Nested Inner classes

IT Academy

softserve

Agenda

- Nested Classes
  - Non-static Nested Classes
  - Static Nested Classes
  - Local Inner Classes
  - Anonymous Inner Class
- Object cloning
- Wrapper pattern
- Generic in Java

softserve

Nested Classes

- In Java, just like ***methods***, ***variables*** of a ***class*** too can have ***another class as its member***.
- The class written within is called the **nested class**, and the class that holds the inner class is called the **outer class**.
- **Nested classes** are divided into ***two types***:
  - **non-static nested classes** − these are the ***non-static*** members of a class.
  - **static nested classes** − these are the ***static*** members of a class.

softserve

Non-static Nested Classes

- Inner classes are a ***security mechanism*** in Java.

```
public class Person {
    private FullName fullName = new FullName();
    private int age;

    public Person(String firsName, String lastName, int age) {
        fullName.firstName = firsName;
        fullName.lastName = lastName;
        this.age = age;
    }

    // getters and setters

    private class FullName {
        private String firstName;
        private String lastName;
    }
}
```

A class cannot be associated with the access modifier **`private`**, but if we have the class ***as a member*** of other class, then the inner class can be made **`private`**.

softserve

Non-static Nested Classes

- In the given example, we make the inner class **`private`** and access the class through fields.

```
// not allowed
// FullName = fullName = new FullName();

Person person = new Person("Vasyl", "Petrenko", 25);

String fullName = person.getFullName();
int age = person.getAge();

System.out.println(fullName + ", " + age + " years old");
```

```
Vasyl Petrenko, 25 years old
```

Non-static Nested Classes

- If inner class **isn't private** you can ***instantiate it*** !

```
public class Person {
    private int age;

    public Person(int age) {
        this.age = age;
    }

    public class FullName {
        private String firstName;
        private String lastName;

        public FullName(String firstName, String lastName) {
            this.firstName = firstName;   this.lastName = lastName;
        }

        public void info() {
            System.out.println(firstName + " " + lastName + ", " + age + " years old");
        }
    }
}
```

Non-static Nested Classes

- To ***instantiate the inner class***, initially you have to ***instantiate the outer class***.
- Using the ***object of the outer class***, following is the way in which you can ***instantiate the inner class***.

```
Person person = new Person(25);
Person.FullName personWithName = person.new FullName("Vasyl", "Petrenko");
```

or

```
Person.FullName personWithName =
        new Person(25).new FullName("Vasyl", "Petrenko");
```

```
personWithName.info();
```

```
Vasyl Petrenko, 25 years old
```

Static Nested Classes

- A **static inner class** is a ***nested class*** which is a ***static member*** of the outer class.
- It can be ***accessed without instantiating*** the outer class, using other static members.
- Like static members, a static nested class ***does not have access*** to the instance variables and methods of the outer class.

```
class Entity {
    private static int count = 0;

    public Entity() {
        new Counter().setCount();
    }

    public static int getCount() {
        return count;
    }

    private static class Counter {
        private void setCount() {
            count = count + 1;
        }
    }
}
```

softserve

Static Nested Classes

- If you compile and execute the above program, you will get the following result:

```
Entity e1 = new Entity();
Entity e2 = new Entity();
Entity e3 = new Entity();

System.out.println("Count of Entity objects = "
        + Entity.getCount());
```

```
Count of Entity objects = 3
```

softserve

Local Inner Classes

- A **local inner class** can be instantiated ***only within the method*** where it is defined.

```
class Process extends Thread {
    private int randomNumber = 0;
    @Override
    public void run() {
        final int bound = 100;
        class NumberGenerator {
            void setRandomNumber() {
                Random random = new Random();
                randomNumber = random.nextInt(bound);
            }
            void printRandomNumber() {
                System.out.println("Random Number: " +
randomNumber);
            }
        }
        NumberGenerator generator = new NumberGenerator();
        generator.setRandomNumber();
        generator.printRandomNumber();
    }
}
```

It ***cannot access*** the local variables of the enclosing method ***unless*** they are **`final`** or ***effectively final*** and it ***cannot*** be **`private`**

```
new
Process().start();
new
Process().start();
```

```
Random Number: 47
Random Number: 34
```

softserve

Anonymous Inner Class

- **Anonymous inner class** is a class that has ***no name*** and is used if you need to create a ***single instance*** of the class.
- Any ***parameters*** needed to create an anonymous object class, are given in parentheses following name ***supertype*** :

```
new Supertype(list_of_parameters) {
      // body
};
```

Anonymous Inner Class

```
Arrays.sort(people, new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        if (p1.getAge() != p2.getAge())
            return Integer.compare(p1.getAge(), p2.getAge());
        else
            return p1.getName().compareTo(p2.getName());
    }
} );
```

- In this case:
  - operator **`new`** creates an object.
  - The **`Comparator()`** begins definition of anonymous class, similar to:

```
class PersonAgeComparator implements Comparator<Person> { }
```

  - Brace (**`{`**) begins class definition.

softserve

Default Methods for Interfaces

- Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the default keyword. This feature is also known as ***Extension Methods***.

```
public interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }

}
```

Anonymous Class and Default Methods

- Interface **`Formula`** defines a ***default method*** **`sqrt()`** which can be accessed from each formula instance ***including anonymous objects***.

```
Formula formula = new Formula() {
    @Override
    public double calculate(int a) {
        return sqrt(a * 5);
    }
};

double result = formula.calculate(20);

System.out.println("Square root of 100 is " + result);
```

```
Square root of 100 is 10.0
```

softserve

Objects Cloning

- **Object Cloning** is a process of generating the ***exact field-to-field copy*** of object with the ***different*** name.
- The cloned object has ***its own space in the memory*** where it copies the content of the original object.
- For cloning objects in Java you can use **`clone()`** method defined in **`Object`** class.

```
protected Object clone() throws CloneNotSupportedException
```

The `clone()` method

- Example:

```
public class Person {
    private FullName fullName;
    private int age;

    public Person(FullName fullName, int age) {
        this.fullName = fullName;
        this.age = age;
    }
    public static void main(String[] args) {
        Person person = new Person(new FullName("Mike", "Green"), 25);
        try {
            Person copyOfPerson = (Person) personclone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

```
class FullName {
    public String firstName;
    public String lastName;

    // constructor
}
```

```
java.lang.CloneNotSupportedException: com.softserve.itacademy.Person
    at java.base/java.lang.Object.clone(Native Method)
    at com.softserve.itacademy.Person.main(Person.java:25)
```

The `Cloneable` Interface

- The program would throw **`CloneNotSupportedException`** if we don't implement the **`Cloneable`** interface.
- A class implements the **`Cloneable`** interface to indicate to the **`Object.clone()`** method that it is legal for that method to make a field-for-field copy of instances of that class.
- The **`Cloneable`** is a **marker interface**, which means it doesn't has any clone method specification.

The `Cloneable` Interface

- Example:

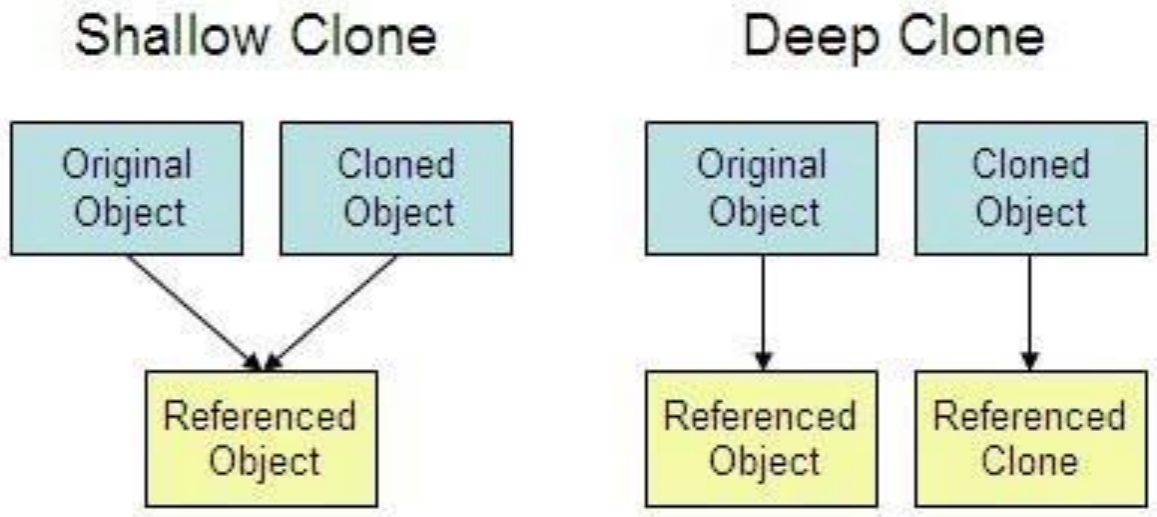
```
Person person = new Person(new FullName("Mike", "Green"), 25);
Person copyOfPerson = (Person) person.clone();

System.out.println("Original: " + person.fullName.firstName + " " +
        person.fullName.lastName + ", " + person.age);
System.out.println("Cloned: " + copyOfPerson.fullName.firstName + " " +
        copyOfPerson.fullName.lastName + ", " + copyOfPerson.age);

copyOfPerson.fullName.firstName = "Nick";
copyOfPerson.fullName.lastName = "Brown";
copyOfPerson.age = 37;

System.out.println("==========================");

System.out.println("Original: " + person.fullName.firstName + " " +
        person.fullName.lastName + ", " + person.age);
System.out.println("Cloned: " + copyOfPerson.fullName.firstName + " " +
        copyOfPerson.fullName.lastName + ", " + copyOfPerson.age);
```

```
Original: Mike Green, 25
Cloned: Mike Green, 25
==========================
Original: Nick Brown, 25
Cloned: Nick Brown, 37
```

softserve

Deep Copy vs Shallow Copy

- **Shallow copy** is "*default implementation*" in Java and if you are not cloning all the object types (not primitives), then you are making a shallow copy.
- In the **Deep copy**, we create a clone which is independent of original object and making changes in the cloned object should ***not affect*** original object.



softserve

Deep Copy

- Example:

```
public class Person implements Cloneable {

    // ...
    @Override
    protected Object clone() throws CloneNotSupportedException {
        Person copyOfPerson = (Person) super.clone();
        copyOfPerson.fullName = (FullName)copyOfPerson.fullName.clone();
        return copyOfPerson;
    }
}
```

```
class FullName implements Cloneable {

    // ...
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

softserve

Wrapper Pattern

Non-generic Box class

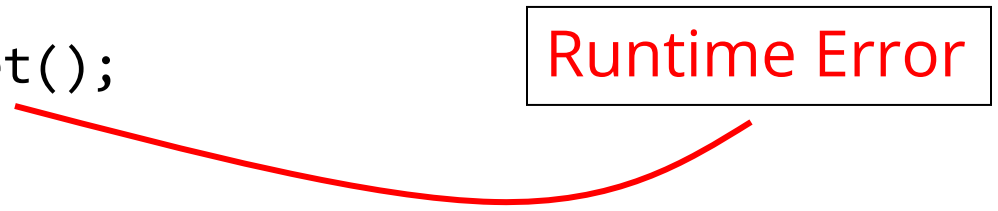
```
public class Box {
  private Object obj;
  public void set(Object obj) { this.obj = obj; }
  public Object get( ) { return obj; }
}
```

Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types

There is no way to verify, at compile time, how the class is used

Wrapper Pattern

```
public class Appl {
  public static void main(String[ ] args) {
      String text = "Hello World";
      Box box = new Box();
      box.set(text);
      Integer i = (Integer) box.get();
    }
}
```



One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

softserve

Wrapper Pattern

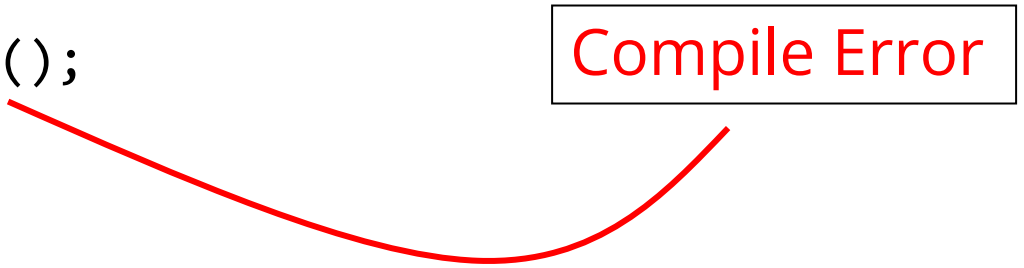
Wrapper (or Decorator) is one of the most important design patterns.

One class takes in another class, both of which extend the same abstract class, and adds functionality

```
public class WrapperBox {
  private Box box = new Box();
  public void set(String text) { this.box.set(text); }
  public String get( ) { return box.get(); }
}
```

Wrapper Pattern

```
public class Appl {
  public static void main(String[ ] args) {
      String text = "Hello World";
      WrapperBox box = new WrapperBox();
      box.set(text);
      Integer i = (Integer) box.get();
    }
}
```



The basic idea of a wrapper is to call-forward to an underlying object, while simultaneously allowing for new code to be executed just before and/or just after the call

Generic in Java

Generics, introduced in Java SE 5.0

- A generic type is a generic class or interface that is parameterized over types.
- Generics add a way to specify concrete types to general purpose classes and methods that operated on Object before.
- With Java's Generics features you can set the type for classes.

Generic class is defined with the following format:

```
class Name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name.

Generic in Java

To update the Box class to use generics, you create a generic type declaration by changing the code

```
public class Box
```

to

```
public class Box<T>
```

This introduces the type variable, T, that can be used anywhere inside the class.

To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

```
Box<Integer> integerBox = new Box<Integer>();
```

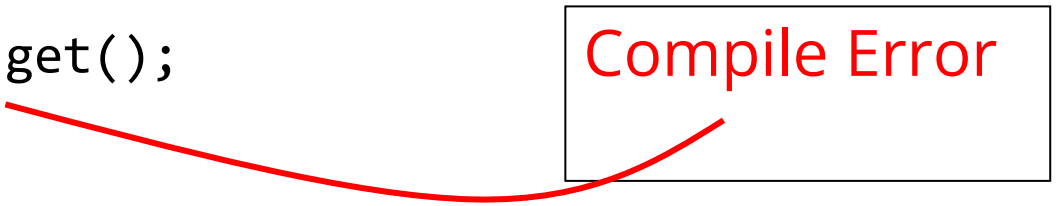
Generic in Java

```
public class Box<T> {
    // T stands for "Type".
    private T t;
    public void set(T t) { this.t = t; }
    public T get( ) { return t; }
}
```

All occurrences of Object are replaced by T.
A type variable can be any non-primitive type you specify: any class type, any interface type, any array type, or even another type variable.
The same technique can be applied to create generic interfaces.

Generic in Java

```
public class Appl {
  public static void main(String[ ] args) {
     String text = "Hello World";
     Box<String> box = new Box<String>();
     box.set(text);
     Integer i = (Integer) box.get();
  }
}
```



Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

softserve

Type Parameter Naming Conventions

- The most commonly used type parameter names are:
  - **`E`** – element (used extensively by the Java Collections Framework)
  - **`K`** – key
  - **`N`** – number
  - **`T`** – type
  - **`V`** – value
  - **`S`**, **`U`**, **`V`** etc. – 2nd, 3rd, 4th types

Generic in Java

Java method can be parametrized, too:

```
<T> getRandomElement(List<T> list) { ... }
```

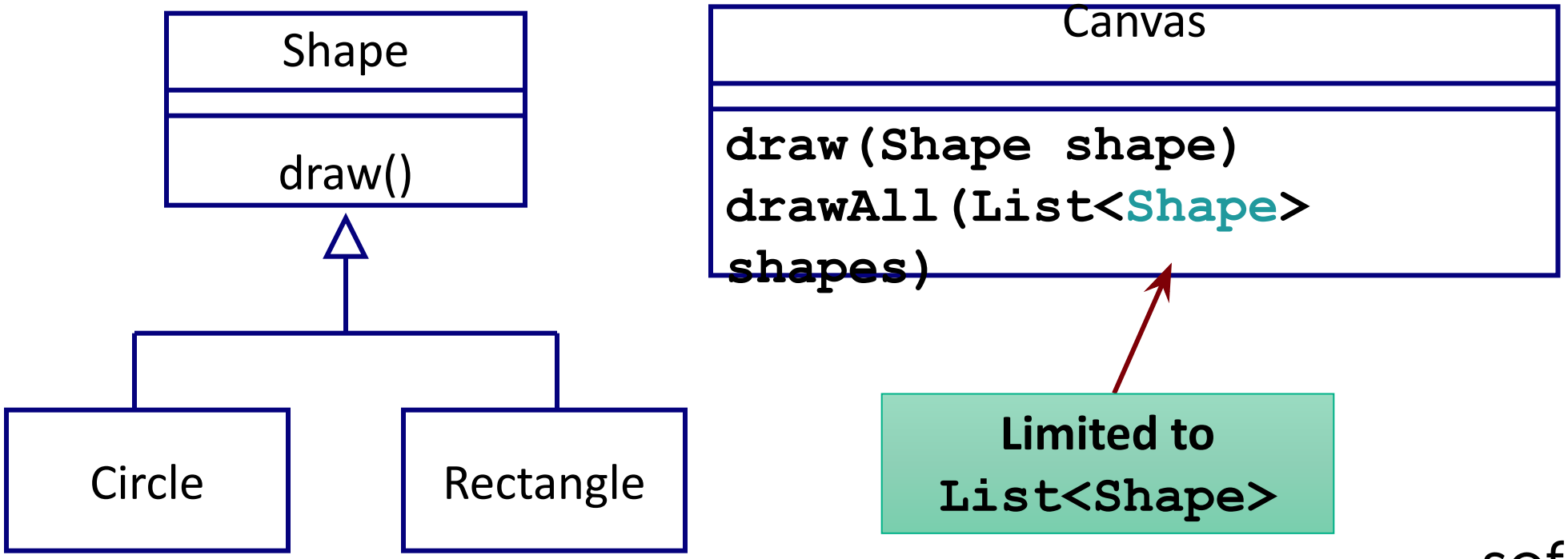
As with class definitions, you often want to restrict the type parameter in the method.

For example, a method which takes a list of Vehicles and returns the fastest vehicle in the list can have the following type.

```
<T extends Vehicle> T getFastest(List<T> list) {...}
```

Bounded Arguments

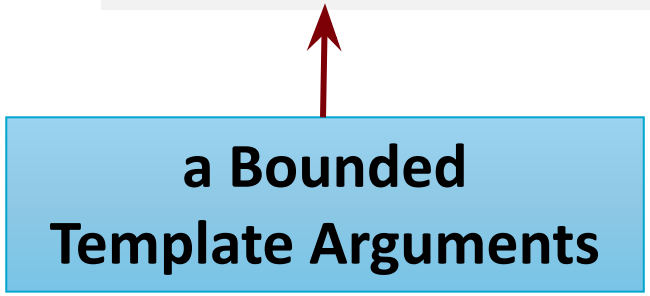
- Consider a simple drawing application to draw shapes (*circles*, *rectangles*, …)



Bounded Arguments

- A List of any kind of **`Shape`** …

```
<T extends Shape> void drawAll(List<T> shapes)
```

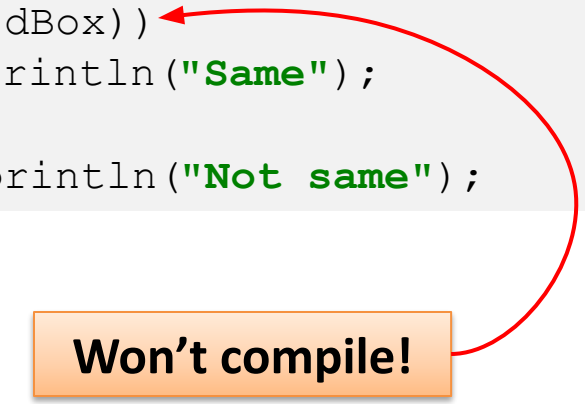


- Shape is the ***upper bound*** of the wildcard.

Template Arguments

```
class Box<T extends Number> {
    T[] nums;

    Box(T[] o) { nums = o; }

    double average() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++) {
            sum += nums[i].doubleValue();
        }
        return sum/nums.length;
    }

    boolean sameAvg(Box<T> obj) {
        return average() == obj.average();
    }
}
```

```
Integer inums[] = {1, 2 , 3 , 4, 5};
Double dnums[] = {1.0, 2.0, 3.0, 4.0, 5.0};

Box<Integer> iBox = new Box<Integer>(inums);
Box<Double> dBox = new Box<Double>(dnums);

if(iBox.sameAvg(dBox))
    System.out.println("Same");
else
    System.out.println("Not same");
```



Template Arguments

```
boolean sameAvg(Box<? extends Number> obj) {
    if(average() == obj.average())
        return true;
    return false;
}
```

- The "**collection of unknown**" is a collection whose element type matches anything – ***template arguments***.

More fun with Generics

```
public void pushAll(Collection<? extends E> collection) {
    for (E element : collection) {
        this.push(element);
    }
}
```

All elements must be ***at least*** an **E**

```
public List<E> sort(Comparator<? super E> comp) {
    List<E> list = this.asList();
    Collections.sort(list, comp);
    return list;
}
```

The comparison method must require ***at most*** an **E**

Generic in Java

Disadvantages

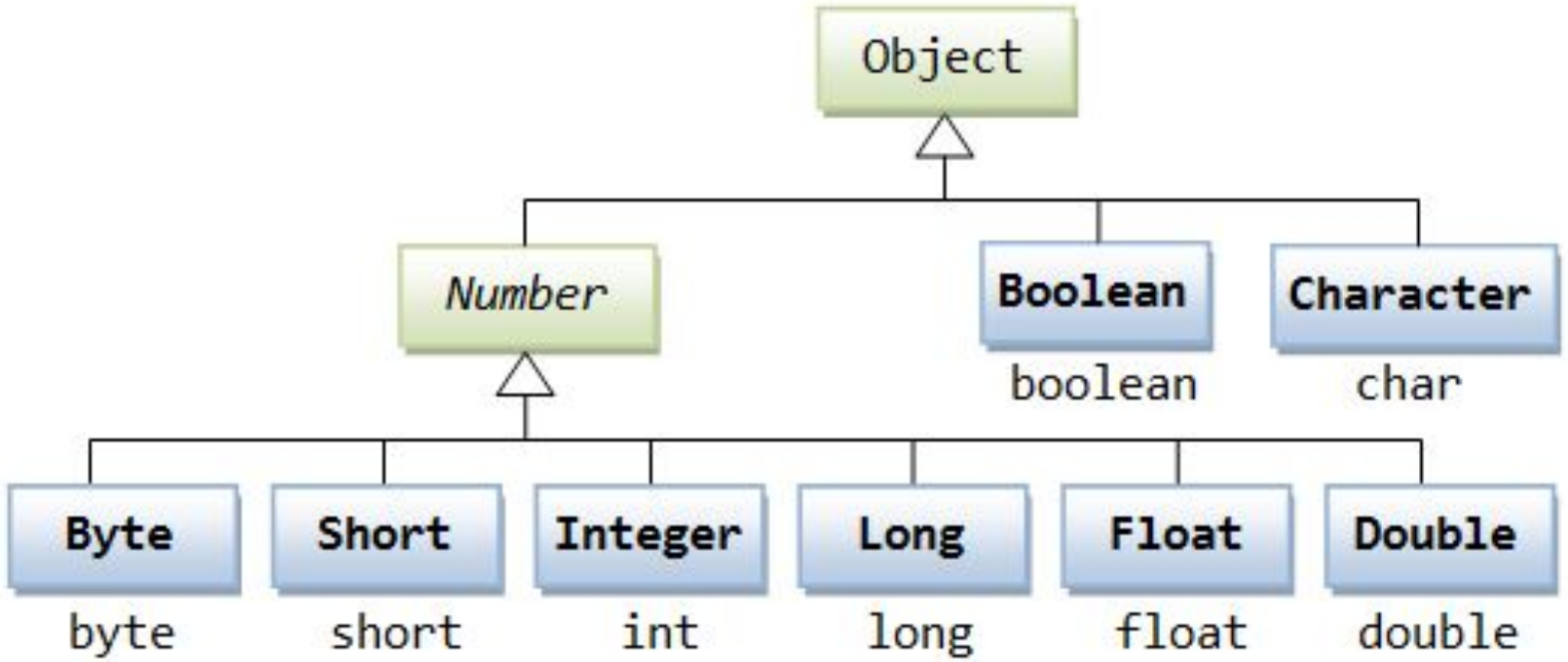
- Generic-fields can not be static.
- Static methods can not have generic parameters or use generic fields.
- Can not be made an explicit call to the constructor generic-type:

```
class Optional<T> {
      T value = new T();
}
```

The compiler does not know what constructor can be caused and the amount of memory to be allocated when an object.

Wrapper Classes

- The **wrapper classes** are objects ***encapsulating primitive Java types***.
- Each Java primitive has a corresponding **wrapper**:



Autoboxing and Unboxing

- After Java 5 the ***conversion primitive value to a wrapper object*** and ***from a wrapper object to a primitive value*** can be done **automatically** by using features called **autoboxing** :

```
Box<Integer> number = new Box<>();

number.set(1);      // autoboxing
Integer val = 2;    // autoboxing
```

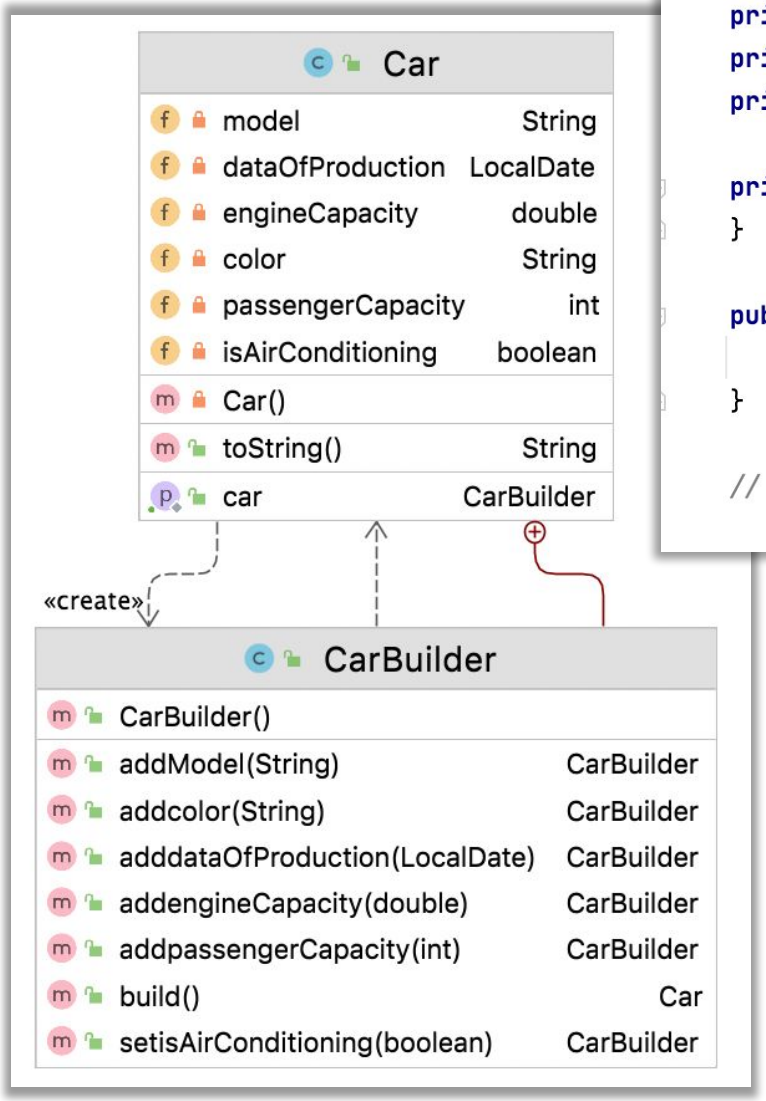
and **unboxing** :

```
Integer object = new Integer(1);

int val1 = getSquareValue(object);  //unboxing
int val2 = object;                  //unboxing

public static int getSquareValue(int i) {
    return i*i;
}
```

softserve

Practical part

1. Suppose we have the class `Car`. Create public inner class `CarBuilder` inside of `Car` class correspond to the next class diagram.
   Create a car with four different parameters and print info about this car and its parameters



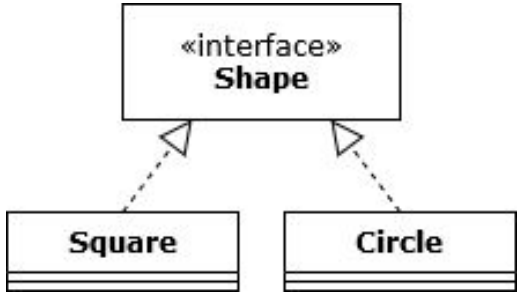

```java
public class Car {

    private String model;
    private LocalDate dataOfProduction;
    private double engineCapacity;
    private String color;
    private int passengerCapacity;
    private boolean isAirConditioning;

    private Car() {
    }

    public static CarBuilder getCar() {
        return new Car().new CarBuilder();
    }

    // TODO inner class, toString method ...
```

softserve

:

Practical part

2. Suppose we have the next diagram



Create `Wrapper` class which should wrap any objects which implements `Shape` interface

For example,

```
Wrapper<Shape> squareWrapper = new Wrapper<>(new Square()); // Good
Wrapper<String> stringWrapper = new Wrapper<>("Hello!");    // Wrong
```

Homework

Task 1

- Develop a `FullName` class with the `firstName` and `lastName` fields of type `String`, which would correspond to the principle of encapsulation.
- Create an abstract `Person` class with `fullName` field of type `FullName` and age of type int.
- In the `Person` class, create a constructor
  `public Person(FullName fullName, int age)`
  and a method called `info()`, which will return a string in the format

  `"First name: <firstName>, Last name: <lastName>, Age: <age>"`

  and an abstract public `activity()` method with a `String` return type.

Homework

- Develop a `Student` class with an `int` field that matches the course the student is taking.
- In the `Student` class, create a constructor with parameters to initialize all fields in the class, and override the `info()` method (which would also add course information to the previous line), and the `activity()` method from the `Person` class. The `activity()` method should return a string value that is the type of activity for the corresponding `Person` subtype, for example for a student - this could be the value "I study at university".
- In the `main(...)` method, create two instances of the `Student` class and output information about them by calling the appropriate methods `info()` and `activity()`.

Homework

Task 2

- Create `Wrapper<T>` class with private field of `T` type which is called `value`.
- In `Wrapper` class create public constructor and `setValue` and `getValue` methods for `value` field.
- Create three objects of the `Wrapper type:` first object should be wrapper for `int type`, second – for `String`, third - for `boolean`.
- Print all three values in the console using method `getValue` from `Wrapper` class.

Thanks for attention!