it is concrete and can be instantiated.
 - ▶ If the subclass does not define all the inherited abstract methods, it must be abstract too.
- You can declare a class to be abstract even if it does not contain any abstract methods.
 - ▶ This just prevents the class from being instantiated.

1 Abstraction

- Abstract Classes
- Interfaces
- Usage of abstract classes
- Usage of interfaces

2 Abstract Data Types

3 Modelling Tools and Languages

4 References

- Suppose you wanted to create a class **Shape**, with subclasses **Oval**, **Rectangle**, **Triangle**, **Hexagon**, etc.
- Each subclass implements a method **draw()** for representing its shape on a 2D graphic panel.



```
class Shape { ... }  
2  
class Star extends Shape {  
4   void draw() { ... }  
   ...  
6 }  
  
8 class Circle extends Shape {  
   void draw() { ... }  
10  ...  
   }
```



```
class Shape { ... }  
2  
class Star extends Shape {  
4   void draw() { ... }  
   ...  
6 }  
  
8 class Circle extends Shape {  
   void draw() { ... }  
10  ...  
   }
```



```
// Legal, but unwanted  
2 Shape shape = new Shape();  
   // Illegal, Shape does not have draw()  
4 shape.draw();  
   // Legal, because a Star is a Shape  
6 shape = new Star();  
   // Illegal, Shape does not have draw()  
8 shape.draw();  
  
10 // Same problem, another view  
   Shape[] shapes = new Shape[16];  
12 shapes[0] = new Circle();  
   shapes[1] = new Star();  
14 ...  
  
16 for (Shape s : shapes) {  
   // Illegal, Shape doesn't have draw()  
18   s.draw();  
   }
```



```
class Shape {  
2   void draw() { ... }  
}  
4  
class Star extends Shape {  
6   void draw() { ... }  
   ...  
8 }  
10 class Circle extends Shape {  
    void draw() { ... }  
12   ...  
   }  
14  
Shape shape;  
16 shape = new Shape(); // Legal, but unwanted, because Shape does not need to exist in reality.  
   shape.draw();       // Legal, but unwanted, because Shape has no specific shape.  
18 shape = new Star();  // Legal, because a Star is a Shape.  
   shape.draw();       // Legal, Shape does have draw().
```



```
1  abstract class Shape {  
    abstract void draw();  
3  }  
  
5  class Star extends Shape {  
    void draw() { ... }  
7    ...  
    }  
9  
11 class Circle extends Shape {  
    void draw() { ... }  
    ...  
13 }  
  
15 Shape shape;  
    shape = new Shape();           // Illegal, Shape is abstract.  
17 shape = new Star();             // Legal, because a Star is a Shape.  
    shape.draw();                  // Legal, Shape does have draw().
```


- Let's suppose that all shapes must have two capabilities:
 - ▶ Drawing their own shape (**draw()** method).
 - ▶ Storing an unique Id (**setId()** method).
- We can keep the **draw()** method abstract while providing an implementation of the **setId()** method. In this way, **Shape** cannot be instantiated and **Shape** subclasses can implement **draw()**.
- Benefit: repeated code minimized!
- Drawback: completely abstract concepts are missing!



```
1  abstract class Shape {  
2      void setId() { ... };  
3      abstract void draw();  
4  }  
  
6  class Star extends Shape {  
7      void draw() { ... }  
8      ...  
9  }  
10  
12 class Circle extends Shape {  
13     void draw() { ... }  
14     ...  
15 }
```

1 Abstraction

- Abstract Classes
- Interfaces
- Usage of abstract classes
- Usage of interfaces

2 Abstract Data Types

3 Modelling Tools and Languages

4 References

- Completely abstract concepts can be defined by making use of **Interfaces**.
- Interfaces are particular classes containing exclusively abstract methods. All their methods are **implicitly public and abstract**. Thus, Interfaces cannot be instantiated.
- When a class implements an interface, it promises to the compiler to provide an **implementation for all the methods declared within the interface**.
- Benefit: completely abstract concepts!
- Drawback: repeated code possible!



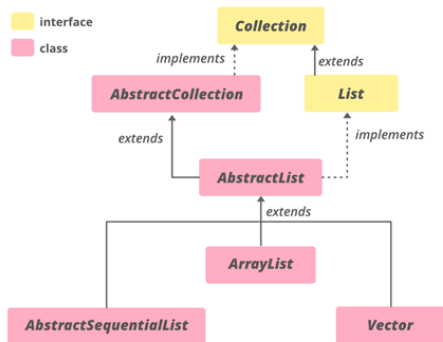
```
1 interface Shape {  
    public abstract void setId();  
3    public abstract void draw();  
    }  
5  
6    class Star implements Shape {  
7        void setId() { ... }  
8        void draw() { ... }  
9        ...  
10    }  
11  
12    class Circle implements Shape {  
13        void setId() { ... }  
14        void draw() { ... }  
15        ...  
16    }
```

- Interfaces can be specialized.
- Specializing an interface means adding new methods in derived interfaces. Overriding methods does not make sense in interfaces because code is absent.



```
1 interface Shape {  
    public abstract void setId();  
3    public abstract void draw();  
    }  
5  
    interface MovingShape extends Shape {  
7        public abstract void move(Point p);  
    }
```

- Interfaces can be partially implemented in abstract classes. The approach provides both abstract concepts and reduced code repetition.
- Widely used inside the Java API.



```

1 interface Shape {
2     public abstract void setId();
3     public abstract void draw();
4 }

6 abstract class AbstractShape implements Shape {
7     void setId() { ... };
8     abstract void draw();
9 }

10
12 class Star extends AbstractShape {
13     void draw() { ... }
14 }

16 class Circle extends AbstractShape {
17     void draw() { ... }
18 }

```

- In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to an interface, **all the implementations will be forced to implement them too.**
- Default interface methods are an efficient way to deal with this issue.
- They allow us to add new methods to an interface that are automatically available in the implementations. Therefore, we don't need to modify the implementing classes.



```
1 public interface Shape {  
    void setId();  
3    void draw();  
  
5    default String getDescription() {  
        return "This is a Shape.";  
7    }  
}  
9  
10 public class Circle implements Shape {  
11    void setId() { ... }  
    void draw() { ... }  
13    ...  
14 }  
15  
17 Circle circle = new Circle();  
    circle.getDescription();
```

- In addition to declaring default methods in interfaces, Java also allows us to define and implement **static methods** in interfaces.
- Since static methods don't belong to a particular object, they're not part of the API of the classes implementing the interface, therefore, **they have to be called by using the interface name preceding the method name.**



```
interface Shape {  
2   void setID ();  
   void draw ();  
  
4  
   public static String getDescription() {  
6       return "This is a Shape";  
   }  
8 }  
  
10 class Circle implements Shape {  
   void setID () { ... }  
12   void draw () { ... }  
   ...  
14 }  
  
16  
   Shape . getDescription () ;
```

- In Java, a class can only extend one class, but can **implement multiple interfaces**. This lets the class fill multiple roles (i.e., multiple set of methods).
 - ▶ Collections (e.g., **LinkedList**) commonly implement multiple interfaces.
 - ▶ Graphical containers (e.g., **JFrame**) commonly implement several listeners (i.e., interfaces).



```
1 class LinkedList extends AbstractList implements List , Queue {  
2     ...  
3 }  
4  
5 class Application extends JFrame implements ActionListener , KeyListener {  
6     ...  
7 }
```




```
1 public class GroudVehicle {  
    public activateWheels () {  
3        ...  
    }  
5 }  
  
7 public class WaterVehicle {  
    public activateWaterFans () {  
9        ...  
    }  
11 }  
  
13 // Not allowed in Java!! Only one class can be extended!  
15 public class Amphibian extends GroudVehicle , WaterVehicle {  
    ...  
17 }
```



```
1 public interface GroudVehicle {  
    activateWheels ();  
3 }  
  
5 public interface WaterVehicle {  
    activateWaterFans ();  
7 }  
  
9  
11 public class Amphibian implements GroudVehicle , WaterVehicle {  
    activateWheels () {  
        ...  
13    }  
  
15    activateWaterFans () {  
        ...  
17    }  
}
```

- **instanceof** is an operator telling whether an object "IS-A" member of a class of objects.
- Membership practically means **having its methods implemented**.



```
class Dog extends Mammal implements Pet, Friend, Fun {  
2    ...  
}  
4  
Dog lessie = new Dog();  
6 lessie instanceof Object    // OK!  
  lessie instanceof Dog      // OK!  
8 lessie instanceof Mammal    // OK!  
  lessie instanceof Pet       // OK!  
10 lessie instanceof Friend    // OK!
```

- 1 Abstraction
- 2 Abstract Data Types**
 - Complex Number
 - List
- 3 Modelling Tools and Languages
- 4 References

■ Abstraction

- ▶ Concentrate on what it **CAN DO** and **NOT** how it does it.
- ▶ E.g. usage of **Abstract Class** and **Interface**.

■ Coupling

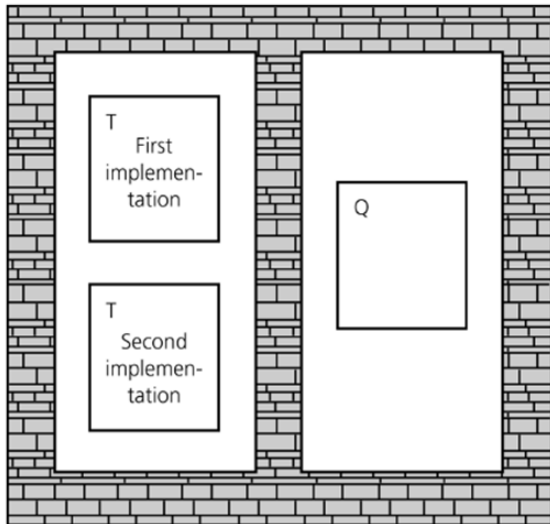
- ▶ Restrict interdependent relationship among classes to the minimum.

■ Cohesion

- ▶ A class should be about a **single entity** only.
- ▶ There should be a clear logical grouping of all functionalities.

■ Information Hiding

- ▶ Expose only necessary information to outside.



- **Information hiding** is like walls building around the various classes of a program.
- The wall around each class **T** prevents the other classes from seeing how **T** works.
- Thus, if class **Q** uses (depends on) **T**, and if the approach for performing **T** changes, class **Q** will not be affected.
- **Makes it easy to substitute new, improved versions of how to do a task later.**

- Information Hiding is not complete isolation of the classes.
- Information released is on a need-to-know basis.
- Class **Q** does not know how class **T** does the work, but it needs to know how to invoke **T** and what **T** produces.
 - E.g.: The designers of the methods of **Math** and **Scanner** classes have hidden the details of the implementations of the methods from you, but provide enough information (the method headers and explanation) to allow you to use their methods.
- What goes in and comes out is governed by the terms of the method's specifications.
 - If you use this method in this way, this is exactly what it will do for you (pre- and post-conditions).

■ Pre-conditions

- ▶ Conditions that must be true before a method is called.
- ▶ "This is what I expect from you".
- ▶ The programmer is responsible for making sure that the pre-conditions are satisfied when calling the method.

■ Post-conditions

- ▶ Conditions that must be true after the method is completed.
- ▶ "This is what I promise to do for you".



```
// Pre-cond: x >= 0
2 // Post-cond: Return the square root of x
  public static double squareRoot(double x) {
4     . . .
  }
```


- **Information Hiding** can also apply to data.
 - ▶ **Data abstraction** asks that you think in terms of **what you can do** to a collection of data independently of **how you do it**.
 - ▶ **Data structure** is a construct that can be defined within a programming language to store a collection of data.
 - ▶ **Abstract Data Type** (ADT) is a collection of data and a specification on the set of operations/methods on that data.
 - Typical operations on data are: add, remove, and query (in general, management of data).
 - Specification indicates what ADT operations do, but not how to implement them.

Collection of data + set of operations on the data

- **Data structure** is a construct that can be defined within a programming language to store a collection of data.
 - ▶ Arrays, which are built into Java, are data structures.
 - ▶ We can create other data structures. For example, we want a data structure (a collection of data) to store both the names and salaries of a collection of employees.



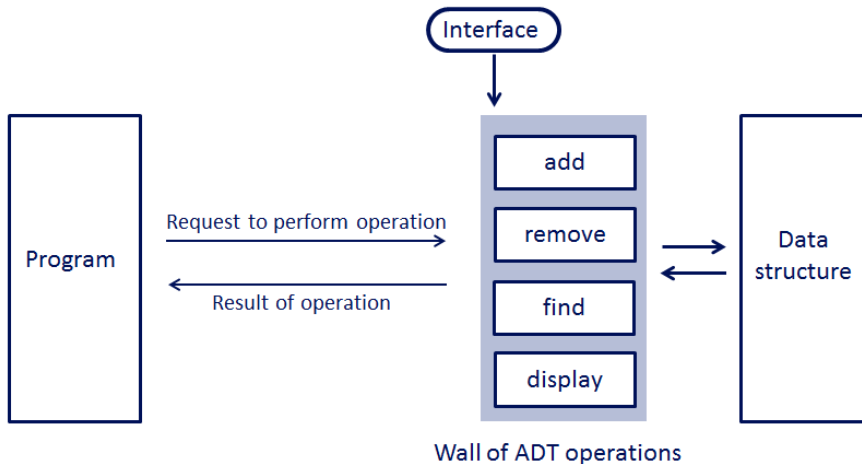
```
1 static final int MAX_NUMBER = 500; // defining a constant
   String[] names = new String[MAX_NUMBER];
3 double[] salaries = new double[MAX_NUMBER];
   // employee names[i] has a salary of salaries[i]
5
   // Or (better choice)
7
   class Employee {
9       static final int MAX_NUMBER = 500;
       private String names;
11      private double salaries;
   }
13 ...
   Employee[] workers = new Employee[Employee.MAX_NUMBER];
```

- An ADT is a collection of data together with a specification of a set of operations on the data.
 - ▶ Specifications indicate what ADT operations do, NOT how to implement them.
 - ▶ Data structures are part of an ADT's implementation.

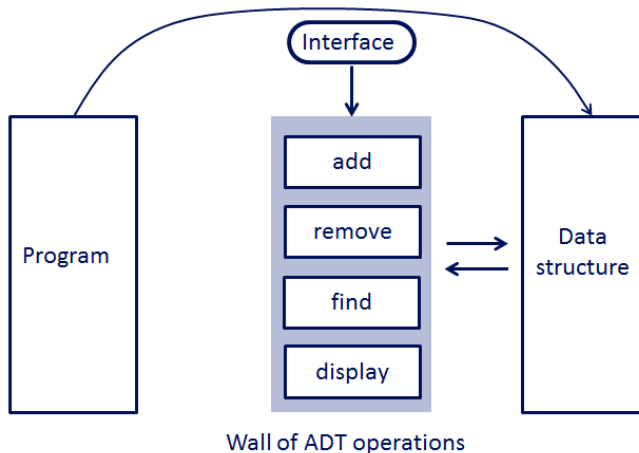


- When a program needs data operations that are not directly supported by a language, you need to create your own ADT.
- You should first design the ADT by carefully specifying the operations before implementation.

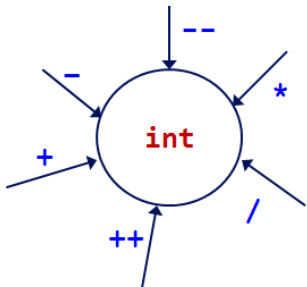
- A **WALL** of **ADT** operations isolates a data structure from the program that uses it.
- An **Interface** is what a program/module/class should understand on using the ADT.



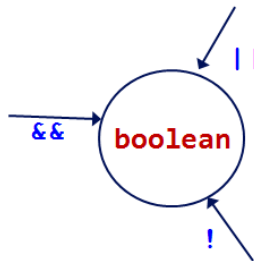
- An interface is what a program/module/class should understand on using the ADT.
- The following bypasses the interface to access the data structure. This violates the wall of ADT operations.



- Java's predefined data types are ADTs.
- Representation details are hidden which aids portability as well.



`int` type with the operations
(e.g.: `--`, `/`) defined on it.



`boolean` type with the operations
(e.g.: `&&`) defined on it.

■ Constructors (to add, create data)



```
int[] z = new int[4];  
2 int[] x = {2,4,6,8};
```

■ Mutators (to modify data)



```
x[3] = 10;
```

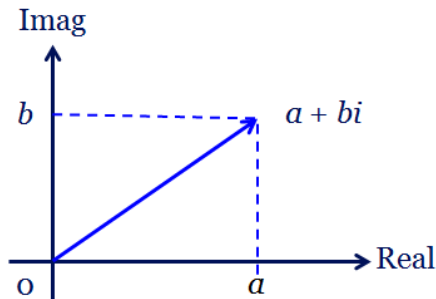
■ Accessors (to query about state/value of data)



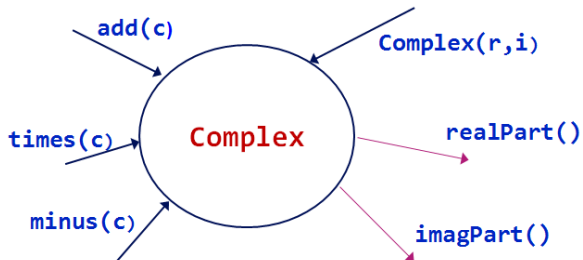
```
1 int y = x[3] + x[2];
```

- 1 Abstraction
- 2 Abstract Data Types
 - Complex Number
 - List
- 3 Modelling Tools and Languages
- 4 References

- A complex number comprises a real part a and an imaginary part b , and is written as $a + bi$.
- i is a value such that $i^2 = -1$.
- Examples: $12 + 3i$, $15 - 9i$, $-5 + 4i$, -23 , $18i$.
- A complex number can be visually represented as a pair of numbers (a, b) representing a vector on the two-dimensional complex plane (horizontal axis for real part, vertical axis for imaginary part).



■ Interface



■ Implementation: two different ways



```
1 class CartesianComplex {  
    private double real;  
    private double imag;  
}
```



```
1 class PolarComplex {  
2     private double ang;  
    private double mag;  
4 }
```

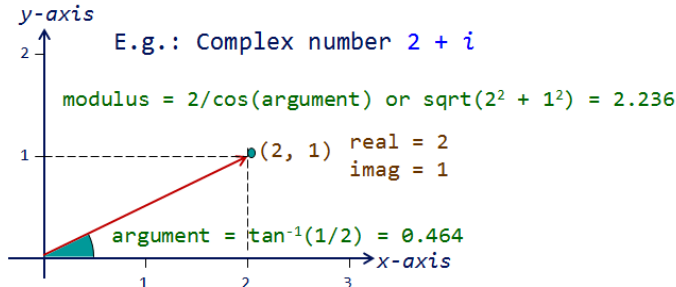
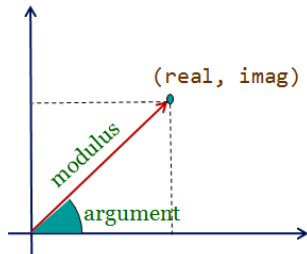
Relationship between Cartesian and Polar representations

From Polar to Cartesian:

$$\begin{aligned} \text{real} &= \text{modulus} * \cos(\text{argument}); \\ \text{imag} &= \text{modulus} * \sin(\text{argument}); \end{aligned}$$

From Polar to Cartesian:

$$\begin{aligned} \text{argument} &= \tan^{-1}(\text{imag}/\text{real}); \\ \text{modulus} &= \text{real} / \cos(\text{argument}); \\ \text{or } \text{modulus} &= \sqrt{\text{real}^2 + \text{imag}^2}; \end{aligned}$$





```
class CartesianComplex {  
2   private double real;  
   private double imag;  
4  
   public Complex(double real , double imag) {  
6       this.real = real;  
       this.imag = imag;  
8   }  
  
10  public double realPart () {  
    return real;  
12  }  
  
14  public double imagPart () {  
    return imag;  
16  }
```



```
2 public double modulus() {  
    return Math.sqrt(real * real + imag * imag);  
}  
4  
6 public double argument() {  
    if (real != 0) {  
        if (real < 0) {  
8            return (Math.PI + Math.atan(imag / real));  
        } else {  
10            return Math.atan(imag / real);  
        }  
12    } else if (imag == 0) {  
        return 0;  
14    } else if (imag > 0) {  
        return Math.PI / 2;  
16    } else {  
        return -Math.PI / 2;  
18    }  
}
```



```
1  public void add(CartesianComplex c) {  
    real += c.realPart();  
3    imag += c.imagPart();  
    }  
  
5  
6  public void minus(CartesianComplex c) {  
7    real -= c.realPart();  
    imag -= c.imagPart();  
9    }  
  
11 public void times(CartesianComplex c) {  
    real = real * c.realPart() - imag * c.imagPart();  
13    imag = real * c.imagPart() + imag * c.realPart();  
    }  
15  
16 public String toString() {  
17     if (imag == 0) {return (real + "");}  
    if (imag < 0) return (real + "-" + imag + "i");  
19     return (real + "+" + imag + "i");  
    }  
21 }
```



```
1 class PolarComplex {  
    private double modulus;  
3    private double argument;  
  
5    public Complex(double modulus, double argument) {  
        this.modulus = modulus;  
7        this.argument = argument;  
    }  
  
9  
    public double realPart() {  
11        return modulus * Math.cos(argument);  
    }  
  
13  
    public double imagPart() {  
15        return modulus * Math.sin(argument);  
    }  
}
```



```
public void add(PolarComplex c) {
2   double real = this.realPart() + c.realPart();
   double imag = this.imagPart() + c.imagPart();
4
   modulus = Math.sqrt(real*real + imag*imag);
6   if (real != 0) {
       if (real < 0) {
8           argument = (Math.PI + Math.atan(imag/real));
       } else {
10          argument = Math.atan(imag/real);
       }
12  } else if (imag == 0) {
       argument = 0;
14  } else if (imag > 0) {
       argument = Math.PI/2;
16  } else {
       argument = -Math.PI/2;
18  }
}
```




```
1 public void minus(PolarComplex c) {  
    double real = mag * Math.cos(ang) - c.realPart();  
3    double imag = mag * Math.sin(ang) - c.imagPart();  
  
5    modulus = Math.sqrt(real * real + imag * imag);  
    if (real != 0) {  
6        if (real < 0) {  
            argument = (Math.PI + Math.atan(imag / real));  
9        } else {  
            argument = Math.atan(imag / real);  
11        }  
    } else if (imag == 0) {  
13        argument = 0;  
    } else if (imag > 0) {  
15        argument = Math.PI / 2;  
    } else {  
17        argument = -Math.PI / 2;  
    }  
19 }
```



```
1  public double modulus() {  
    return this.modulus;  
3  }  
  
5  public double argument() {  
    return this.argument;  
7  }  
  
9  public void times(PolarComplex c) {  
    modulus *= c.modulus();  
11   argument += c.argument();  
    }  
13  
    public String toString() {  
15     if (imagPart() == 0) return (realPart() + "");  
        if (imagPart() < 0) return (realPart() + "-" + imagPart() + "i");  
17     return (realPart() + "+" + imagPart() + "i");  
    }  
19 }
```

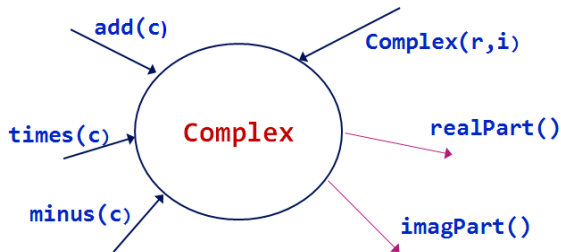


```
1 // Testing CartesianComplex
  CartesianComplex a = new CartesianComplex(10.0, 12.0);
3 CartesianComplex b = new CartesianComplex(1.0, 2.0);
  a.add(b);
5 System.out.println("a = a + b is " + a);
  a.minus(b);
7 System.out.println("a - b (which is the original a) is " + a);
  System.out.println("Angle of a is " + a.angle());
9 a.times(b);
  System.out.println("a = a * b is " + a);
11
  // Testing PolarComplex
13 PolarComplex c = new PolarComplex(10.0, Math.PI/6.0);
  PolarComplex d = new PolarComplex(1.0, Math.PI/3.0);
15 System.out.println("c is " + c);
  System.out.println("d is " + d);
17 c.add(d);
  System.out.println("c = c + d is " + c);
19 c.minus(d);
  System.out.println("c - d (which is the original c) is " + c);
21 c.times(d);
  System.out.println("c = c * d is " + c);
```

- User-defined data types can also be organized as ADTs.
- Let's create a "Complex" ADT for complex numbers.



```
public interface Complex {  
2   public double realPart();  
   public double imagPart();  
4   public double modulus();  
   public double argument();  
6   public void add(Complex c);  
   public void minus(Complex c);  
8   public void times(Complex c);  
}
```





```
class CartesianComplex {  
2    ...  
4    public void add(Complex c) { ... }  
6    public void minus(Complex c) { ... }  
    public void times(Complex c) { ... }  
8 }  
  
10 class PolarComplex {  
12    ...  
14    public void add(Complex c) { ... }  
    public void minus(Complex c) { ... }  
16    public void times(Complex c) { ... }  
}
```



```
1 Complex a = new CartesianComplex(10.0 , 12.0);  
   Complex b = new CartesianComplex(1.0 , 2.0);  
3  
   Complex c = new PolarComplex(10.0 , Math.PI / 6.0);  
5 Complex d = new PolarComplex(1.0 , Math.PI / 3.0);  
  
7 System.out.println("Testing Combined:");  
   System.out.println("a is " + a);  
9 System.out.println("d is " + d);  
   a.minus(d);  
11 System.out.println("a = a - d is " + a);  
   a.times(d);  
13 System.out.println("a = a*d is " + a);  
   d.add(a);  
15 System.out.println("d = d + a is " + d);  
   d.times(a);  
17 System.out.println("d = d*a is " + d);
```

- 1 Abstraction
- 2 Abstract Data Types
 - Complex Number
 - List
- 3 Modelling Tools and Languages
- 4 References

- List is one of the most basic types of data collection.
 - ▶ In general, we keep items of the same type (class) in one list.
- Typical operations on a data collection:
 - ▶ Add data
 - ▶ Remove data
 - ▶ Query data
 - ▶ The details of the operations vary from application to application. The overall theme is the management of data.



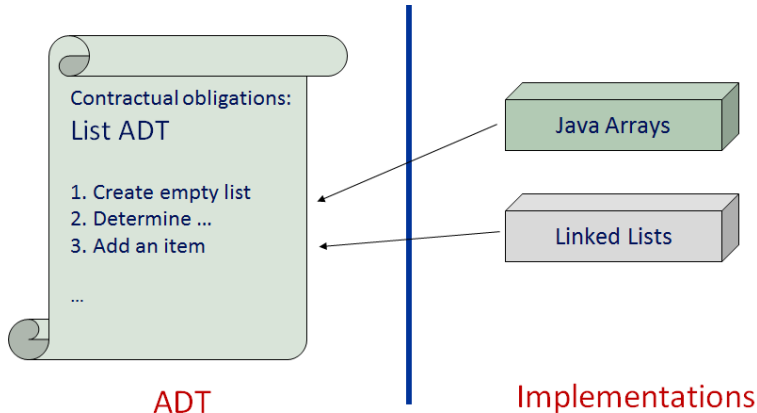
- A List ADT is a dynamic linear data structure.
 - ▶ A collection of data items, accessible one after another starting from the beginning (head) of the list.
- Examples of List ADT operations:
 - ▶ Create an empty list
 - ▶ Determine whether a list is empty
 - ▶ Determine number of items in the list
 - ▶ Insert an item at a given position
 - ▶ Remove an item at a position
 - ▶ Remove all items
 - ▶ Read an item from the list at a position



```
1 public interface MyList {  
    public boolean isEmpty();  
3    public int size();  
    public Object getFirst();  
5    public Object getAt(int index)  
    public boolean contains(Object item);  
7    public void insertFirst(Object item);  
    public void insert(int index, Object item)  
9    public Object removeFirst();  
    public Object remove(int index);  
11   public void print();  
}
```

- The **MyList** above defines the operations (methods) we would like to have in a **List** ADT.
- The operations shown here are just a small sample. An actual **List** ADT usually contains more operations.

- We will consider two implementation of **List** ADT, both using the interface **MyList**.

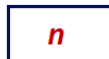


- A fixed-sized List.

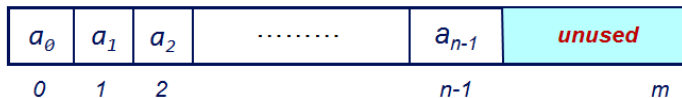


- Use array of a sequence of n elements.

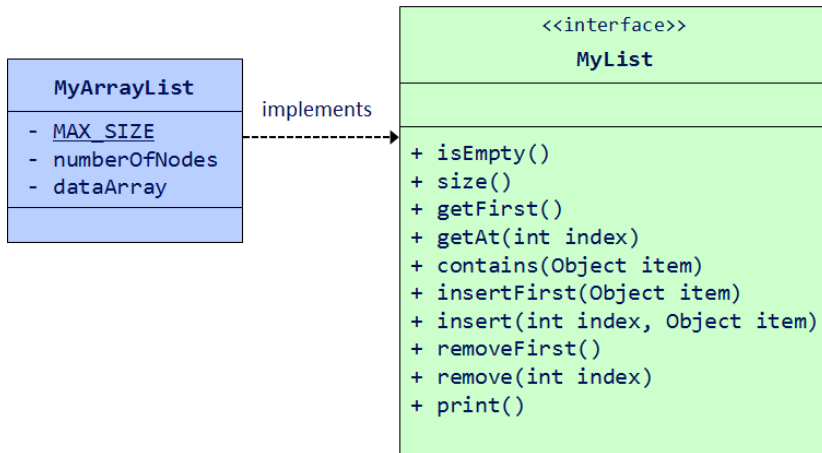
numberOfNodes



arr : array[0..m] of locations

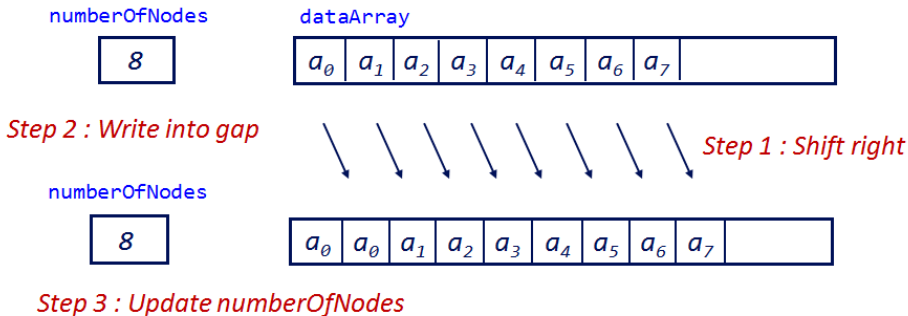


- We now create a class **MyArrayList** as an implementation of the interface **MyList** (a user-defined interface).



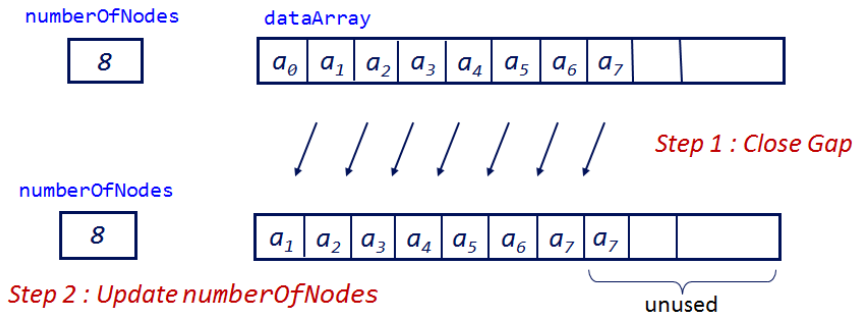
- For insertion into first position, need to shift "right" (starting from the last element) to create room.

Example: `insertFirst("it")`



- For deletion of first element, need to shift “left” (starting from the first element) to close gap.

Example: `removeFirst()`



Need to maintain `numberOfNodes` so that program would not access beyond the valid data.

■ Question: Time Efficiency?

▶ **Retrieval:** `getFirst()`

- Always fast with 1 read operation.

▶ **Insertion:** `insertFirst(Object item)`

- Shifting of all n items – bad!

▶ **Insertion:** `insert(int index, Object item)`

- Inserting into the specified position.
- Best case: No shifting of items (add to the last place).
- Worst case: Shifting of all items (add to the first place).

▶ **Deletion:** `removeFirst(Object item)`

- Shifting of all n items – bad!

▶ **Deletion:** `remove(int index)`

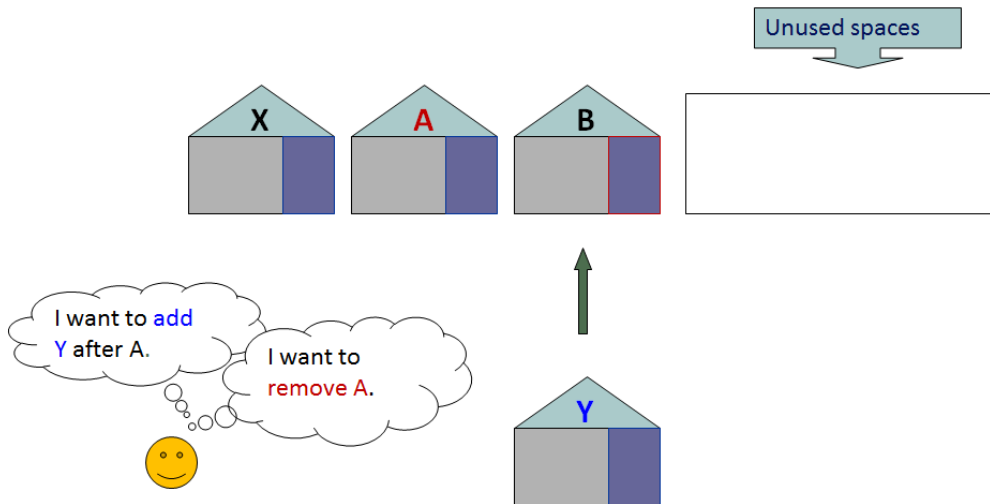
- Delete the item at the specified position.
- Best case: No shifting of items (delete the last item).
- Worst case: Shifting of all items (delete the first item).

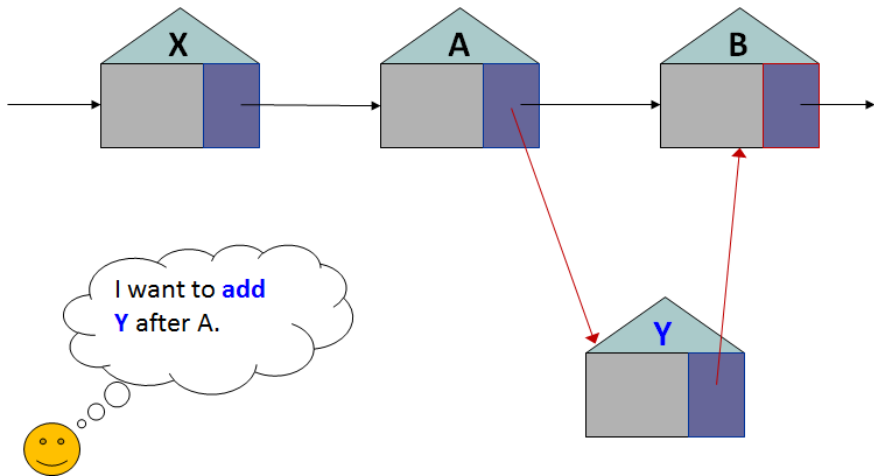
■ Question: What is the Space Efficiency?

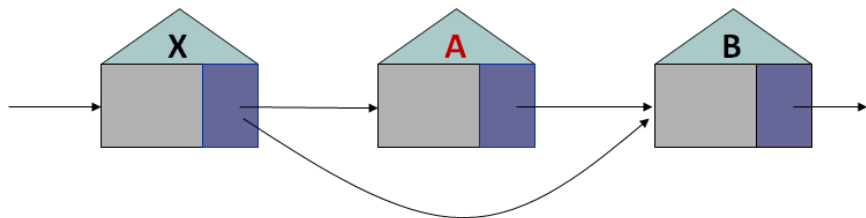
- ▶ Size of array collection limited by MAX_SIZE. Problems
 - We don't always know the maximum size ahead of time.
 - If MAX_SIZE is too liberal, unused space is wasted.
 - If MAX_SIZE is too conservative, easy to run out of space.
- ▶ Idea: make MAX_SIZE a variable, and create/copy to a larger array whenever the array runs out of space.
 - No more limits on size.
 - But copying overhead is still a problem.
- ▶ Insertion: insert(int index, Object item)
 - Inserting into the specified position.
 - Best case: No shifting of items (add to the last place).
 - Worst case: Shifting of all items (add to the first place).
- ▶ When to use such a list?
 - For a fixed-size list, an array is good enough!
 - For a variable-size list, where dynamic operations such as insertion/deletion are common, an array is a poor choice; better alternative – **Linked List**.

Variable-size list





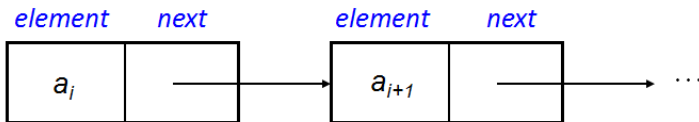




Node **A** becomes a *garbage*. To be removed during garbage collection.

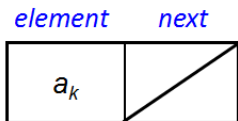
■ Idea

- ▶ Each element in the list is stored in a node, which also contains a next pointer (reference).
- ▶ Allow elements in the list to occupy non-contiguous memory.
- ▶ Order the nodes by associating each with its neighbour(s).



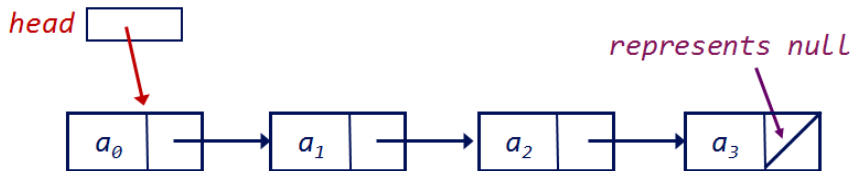
This is one node
of the collection...

... and this one comes after it in the collection
(most likely not occupying contiguous memory
that is next to the previous node).


















Next pointer of this node is "null",
i.e. it has no next neighbour.



















- A sequence of 4 items $\langle a_0, a_1, a_2, a_3 \rangle$.
- We need a **head** to indicate where the first node is.
- From the **head** we can get to the rest.



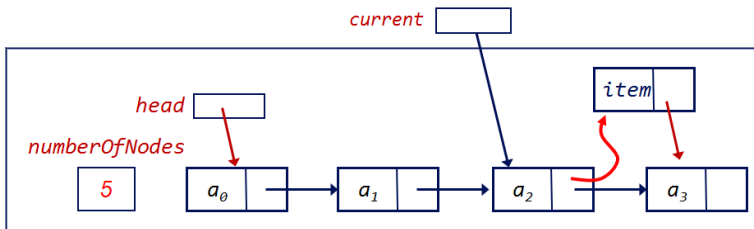
- The insertFirst() method.

Case	Before: list	After: list.insertFirst(99)
0 item	<p>head  numberOfNodes 0</p>	<p>head  numberOfNodes 1</p> <p></p>
1 item	<p>head  numberOfNodes 1</p> <p></p>	<p>head  numberOfNodes 2</p> <p> </p>
2 or more items	<p>head  numberOfNodes n</p> <p> → </p>	<p>head  numberOfNodes n+1</p> <p>  → </p>

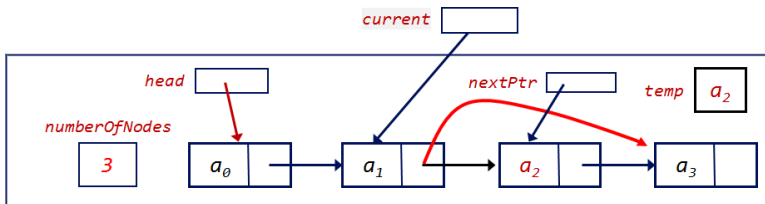
- The `removeFirst()` method.

Case	Before: list	After: list.removeFirst()
0 item	<p>head  numberOfNodes </p>	Can't remove
1 item	<p>head  numberOfNodes </p> <p></p>	<p>head  lead  numberOfNodes </p> <p></p>
2 or more items	<p>head  numberOfNodes </p> <p> → </p>	<p>head  lead  numberOfNodes </p> <p> → </p>

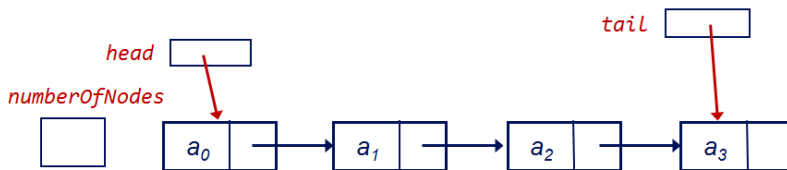
- The insert(int index, Object item) method.



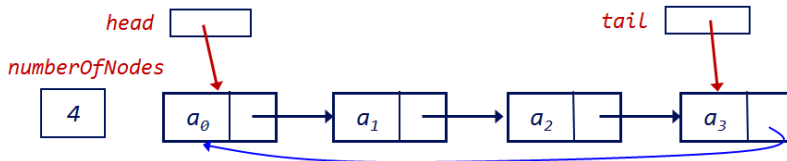
- The removeAt(int index) method.



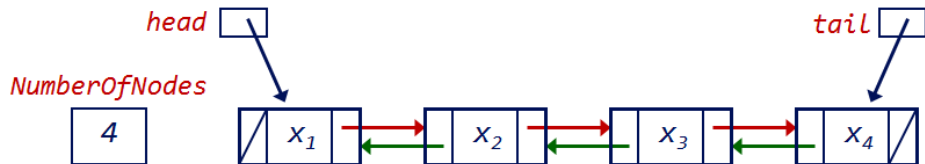
- To address the issue that adding to the end is slow, add an extra data member called tail.



- Allow cycling through the list repeatedly.



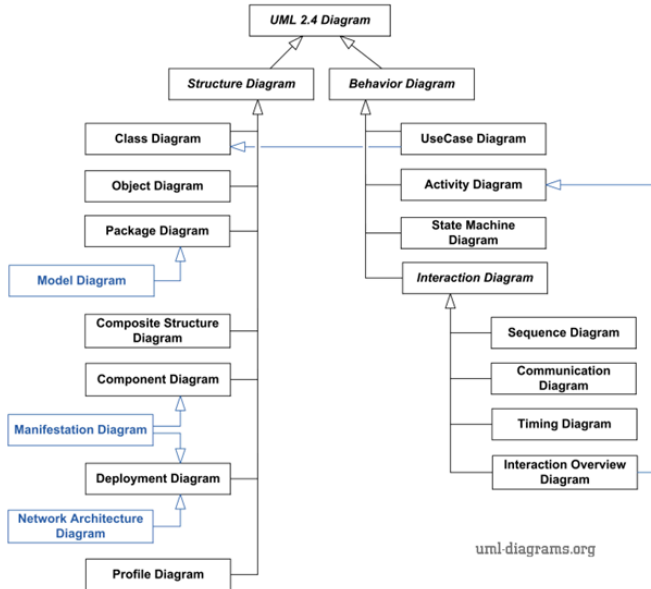
- Often, we need to move backward as well.
- Use a "prev" pointer to allow backward traversal.



- 1 Abstraction
- 2 Abstract Data Types
- 3 Modelling Tools and Languages**
- 4 References

- UML stands for **Unified Modeling Language**.
 - ▶ Language: express idea, not a methodology.
 - ▶ Modeling: Describing a software system at a high level of abstraction.
 - ▶ Unified: UML has become a world standard.
- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems.
- UML uses mostly graphical notations to express the OO analysis and design of software projects.
- **Simplifies the complex process of software design.**

- Use **graphical notation**: more clearly than natural language (imprecise) and code (too detailed).
- Help acquire an **overall view of a system**.
- UML is **not dependent on any one language or technology**.
- UML moves us from fragmentation to **standardization**.



- A class diagram depicts **classes and their interrelationships**.
- Used for describing the **internal structure of a software system**.
- Provide a conceptual model of the system in terms of entities and their relationships.
- Detailed class diagrams are used for developers.



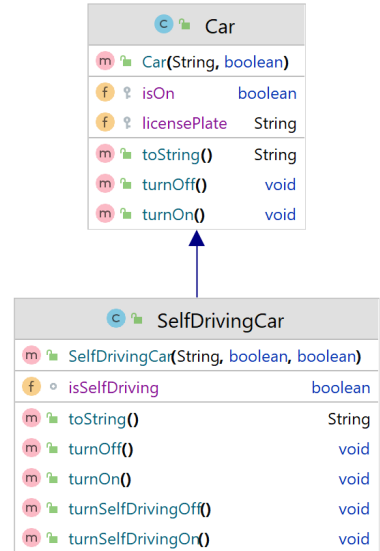
```
public class Car {  
2   protected String licensePlate;  
   protected boolean isOn;  
  
4   public Car(String licensePlate, boolean isOn) {  
6       this.license = licensePlate;  
       this.isOn = isOn;  
8   }  
  
10  void turnOn() { ... }  
  
12  void turnOff() { ... }  
  
14  @Override  
   public String toString() { ... }  
16 }
```

c Car		
m	Car(String, boolean)	
f	isOn	boolean
f	licensePlate	String
m	toString()	String
m	turnOff()	void
m	turnOn()	void



```

1 public class SelfDrivingCar extends Car {
    boolean isSelfDriving;
3
    public SelfDrivingCar(boolean isOn,
        String license,
        boolean isSelfDriving) {
6
        super(isOn, license);
        this.isSelfDriving = isSelfDriving;
9    }
11
    @Override
    void turnOn() { ... }
13
    @Override
    void turnOff() { ... }
15
    void turnSelfDrivingOn() { ... }
    void turnSelfDrivingOff() { ... }
17
    @Override
    public String toString() { ... }
21
}
    
```





```

1 public abstract class Shape {
2     protected String color;
3     protected boolean filled;
4
5     public Shape() { ... }
6     ...
7
8     public abstract double getArea();
9     public abstract double getPerimeter();
10 }
11
12 public class Circle extends Shape {
13     private double radius;
14
15     public Circle() { ... }
16     ...
17
18     @Override
19     public double getArea() { ... }
20
21     @Override
22     public double getPerimeter() { ... }
23 }
    
```

Shape		
m	Shape(String, boolean)	
m	Shape()	
f	color	String
f	filled	boolean
m	getArea()	double
m	getColor()	String
m	getPerimeter()	double
m	isFilled()	boolean
m	setColor(String)	void
m	setFilled(boolean)	void
m	toString()	String



Circle		
m	Circle(double)	
m	Circle()	
m	Circle(String, boolean, double)	
f	radius	double
m	getArea()	double
m	getPerimeter()	double
m	getRadius()	double
m	setRadius(double)	void
m	toString()	String

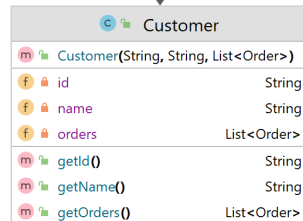
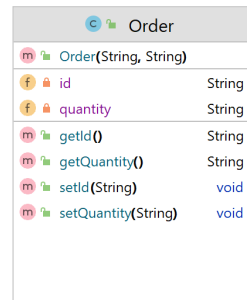


```

1 public class Customer {
    private String id;
3   private String name;
    private List<Order> orders;
5
    public Customer( ... ) {
7       ...
    }
9 }

11 public class Order {
    private String itemId;
13   private String quantity;

15   public Order(String itemId, String quantity) {
        this.itemId = itemId;
17     this.quantity = quantity;
    }
19 }
    
```





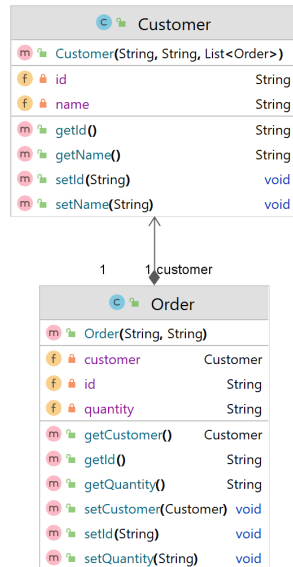
```

1 public class Customer {
    private String id;
3    private String name;

5    public Customer( ... ) {
        ...
7    }
8 }

9
10 public class Order {
11     private String itemId;
12     private String quantity;
13     private Customer customer;

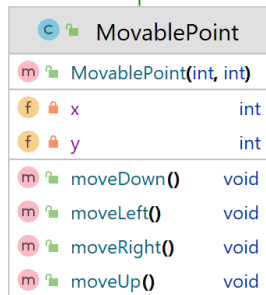
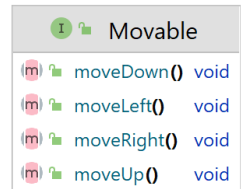
14     public Order( ... ) {
15         ...
16     }
17 }
    
```

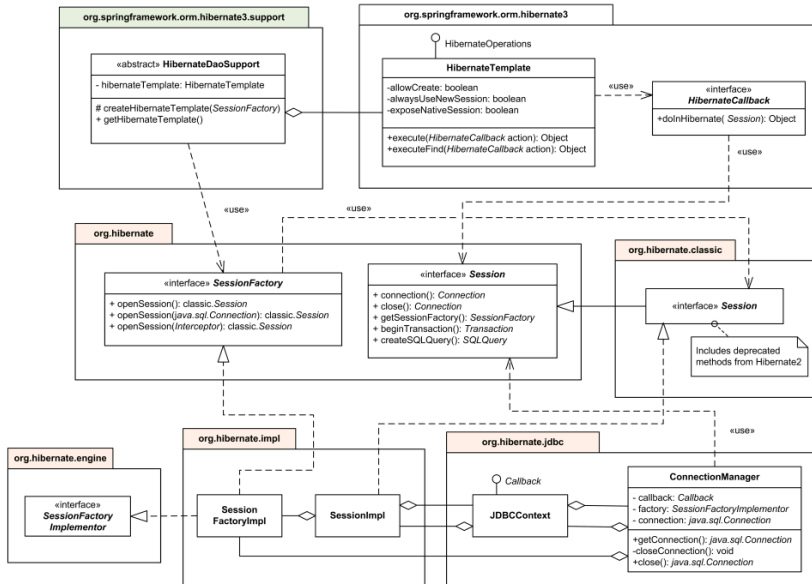













```

1 public interface Movable {
    void moveUp();
3   void moveDown();
    void moveLeft();
5   void moveRight();
    }
7
8   public class MovablePoint implements Movable {
9       private int x;
10      private int y;
11
12      public MovablePoint( ... ) { ... }
13
14      @Override
15      public void moveUp() { ... }
16
17      ...
18  }
    
```





- 1 Abstraction
- 2 Abstract Data Types
- 3 Modelling Tools and Languages
- 4 References**

-  ALLEN B. DOWNEY, CHRIS MAYFIELD, ***THINK JAVA***, (2016).
-  GRAHAM MITCHELL, ***LEARN JAVA THE HARD WAY***, 2ND EDITION, (2016).
-  CAY S. HORSTMANN, ***BIG JAVA - EARLY OBJECTS***, 7E-WILEY, (2019).
-  JAMES GOSLING, BILL JOY, GUY STEELE, GILAD BRACHA, ALEX BUCKLEY, ***THE JAVA LANGUAGE SPECIFICATION - JAVA SE 8 EDITION***, (2015).
-  MARTIN FOWLER, ***UML DISTILLED - A BRIEF GUIDE TO THE STANDARD OBJECT MODELING LANGUAGE***, (2004).
-  RICHARD WARBURTON, ***OBJECT-ORIENTED VS. FUNCTIONAL PROGRAMMING - BRIDGING THE DIVIDE BETWEEN OPPOSING PARADIGMS***, (2016).
-  NAFTALIN, MAURICE WADLER, PHILIP ***JAVA GENERICS AND COLLECTIONS***, O'REILLY MEDIA, (2009).
-  ERIC FREEMAN, ELISABETH ROBSON ***HEAD FIRST DESIGN PATTERNS - BUILDING EXTENSIBLE AND MAINTAINABLE OBJECT***, ORIENTED SOFTWARE-O'REILLY MEDIA, (2020).
-  ALEXANDER SHVETS ***DIVE INTO DESIGN PATTERNS***, (2019).

THANK YOU!