תרגול 3
תכנות מונחה עצמים

נושאים:

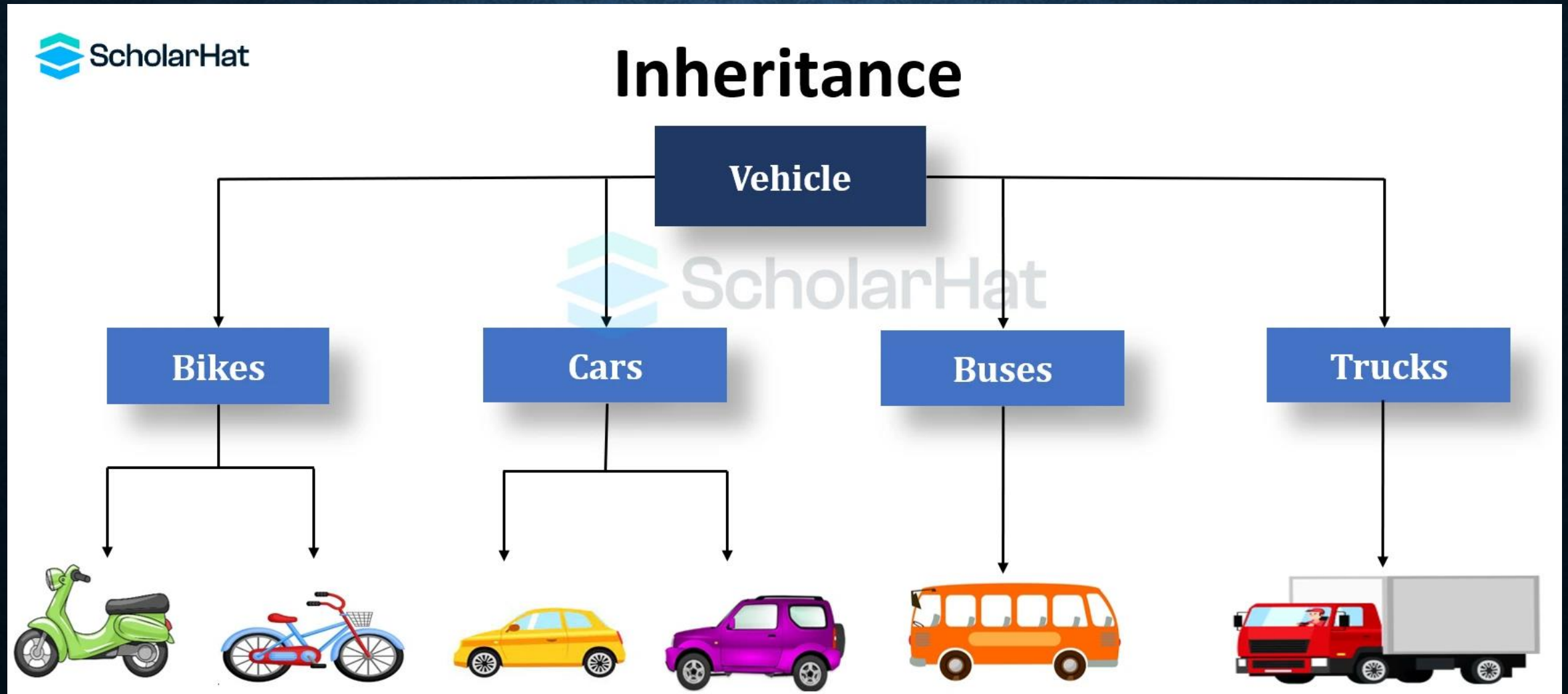
- ירושה
- ממשקים
- מחלקות אבסטרקטיות
- Comperator
- Iterator

ירושה

מוטיבציה:

- נניח שאנו רוצים לבנות מחלקה של student, לכל סטודנט יש מאפיינים כמו ת.ז, שם, תאריך לידה וממוצע ציונים
- ואם נרצה לבנות מחלקה של employee, גם כאן לכל עובד יש ת.ז, שם, תאריך לידה ושכר חודשי
- הבעיה – אנחנו מגדירים במחלקות שונות שוב ושוב את אותן השדות וזה יגרור כמובן גם כתיבה של עוד ועוד מתודות
- אבחנה פשוטה – מרבית מהשדות של employee מגדירים את המחלקה person, והמחלקה שלנו רק מרחיבה אותה

ירושה


ScholarHat
Inheritance
Vehicle
Bikes
Cars
Buses
Trucks
ScholarHat

ירושה

הגדרה:

המטרה של ירושה זה לקחת מחלקה קיימת ולהרחיב אותה למשהו גדול יותר

לדוגמא, לקחת בן אדם ולהוסיף לו את הפונקציונאליות הדרושה בשביל להיות סטודנט\עובד

מחלקה A שיורשת את מחלקה B מגדירה יחס של B הוא "סמ" A
למשל – סטודנט הוא סמ בן אדם

- המחלקה ממנה יורשים נקראת מחלקת האב (superClass) והמחלקה היורשת נקראת מחלקת הבן (subclass)

ירושה

איך זה עובד?

- ירושה בjava מתבצעת באמצעות המילה השמורה extends
- המחלקה היורשת מקבלת באופן אוטומטי גישה לכל השדות והמתודות של מחלקת האב
- בjava כל מחלקה יכולה לרשת מחלקה אחת בלבד
- הירושה טרנזטיבית – כלומר אם C יורש את B שיורש את A, זה אומר שC יורש גם את A ויש לו גישה לכל השדות והמתודות שלה
- כל המחלקות בjava יורשות ממחלקת object באופן אוטומטי

```
public class Student extends Person{

}
```

ירושה

למה להשתמש בירושה?

- חוסך משמעותית בכתיבת קוד חוזר
- מפשט מחלקות שעלולות להיות מורכבות לקטנות יותר
- מאפשר לשנות מתודות מבלי לשנות את מימוש המחלקה

- Inheritance allows you to create new class based on another one.
- New class can extend existing one and share properties and behavior.
- New class is called child class and basic class is called parent class.
- Inheritance allows you to reuse code and to create new classes without reinventing the wheel.

ממשק - INTERFACE

מוטיבציה:

יש עצמים רבים בעולם ששונים בצורת ההתנהגות שלהם אך בבסיס שלהם הם אותו דבר

למשל:

- חתול, חמור וכלב הם סוגים שונים של בעלי חיים

  - כולם משמיעים קול, אבל לכל אחד יש קול שונה

- עיגול, ריבוע ומשולש הם סוגים שונים של צורה

  - לכולם יש פונקציה של חישוב שטח, אבל לכל צורה חישוב השטח נעשה בצורה אחרת

INTERFACE - ממשק

הגדרה:

המטרה של ממשק זה לאגד עצמים שונים תחת אותה קטגוריה, ולספק להם תשתית למימוש המחלקות השונות בצורה אחידה

מחלקה A שמממשת את מחלקה B מגדירה יחס של B זה "סוג של" A
למשל – עיגול זה סוג של צורה

- ממשק משמש כסוג של חוזה שמי שמממש אותו חייב לכבד

interface - ממשק

איך זה עובד?

- מימוש ממשק בjava מתבצע באמצעות המילה השמורה implements
- המחלקה המממשת חייבת לממש את כל השדות המוגדרים בממשק
- בjava ניתן לממש כמה ממשקים שרוצים
- ממשק מכיל רק **הצהרה** על מתודות, לא מימוש של המתודות וללא שדות (למעט שדות מסוימים שמוגדרים להיות אוטומטית static final בלי אפשרות שינוי)

```
public class Circle implements Shape {
```

ממשק - interface

<u>למה להשתמש בממשק?</u>

- גורם לקוד להיות בהיר ומובן, על ידי פונקציונאליות ידועה מראש
- מכריח את המתכנת לממש את כל המחלקות בצורה זהה, ומאפשר למחלקות לתקשר ביניהן
- מאפשר להתייחס לעצם מהסוג של הממשק ולהשתמש בפונקציונאליות שלו מבלי לדעת את הסוג הספציפי שלו
- מאפשרות לשייך מחלקה לסוג אליו היא שייכת לפי המתודות המוגדרות בה (למשל, מחלקה המממשת ממשק עם מתודות push-pop תוגדר כסוג של stack)

INTERFACE - ממשק

מימוש ממשק באמצעות מחלקה אנונימית:

מלבד מימוש ממשק על ידי מחלקה רגילה, ניתן לממש באמצעות מחלקה אנונימית

```
public static void main(String[] args) {
    Shape circle = new Shape() {
        2 usages
        private double radius = 5.0;
        1 usage
        @Override
        public double area() {
            return Math.PI * radius * radius;
        }
    };
    System.out.println("Circle Area: " + circle.area());
}
```

שימושים

- כאשר כותבים מחלקה לשימוש חד פעמי
- כאשר כותבים כמה מחלקות קטנות עם מימוש שונה בכל אחת מהם

אזהרה: זה עלול לסבך את הקוד ולהקשות על ההבנה שלו, מומלץ רק במקרים מסוימים!

INTERFACE - ממשק

מימוש באמצעות ביטוי lambda:

אם הממשק מכיל הצהרה על מתודה אחת בלבד, ניתן לממש את הממשק על ידי ביטוי lambda שמשמעותו תהיה מימוש הפונקציה הבודדת:

```
public static void main(String[] args) {

    double radius = 5.0;
    Shape circle = () -> {
            return Math.PI * radius * radius;
        };

    System.out.println("Circle Area: " + circle.area());
}
```

שימושים

- כפרמטר לפונקציה
- כאשר מדובר במימוש קטן ונקודתי למשהו ספציפי

ABSTRACT CLASS

מוטיבציה:

- במצב בו יש כמה מחלקות שמממשות את אותו אובייקט וחלק מהמתודות שלהם ממומשות באותה הדרך – יש כתיבה כפולה של אותו קוד
- אם נרצה להגדיר שדות שמשותפים לכל המחלקות לא ניתן לעשות זאת בממשק
- אם נרצה לספק מחלקת אב עם מימוש שמהווה בסיס מימוש למחלקות היורשות אותה, ומאפשרת להם לדרוס את המימוש של פונקציות מסוימות במידת הצורך

ABSTRACT CLASS

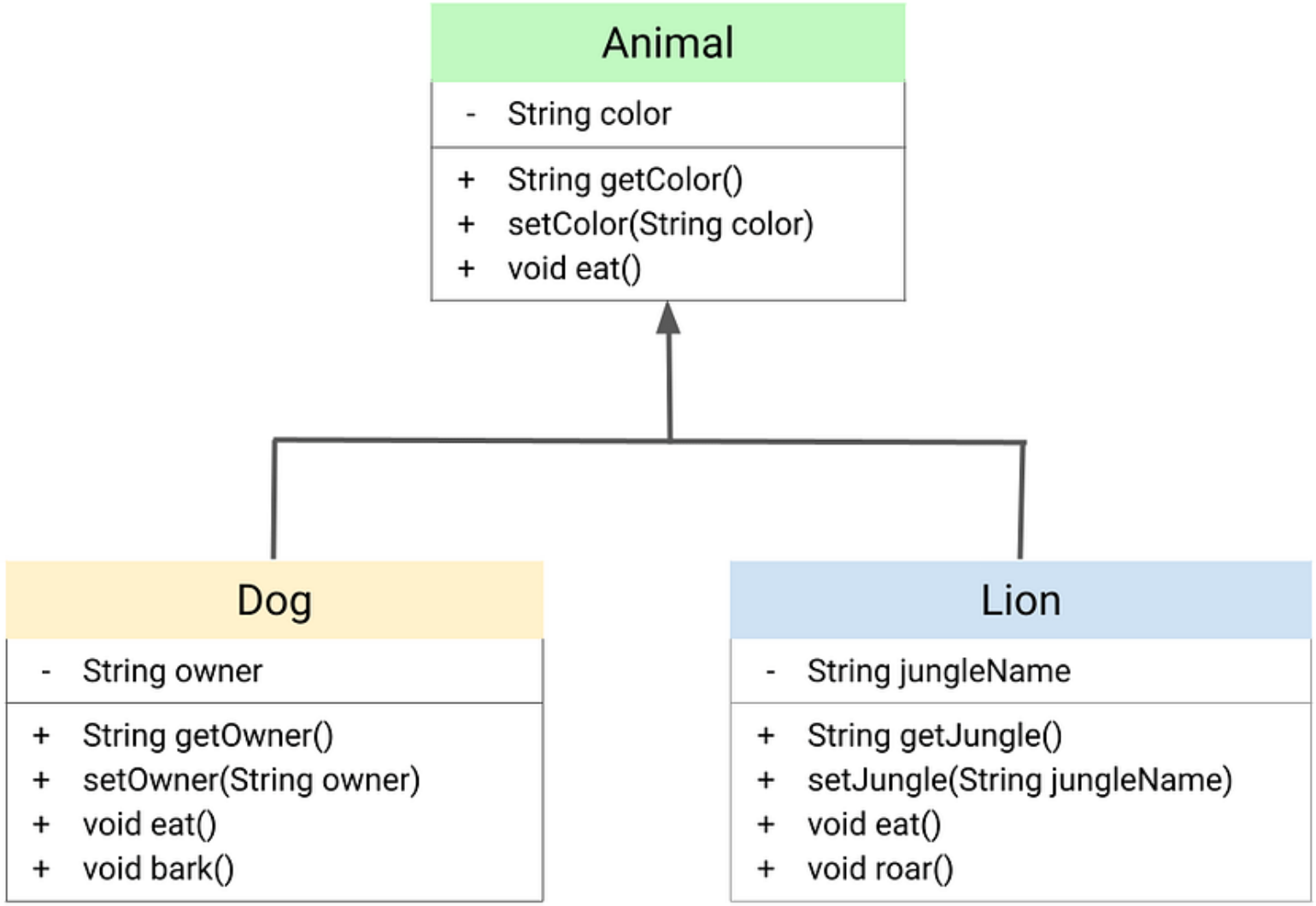
הגדרה:

מחלקה אבסטרקטית היא מחלקה רגילה, מלבד כך שאי אפשר ליצור ממנה אובייקטים באופן ישיר.

המטרה של מחלקה אבסטרקטית היא להגדיר אוסף של תכונות המשותפות לכמה מחלקות, ולשמש רק כבסיס ליצירת מחלקות קונקרטיות (מחלקות שיש להן מימוש מלא לכל המתודות שלהן) שמהן יהיה ניתן ליצור אובייקטים

בפשטות, כל מטרתה היא לאפשר למחלקות אחרות לרשת ממנה ולהשתמש בשדות והמתודות שהיא הגדירה, והיא משמשת כמחלקת בסיס עם קוד כתוב שממנה משנים ובונים עוד בהתאם לצורך

ABSTRACT CLASS



ABSTRACT CLASS

איך זה עובד?

- ירושה ממחלקה אבסטרקטית עובדת בדיוק כמו ירושה רגילה, עם extends
- ניתן להגדיר במחלקה אבסטרקטית שדות כרצוננו
- ניתן לממש רק חלק מהמתודות וחלק להשאיר ללא מימוש (צריך להצהיר על כך באמצעות הוספת abstract בחתימה של הפונקציה)
- מחלקה אבסטרקטית יכולה לרשת ממחלקות אחרות וגם לממש ממשקים

```
public abstract class abs  {
```

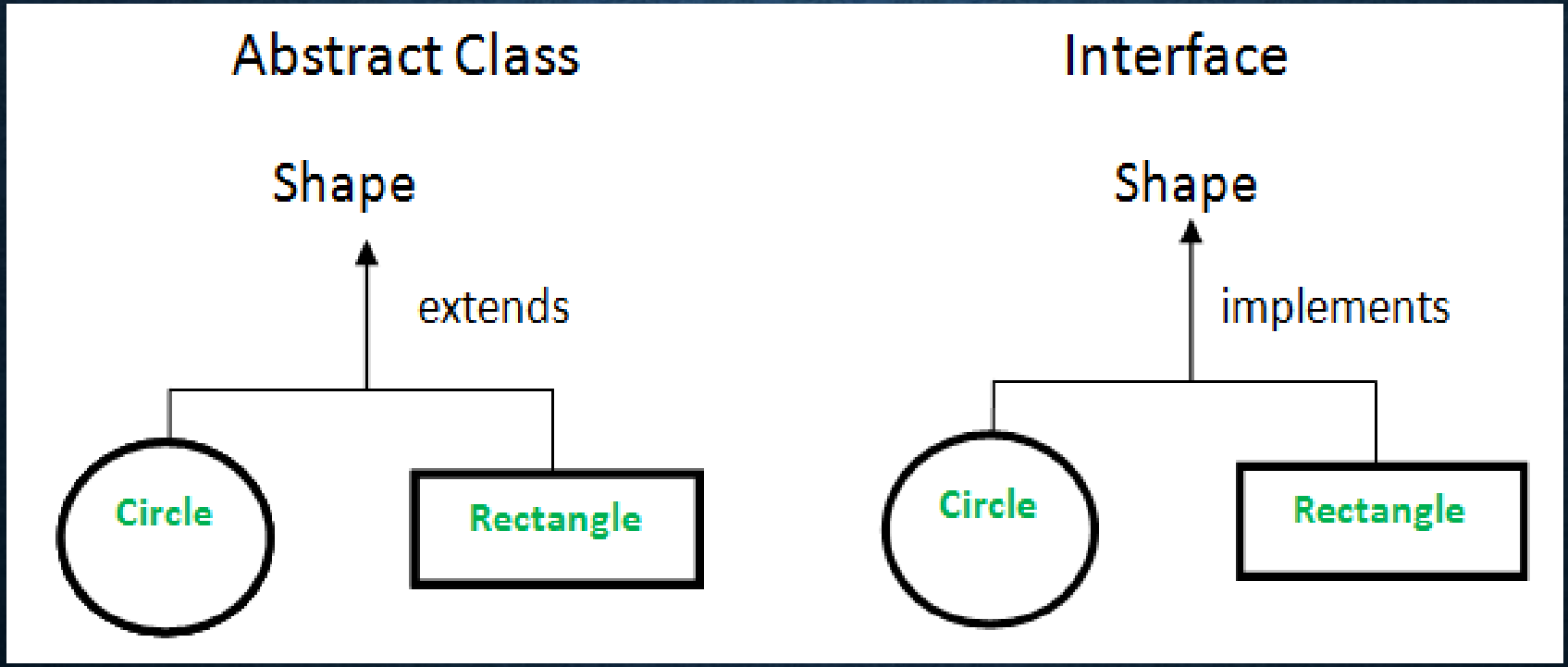
```
public abstract int foo();
```

ABSTRACT CLASS

למה להשתמש בabstract class?

- מאפשר בנייה מלאה של מחלקה, עם אפשרות לשנות אותה בהתאם לצרכים במחלקה היורשת
- כדי לאפשר מימוש של מתודות בעלות מימוש זהה, והצהרה ללא מימוש של מתודות בעלות מימוש נפרד
- כדי לאפשר הגדרה של שדות
- חוסך קוד כפול במספר מחלקות ומאפשר תחזוקה ושינויים של הקוד בקלות

INTERFACE VS. ABSTRACT CLASS



INTERFACE VS. ABSTRACT CLASS

Abstract class	Interface
ירושה באמצעות extends	מימוש באמצעות implements
בjava ניתן לרשת מחלקה אחת בלבד	ניתן לממש כמה ממשקים שנרצה
ניתן להצהיר על שדות	לא ניתן להצהיר על שדות (למעט קבועים)
ניתן לממש את המתודות	רק הצהרה על מתודות ללא מימוש
ניתן להצהיר ולממש בנאי	לא ניתן להצהיר על בנאי
ניתן <u>לממש</u> ממשקים אחרים	ניתן <u>לרשת</u> ממשקים אחרים (אפילו יותר מאחד)
ניתן לרשת מחלקה אחרת	לא ניתן לרשת מחלקה
יכול להכיל את כל סוגי השדות	יכול להכיל רק שדות קבועים בלבד

Comparator

d

Comparator Interface: The Comparator interface in Java allows custom sorting of objects that do not implement the Comparable interface.

Custom Sorting: While Comparable provides a natural ordering for objects, Comparator lets you define your own sorting logic.

Comparison Logic: Implement the compare method of the Comparator interface to define how objects should be compared for ordering.

Flexible Sorting: You can create multiple Comparator implementations to sort objects differently without modifying their original class.

Use Cases: Sorting collections of complex objects, objects of library classes, or third-party classes that you can't modify.

Anonymous Comparators: You can define Comparator instances inline using anonymous classes or lambda expressions.

Option 1: Using a Separate Comparator Class

```java
import java.util.Comparator;

public class StudentAgeComparator implements Comparator<Student>
{   @Override
    public int compare(Student s1, Student s2) {
        return Integer.compare(s1.getAge(), s2.getAge());
    }
}
```

```java
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student("Alice", 21),
            new Student("Bob", 19),
            new Student("Charlie", 20)
        );

        // Using the separate Comparator class
        students.sort(new StudentAgeComparator());

        for (Student student : students) {
            System.out.println(student.getName() + " - " +
student.getAge());
    }
}
```

שימו לב!!

הדוגמא כאן מראה מחלקה נפרדת (חיצונית) לצורך השוואה אפשר ואף רצוי מבחינה עיצובית לממש מחלקה זו כמחלקה פנימית בתוך המחלקה עצמה

Option 2: Using Anonymous Class

```java
List<Student> students = Arrays.asList(
    new Student("Alice", 21),
    new Student("Bob", 19),
    new Student("Charlie", 20)
);

Comparator<Student> ageComparatorAnonymous = new Comparator<Student>() {
    @Override
    public int compare(Student s1, Student s2) {
        return Integer.compare(s1.getAge(), s2.getAge());
    }
};

students.sort(ageComparatorAnonymous);

for (Student student : students) {
    System.out.println(student.getName() + " - " + student.getAge());
}
```

Option 3: Using Lambda Expression Student

```java
List<Student> students = Arrays.asList(
    new Student("Alice", 21),
    new Student("Bob", 19),
    new Student("Charlie", 20)
);

Comparator<Student> ageComparatorLambda = (s1, s2) -> Integer.compare(s1.getAge(), s2.getAge());

students.sort(ageComparatorLambda);

for (Student student : students) {
    System.out.println(student.getName() + " - " + student.getAge());
}
```

Iterators

Iterator Interface: The Iterator interface in Java provides a way to iterate over elements in a collection without exposing the underlying implementation details.

Traversal of Collections: Iterators allow sequential access to elements in collections like lists, sets, and maps.

Advantages: Iterators are memory-efficient as they don't require creating additional data structures to store the collection's elements.

Basic Methods:

- boolean hasNext(): Returns true if there are more elements to iterate over.
- next(): Returns the next element in the collection and advances the iterator.
- void remove(): Removes the last element returned by next().

```java
List<String> fruits = Arrays.asList("Apple", "Banana", "Orange");
Iterator<String> iterator = fruits.iterator();

while (iterator.hasNext()) {
    String fruit = iterator.next();
    System.out.println(fruit);
}

/**
* Enhanced For Loop:
* Java provides a convenient way to use an iterator
* behind the scenes using an enhanced for loop:
**/
for (String fruit : fruits) {
    System.out.println(fruit);
}

```