Chapter Two

Core Object Oriented Technologies with Java

Chapter Outlines

- Java Language Fundamentals
- Encapsulation
- Polymorphism
- Implementation
- Abstraction

What is a Java Class?

- Classes are constructs that define objects of the same type
- A Java class uses variables to define ***data*** fields and methods to define ***behaviors***
- Class provides a special type of methods, known as constructors, which are invoked to construct objects from the class

A Student Class

```
public class Student {
  private int studentID;
  private String studentName;

  public Student(int id, String name) {
    this.studentID = id;
    this.studentName = name;
  }

  public int getStudentID() {
    return studentID;
  }

  public String getStudentName() {
    return studentName;
  }
}
```

About Student Class

- A class with capitalized name *Student*
- It has two variables as in *Line-2* and *Line-3*
    - studentID, studentName
    - They are defined as private type, and not readable by other classes
- It has a constructor as in *Line-5*
    - public Student(int id, String name)
    - It will create an instance of *Student* class
- It has two methods defined as public as in *Line-10* and *Line-14*, and other classes can access to them
    - public int getStudentID()
    - public String getStudentName()

Create Student instances

- We can easily construct two *Student* ***objects*** with the *Student* class
- We can send the object a ***message*** by triggering the method as in *Line-3* (*getStudentName()*) and behave what it is defined in the method
- Each object instance contains its own set of ***data*** (*studentID* and *studentName*) and ***behaviors*** (*getStudentID()* and *getStudentName()*)

```
01.        Student johnSmith = new Student(1234, "John Smith");
02.        Student peterPan = new Student(5678, "Peter Pan");
03.        System.out.println(johnSmith.getStudentName());
04.        System.out.println(peterPan.getStudentName());
```

What is an object?

- An object is a tangible entity that exhibits some well-defined behaviors
- Stefik and Bobrow define objects as “entities that combine the properties of procedures and data since they perform computations and save local state.”
- An object is an ***instance*** of a class

Class Variables & Class Methods

- In Java, there is a keyword *static* for defining class level variables and methods
  - They are unique to object level
- These static variables and methods can be accessed directly without creating an instance
- Class Variable
  - public *static* final double PI = 3.1415926535
- Class Method
  - public *static* double sin(double x)

A ESAPStudent Class

```
import java.text.DecimalFormat;

public class ESAPStudent {
  // Class Level Variables
  public final static String SCHOOL_NAME = "School of Public Administration";
  public static int NUM_OF_STUDENTS = 0;

  // Local Variables
  private int studentID;
  private String studentName;

  // Constructor
  public ESAPStudent(int id, String name) {
    this.studentID = id;
    this.studentName = name;
    NUM_OF_STUDENTS++;
  }
```

ESAPStudent Class (cont.)

```java
  public int getStudentID() {
    return studentID;
  }

  public String getStudentName() {
    return studentName;
  }

  // Class Method
  public static String formatID(int id) {
    DecimalFormat df = new DecimalFormat("00,00,00");
    String newID = df.format(id);
    newID = "P-" + newID.replaceAll(",", "-");
    return newID;
  }
}
```

Create ESAPStudent Objects

```
public class TestESAPStudent {
  public static void main(String args[]) {
    ESAPStudent johnSmith = new ESAPStudent(99, "John Smith");
    System.out.print(ESAPStudent.SCHOOL_NAME + " has ");
    System.out.println(ESAPStudent.NUM_OF_STUDENTS + " student(s)");
    System.out.println(ESAPStudent.formatID(johnSmith.getStudentID()));
    System.out.println(johnSmith.getStudentName());

    ESAPStudent peterPan = new ESAPStudent(999, "Peter Pan");
    System.out.print(ESAPStudent.SCHOOL_NAME + " has ");
    System.out.println(ESAPStudent.NUM_OF_STUDENTS + " student(s)");
    System.out.println(ESAPStudent.formatID(peterPan.getStudentID()));
    System.out.println(peterPan.getStudentName());
  }
}
```

About ESAPStudent Object

- *ESAPStudent* class has two instances
  - johnSmith and peterPan
  - We can create many ESAP students (multiple instances)
- *ESAPStudent* class has two class level variables
  - SCHOOL_NAME, NUM_OF_STUDENTS
  - There is only one copy of them (constant variables)
  - Each instance of MPI Student object accesses the same variables of SCHOOL_NAME, NUM_OF_STUDENTS
- *ESAPStudent* class has one class level method
  - public static String formatID(int id)
  - Each instance of MPI Student object accesses the same method

Stateful and Stateless Objects

- What are stateful and stateless objects?
  - Stateful objects for individual user (multiple instances)
  - Stateless objects for all users (constant variables)
- Stateful objects are not shared among users
  - Each object has different set of value and each hold its own state for individual user
  - Changing the state of an object will not affect the others
- Stateless object serves all users
  - It is the same object visible to all users but doesn’t hold state for individual user

Example of Stateful and Stateless

- Stateful
  - Multiple instances
  - User level variables
  - Hold state for each user to have different set of value
  - E.g. shopping carts
- Stateless
  - Unique instance
  - Application level variables
  - Does not hold state for individual user
  - E.g. counter

More about the usage of *static*

- We can set a block of code (*Line-9* to *14*) to be *static*
- The codes inside the static block will be executed only **<u>once</u>** on the first execution time

```
01.    import java.util.Arrays;
02.
03.    public class ESAPCourse {
04.      public final static String SCHOOL_NAME = "School of Public Administration";
05.      public static String[] campus = new String[3];
06.      private int courseCode;
07.      private String classroom;
08.
09.      static {
10.        campus[0] = "Chi-Un Building";
11.        campus[1] = "Meng-Tak Building";
12.        campus[2] = "Wui-Chi Building";
```

ESAPCourse Class (cont.)

```
    System.out.println(Arrays.toString(campus));
  }

  public ESAPCourse(int code, String room) {
    this.courseCode = code;
    this.classroom = room;
  }

  public int getCourseCode() {
    return courseCode;
  }

  public String getClassroom() {
    return classroom;
  }
}

```

Creating an instance

- The following program creates two instance of *ESAPCourse* (comp221 and comp413)

```
public class TestESAPCourse {
  public static void main(String args[]) {
    ESAPCourse comp221 = new ESAPCourse(221, ESAPCourse.campus[2]);
    ESAPCourse comp413 = new ESAPCourse(413, ESAPCourse.campus[0]);
  }
}
```

- Since the initialization work for *campus* is inside the *static* block, it will execute only once to save the process time
- Try to omit the keyword *static* on *Line-9* of *ESAPCourse* and run the *TestESAPCourse* program again to see what will happen

Singleton

- Sometimes, an application needs to create an object that is unique to all users (Stateless)
    - A printer pool instance should be the same for all users
- Singleton is designed to restrict clients to create multiple instances (Stateful)
    - Define a `private` constructor as in *Line-4*
    - Create a `private` unique local instance as in *Line-3*
    - Provide a `public` method for client to retrieve the local instance as in *Line-8*

Printer Pool Example

- The printer pool singleton implementation

```
import java.time.OffsetTime;
public class PrinterPool {
  private static PrinterPool instance = new PrinterPool();
  private PrinterPool() {
    System.out.println("Created at " + OffsetTime.now());
  }

  public static PrinterPool getInstance() {
    return instance;
  }

  public void startPrint(String message) {
    System.out.println(message);
  }
}
```

Singleton Instances

- The output shows that different users are using the same instance:

```
Created at 17:37:52.291+08:00
Print Job 1: ipm.esap.comp221.TestPrinterPool@58372a00
Print Job 2: ipm.esap.comp221.TestPrinterPool@58372a00
The same instance.
```

```
public class TestPrinterPool {
  public static void main(String[] args) {
    PrinterPool job1 = PrinterPool.getInstance();
    job1.startPrint("Print Job 1: " + job1);
    PrinterPool job2 = PrinterPool.getInstance();
    job2.startPrint("Print Job 2: " + job2);
    System.out.printf("%she same instance.", (job1 == job2 ? "T" : "Not t"));
  }
}
```

Abstraction: “is a”

- Suppose we have developed two classes
  - *Rectangle* class
  - *Triangle* class
- These two classes have something in common
  - Properties: width and height
  - Functions: calculate the area and perimeter

A Rectangle Class

```
public class Rectangle {
  private double width;
  private double height;

  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }

  private double getArea() {
    return width * height;
  }

  private double getPerimeter() {
    return 2 * (width + height);
  }

```

Rectangle Class (cont.)

```
  private String getName() {
    return this.getClass().getSimpleName();
  }

  public void showInfo() {
    System.out.println(getName() + " Information:");
    System.out.println("Area is " + getArea());
    System.out.println("Perimeter is " + getPerimeter());
  }
}

```

A Triangle Class

```
public class Triangle {
  private double width;
  private double height;

  public Triangle(double width, double height) {
    this.width = width;
    this.height = height;
  }

  private double getArea() {
    return width * height / 2;
  }

  private double getPerimeter() {
    return (width + height) + Math.sqrt(width * width + height * height);
  }

```

Triangle Class (cont.)

```
  private String getName() {
    return this.getClass().getSimpleName();
  }

  public void showInfo() {
    System.out.println(getName() + " Information:");
    System.out.println("Area is " + getArea());
    System.out.println("Perimeter is " + getPerimeter());
  }
}

```

Ellipse

- Now, we need to create another class
  - *Ellipse* class
  - It also has width, height properties
  - It also has area, perimeter functions
- What should we do?
  - Copy the codes?
  - Rewrite a new one from scratch?
- Better to make an abstract class: *Shape*

Abstract Class: Shape

- One of the solutions is to make an abstract class *Shape* to keep the commonality

```
public abstract class Shape {
  public abstract double getArea();
  public abstract double getPerimeter();

  public String getName() {
    return this.getClass().getSimpleName();
  }

  public void showInfo() {
    System.out.println(getName() + " Information:");
    System.out.println("Area is " + getArea());
    System.out.println("Perimeter is " + getPerimeter());
  }
}
```

The new Rectangle Class

```
public class RectangleShape extends Shape {
  private double width;
  private double height;

  public RectangleShape(double width, double height) {
    this.width = width;
    this.height = height;
  }
  @Override
  public double getArea() {
    return width * height;
  }
  @Override
  public double getPerimeter() {
    return 2 * (width + height);
  }
}
```

The new Triangle Class

```
public class TriangleShape extends Shape {
  private double width;
  private double height;

  public TriangleShape(double width, double height) {
    this.width = width;
    this.height = height;
  }
  @Override
  public double getArea() {
    return width * height / 2;
  }
  @Override
  public double getPerimeter() {
    return (width + height) + Math.sqrt(width * width + height * height);
  }
}
```

An Ellipse Class

```
public class EllipseShape extends Shape {
  private double width;
  private double height;

  public EllipseShape(double width, double height) {
    this.width = width;
    this.height = height;
  }
  @Override
  public double getArea() {
    return width * height * Math.PI;
  }
  @Override
  public double getPerimeter() {
    return 2 * Math.PI * Math.sqrt((width * width + height * height) / 2);
  }
}
```

The Benefits of Abstraction

- Super class: *Shape*
- Subclasses: *Rectangle*, *Triangle*, *Ellipse*
- Subclass "**is a**" Super class
  - Rectangle is a Shape, Triangle is a Shape, Ellipse is a Shape
- We can write less codes when creating new shapes (circle, square, pentagon, hexagon, etc.)
- We can focus more on the implementation work (calculate the area and perimeter functions)
- We can centralize the common codes (show info. function) and can change them consistently

Aggregation: “has a”

- If a class has an entity reference, it is known as Aggregation
- Aggregation represents “has a” relationship
- *MPIStudent* class has a *YearTutor* class

```
public class MPIStudent {
  private YearTutor yearTutor;
  private int studentID;
  private String studentName;

  public MPIStudent(int id, String name) {
    this.yearTutor = new YearTutor(id);
    this.studentID = id;
    this.studentName = name;
  }
```

MPIStudent Class (cont.)

```

  public String getTutorName() {
    return yearTutor.getTutorName();
  }

  public int getStudentID() {
    return studentID;
  }

  public String getStudentName() {
    return studentName;
  }
}


```

A YearTutor Class

```
public class YearTutor {
  private String schoolName = "School of Public Administration";
  private String tutorList[] = { "Mario", "Luigi" };
  private String tutorName;

  public YearTutor(int studentID) {
    tutorName = (studentID > 201500000) ? tutorList[0] : tutorList[1];
  }

  public String getSchoolName() {
    return schoolName;
  }

  public String getTutorName() {
    return tutorName;
  }
}
```

The Benefits of Aggregation

- *YearTutor* class can be reused by other classes
- *YearTutor* class can be changed individually
- Code reuse is also best achieved by aggregation when there is no “is a” relationship
- Inheritance should be used only if the relationship “is a” is maintained throughout the lifetime of the objects involved. Otherwise, aggregation is the best choice
- Information hiding: we can leave the *YearTutor* class logic alone and *MPIStudent* will never know

Generic in Java

- The following method sends a message to a unknown class

```
import java.lang.reflect.Method;

public class Sender {
  public static void sendMsg(String className, String methodName, String value) {
    try {
      Class<?> myClass = Class.forName(className);
      Class<?>[] myParms = new Class[1];
      myParms[0] = String.class;
      Method myMethod = myClass.getDeclaredMethod(methodName, myParms);
      myMethod.invoke(myClass.newInstance(), value);
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Generic Example

- The *Sender* class calls a unknown type method of a class with unknown type
- For example, there are two different classes (PrintMessage, and MyFormatter) with different methods

```
public class PrintMessage {
  public void print(String message) { System.out.println(message); }
}
```

```
import java.text.DecimalFormat;
public class MyFormatter {
  public void formatAmount(String amount) {
    DecimalFormat df = new DecimalFormat("$#,##0.00");
    System.out.println(df.format(Double.parseDouble(amount)));
  }
}
```

Generic Example (cont.)

- Execute the *Sender* class with different classes to invoke their methods

```
public class TestSender {
  public static void main(String args[]) {
    Sender.sendMsg("PrintMessage", "print", "Hello World!");
    Sender.sendMsg("MyFormatter", "formatAmount", "9999");
  }
}
```

- Generic allows classes and methods to be called with arguments of different types

Lambda Expression

- Scala programming language provides elegant functional programming style and abilities
- Java 8 uses lambda expression to reply

```
public class Student {
  private int studentID;
  private String studentName;

  public Student(int id, String name) {
    this.studentID = id;
    this.studentName = name;
  }

  public int getStudentID() {return studentID;}
  public String getStudentName() {return studentName;}
}
```

Sort Student (Inner Class Style)

```
public class SortStudent1 {
  public static void main(String args[]) {
    ArrayList<Student> studentList = new ArrayList<Student>();
    studentList.add(new Student(68, "Mary Anne"));
    studentList.add(new Student(36, "John Smith"));
    studentList.add(new Student(134, "Peter Pan"));
    studentList.sort(new RankStudent());
    for (Student st : studentList) {
      System.out.println(st.getStudentID() + ":" + st.getStudentName());
    }
  }
}
class RankStudent implements Comparator<Student> {
  public int compare(Student s1, Student s2) {
    return s1.getStudentID() - s2.getStudentID();
  }
}
```

Anonymous Class Style

```
public class SortStudent2 {
  public static void main(String args[]) {
    ArrayList<Student> studentList = new ArrayList<Student>();
    studentList.add(new Student(68, "Mary Anne"));
    studentList.add(new Student(36, "John Smith"));
    studentList.add(new Student(134, "Peter Pan"));
    studentList.sort(new Comparator<Student>() {
      public int compare(Student s1, Student s2) {
        return s1.getStudentID() - s2.getStudentID();
      }
    });
    for (Student st : studentList) {
      System.out.println(st.getStudentID() + ":" + st.getStudentName());
    }
  }
}

```

Lambda Expression Style

```
public class SortStudent3 {
  public static void main(String args[]) {
    ArrayList<Student> studentList = new ArrayList<Student>();
    studentList.add(new Student(68, "Mary Anne"));
    studentList.add(new Student(36, "John Smith"));
    studentList.add(new Student(134, "Peter Pan"));
    // JDK version 8 or later
    studentList.sort((Student s1, Student s2) ->
                  s1.getStudentID() – s2.getStudentID());
    studentList.forEach((st) -> System.out.println(st.getStudentID() +
                  ":" + st.getStudentName()));
  }
}
```

Variables in Procedural Programming

- Variable names must be unique in some procedural programming languages
- A single procedural program will commonly hold a thousand of variables and many functions
  - In most cases, they cannot be reused
- Java objects are designed to be reused
  - Some can be opened to public
  - Some can be hidden

Introduction to JavaBean

- A Java program designed to be a reusable object for holding data in memory
  - Transactional data, database table entries
- It has no argument constructor (or no constructor) and no main method
- Information Hiding
  - It encapsulates (hides) many objects (String, Double, Integer, other class instances) into a single object (JavaBean)
- Simple programming style and widely used in software development

An UserBean Class

```
public class UserBean {
  private int userID;

  private String userName, password;

  public UserBean() {
    super();
  }

  public int getUserID() {
    return userID;
  }

  public void setUserID(int userID) {
    this.userID = userID;
  }

```

An UserBean Class (cont.)

```
  public String getUserName() {
    return userName;
  }

  public void setUserName(String userName) {
    this.userName = userName;
  }

  public String getPassword() {
    return password;
  }

  public void setPassword(String password) {
    this.password = password;
  }
}

```

About UserBean

- A class with capitalized name *UserBean*
- It has three properties (variables) defined as private
  - userID, userName, password
- It has a no argument constructor or simply no constructor
- It has mutators (setter methods) to change
  - setUserID(int userID), setUserName(String userName), setPassword(String password)
- It has accessors (getter methods) to read
  - getUserID(), getUserName(), getPassword()

JavaBean Encapsulation

- *Information Hiding* to encapsulate objects inside a JavaBean
  - Doesn’t know the format of the embedded objects in compile time but will know the format in runtime
- One of the fundamental OOP concepts
- Insert dynamic content to a JavaBean
  - Information of a transaction record
  - Entry of a table
  - When data format is uncertain
- More flexible and easy to change when new requirements come

TransBean Example

- Using the same JavaBean for two different transactions for holding different set of data
  - T001: Transfer Money Transaction
  - T002: Withdraw Cash Transaction

```
import java.util.List;

public class TransBean implements Serializable {
  private String txnName;

  private List txnData;

  public TransBean() {
    super();
  }
```

TransBean Example (cont.)

```

  public String getTxnName() {
    return txnName;
  }

  public void setTxnName(String txnName) {
    this.txnName = txnName;
  }

  public List getTxnData() {
    return txnData;
  }

  public void setTxnData(List txnData) {
    this.txnData = txnData;
  }
}
```

TestTrans Example

```
public class TestTrans {
  public static void main(String args[]) {
    TransBean t001 = new TransBean(); // transfer money transaction
    t001.setTxnName("T001");
    String[] data001 = { "T001", "AC:12-3456", "256.00", "AC:12-7890" };
    List<String[]> t001Data = new ArrayList<String[]>();
    t001Data.add(data001);
    t001.setTxnData(t001Data);
    TransBean t002 = new TransBean(); // withdraw cash transaction
    t002.setTxnName("T002");
    List<Object> t002Data = new ArrayList<Object>();
    t002Data.add(t002.getTxnName);
    t002Data.add("AC:12-3456");
    t002Data.add(new Integer(200));
    t002.setTxnData(t002Data);
  }
}
```

Encapsulate another JavaBean

- The *PersonBean* class has a special property call *SalaryBean*
- We have no idea about the properties inside this embedded *SalaryBean*
- We know there is a *SalaryBean* class in compile time but the content will be known only in runtime
- Making changes to the *SalaryBean* will not affect *PersonBean*
- The ability to take the uncertain information away from the main logic is called ***Information Hiding***

PersonBean Example

```
public class PersonBean {
  private String name, gender;

  private SalaryBean salaryBean;

  public PersonBean() {
    super();
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

```

PersonBean Example (cont.)

```

  public String getGender() {
    return gender;
  }

  public void setGender(String gender) {
    this.gender = gender;
  }

  public SalaryBean getSalaryBean() {
    return salaryBean;
  }

  public void setSalaryBean(SalaryBean salaryBean) {
    this.salaryBean = salaryBean;
  }
}
```

Sending a message to an Object

- Class A sends a message to class B by invoking the methods of class B
  - b.isOverDoubleLimit(123.45);
- For example, class B has a method to check if the input amount is over a limit. However, each type of input requires a different method

```
private static final int AMOUT_LIMIT = 10000;
public boolean isOverDoubleLimit(double amount) {
  return (amount > AMOUT_LIMIT);
}
public boolean isOverIntegerLimit(int amount) {
  return (amount > AMOUT_LIMIT);
}
```

Polymorphism Concept

- The ability to call the same method on different objects and have each of them respond in their own way
- Two common polymorphism concepts
  - Overloading
    - Same method name with different parameters
  - Overriding
    - Subclasses use the same method name and parameters to replace the super class’s method

Overloading Example

- A *Calculator* class provides a method called *sum* to process the summation operation
- Programmer wants to provide the diversity for users to input different parameters

```
public class Calculator {
  public static int sum(int a, int b) {
    return a + b;
  }

  public static int sum(int a, int b, int c) {
    return sum(a, b) + c;
  }

```

Calculator Class (cont.)

```
  public static int sum(int num[]) {
    int result = 0;
    for (int i = 0; i < num.length; i++) {
      result += num[i];
    }
    return result;
  }

  // It is suggested to have at least one argument (int a) for using varargs
  public static int sum(int a, int... num) {
    int result = a;
    for (int i = 0; i < num.length; i++) {
      result += num[i];
    }
    return result;
  }
}
```

Testing the Overloading Example

- Now users can input various input parameters for calculating the summation

```
public class TestCalculator {
  public static void main(String args[]) {
    // Calling the method on line 2 of Calculator
    System.out.println(Calculator.sum(1, 2));
    // Calling the method on line 6 of Calculator
    System.out.println(Calculator.sum(1, 2, 3));
    // Calling the method on line 10 of Calculator
    System.out.println(Calculator.sum(new int[] { 1, 2, 3, 4 }));
    // Calling the method on line 18 of Calculator
    System.out.println(Calculator.sum(1, 2, 3, 4, 5));
  }
}
```

Overriding the super class

- We can override the super class *Shape,* with its own method *showInfo()*

```
public abstract class Shape {
  public abstract double getArea();
  public abstract double getPerimeter();

  public String getName() {
    return this.getClass().getSimpleName();
  }

  public void showInfo() {
    System.out.println(getName() + " Information:");
    System.out.println("Area is " + getArea());
    System.out.println("Perimeter is " + getPerimeter());
  }
}
```

A Circle Class

```
public class CircleShape extends Shape {
  private double radius;

  public CircleShape(double radius) {
    this.radius = radius;
  }

  public double getArea() {return radius * radius * Math.PI;}
  public double getPerimeter() {return 2 * radius * Math.PI;}

  @Override
  public void showInfo() {
    super.showInfo();
    System.out.println("Diameter is " + 2 * radius);
  }
}

```

Subtype Polymorphism

- Subtype Polymorphism also called *subtyping*
- A name denotes instances of many different classes related by some common superclass
- Working on an abstract class object but not the concrete class object
- Two Subtype Polymorphism Examples
  - Auto Trader
  - Database Application

Auto Trader

- An auto trader company wants to develop a program to count the capacity (available seats) of its sport cars
- The auto trader company only has three types of sport cars
  - Convertible (2 seats)
  - Minivan (6 seats)
  - Sport Utility Vehicle, SUV (4 seats)
- Someone wrote a single program to solve it

Automobile Class

```
public class Automobile {
  public static List<Object> carList = new ArrayList<Object>();
  public static int getTotalSeats() {
    int totalSeat = 0;
    for (int i = 0; i < carList.size(); i++) {
      if (carList.get(i) instanceof Convertible) {
        totalSeat += 2;
      } else if (carList.get(i) instanceof Minivan) {
        totalSeat += 6;
      } else if (carList.get(i) instanceof SUV) {
        totalSeat += 4;
      } else {
        totalSeat += 0;
      }
    }
    return totalSeat;
  }
```

Automobile Class (cont.)

```
  public static void main(String[] args) {
    Automobile.carList.add(new Convertible());
    Automobile.carList.add(new Minivan());
    Automobile.carList.add(new SUV());
    System.out.println(Automobile.getTotalSeats());
  }
}

class Convertible {
  public String getName() {return "Convertible";}
}
class Minivan {
  public String getName() {return "Minivan";}
}
class SUV {
  public String getName() {return "SUV";}
}
```

Problems

- We should separate the single program to different objects
- The *instanceof* verification part makes the program bulky
- If the auto trader company has a new type of sport car, the program main loop will require to change

Create SportCar type super class

- Create a super class SportCar as a type

```
public abstract class SportCar {
  public abstract int getCapacity();
}
```

- Create three separate subclasses

```
public class ConvertibleSportCar extends SportCar {
  public int getCapacity() {return 2;}
}
```

```
public class MinivanSportCar extends SportCar {
  public int getCapacity() {return 6;}
}
```

```
public class SUVSportCar extends SportCar {
  public int getCapacity() {return 4;}
}
```

AutoTrader Class

```
public class AutoTrader {
  public static List<SportCar> carList = new ArrayList<SportCar>();
  public static int getTotalSeats() {
    int totalSeat = 0;
    for (int i = 0; i < carList.size(); i++) {
      totalSeat += carList.get(i).getCapacity();
    }
    return totalSeat;
  }

  public static void main(String[] args) {
    AutoTrader.carList.add(new ConvertibleSportCar());
    AutoTrader.carList.add(new MinivanSportCar());
    AutoTrader.carList.add(new SUVSportCar());
    System.out.println(AutoTrader.getTotalSeats());
  }
}
```

Subtype Polymorphism Benefits

- Not only the code is shorter and simpler, it is easier to introduce new types of car
- The super class acts like a type rather than only share the commonality between subclasses
- This program uses polymorphism to avoid conditional statements to test whether a value is of a particular type

Database Application

- An application to process four typical database transactions (CRUD) which are mapped to database Data Manipulation Languages (DML and DQL)
  - CRUD: Create, Update, Delete, Read
  - DML: INSERT, UPDATE, DELETE and DQL: QUERY
- However, the four transactions are very similar, so abstraction is needed
- Codes are not flexible, and adding a new transaction requires to change the whole program
- Since it is an example, the logics are not fully implemented

Database Application

```
public class DBApplication {
  public DBApplication() {
    System.out.println("Application started");
  }

  public void create() {
    System.out.println("Log: User calls Create Transaction");
    System.out.println("CREATE A RECORD IN DATABASE");
  }

  public void read() {
    System.out.println("Log: User calls Read Transaction");
    System.out.println("READ A RECORD IN DATABASE");
  }

  public void update() {
    System.out.println("Log: User calls Update Transaction");
```

Database Application (cont.)

```
    System.out.println("UPDATE A RECORD IN DATABASE");
  }

  public void delete() {
    System.out.println("Log: User calls Delete Transaction");
    System.out.println("DELETE A RECORD IN DATABASE");
  }

  public static void main(String[] args) {
    DBApplication app = new DBApplication();
    app.create();
    app.read();
    app.update();
    app.delete();
  }
}

```

The Transaction super class

- This abstract class is the transaction type of this application
- Since each transaction will do a database process, the *doDML* method is promoted to be an *abstract* type methods

```
package app.trans;
public abstract class ESAPTrans {
  public abstract void doDML();

  public void showTransName(String transName) {
    System.out.println("Log: User calls " + transName);
  }
}
```

The four Transactions

- Create Transaction

```
package app.trans;
public class CreateTrans extends ESAPTrans {
  public void doDML() {
    System.out.println("CREATE A RECORD IN DATABASE");
  }
}
```

- Read Transaction

```
package app.trans;
public class ReadTrans extends ESAPTrans {
  public void doDML() {
    System.out.println("READ A RECORD IN DATABASE");
  }
}
```

The four Transactions (cont.)

- Update Transaction

```
package app.trans;
public class UpdateTrans extends ESAPTrans {
  public void doDML() {
    System.out.println("UPDATE A RECORD IN DATABASE");
  }
}
```

- Delete Transaction

```
package app.trans;
public class DeleteTrans extends ESAPTrans {
  public void doDML() {
    System.out.println("DELETE A RECORD IN DATABASE");
  }
}
```

The main Application class

```
package app;
public class Application {
  public static final String CREATE = "CreateTrans";
  public static final String READ = "ReadTrans";
  public static final String UPDATE = "UpdateTrans";
  public static final String DELETE = "DeleteTrans";

  public Application() {
    System.out.println("Application started");
  }

  public void doTrans(String transName) throws Exception {
    ESAPTrans trans = (ESAPTrans) Class.forName("app.trans." +
                      transName).newInstance();
    trans.showTransName(transName);
    trans.doDML();
  }
```

Application class (cont.)

```
  public static void main(String args[]) {
    try {
      Application app = new Application();
      app.doTrans(Application.CREATE);
      app.doTrans("ReadTrans");
      app.doTrans(args[0]); // trigger the CRUD outside the Java class
      app.doTrans("DeleteAllTrans"); // call a new DeleteAllTrans class
    } catch (ClassNotFoundException cnfe) {
      System.err.println("Transaction Not Found: " + cnfe.getMessage());
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Discussion

- The code on *Line-13* to *Line-14* creates one of the CRUD transaction instance dynamically on runtime
- The *doDML* method on *Line-16* uses the *ESAPTrans trans* object to present one of the CRUD transactions to process the database logic
- The *doTrans* method on *Line-22* to *Line-25* doesn’t know what transaction will be called by user on compile time but will know it on runtime
- The usage of *doTrans* method to call the CRUD transactions becomes very dynamic
- Adding new transactions will not need to change and compile the main *Application* program

What is an Interface?

- An interface is a class-like construct that contains only constant variables and abstract methods
- It is similar to an abstract class, but the intent of an interface is to specify common behavior for objects
- A Java subclass can only inherit from one super class that is it can *extends* only one *abstract* class. However, a Java class can *implements* multiple interfaces to have their behaviors
- In general, we will *implement* the methods from an Interface, and *extends* the methods from an abstract class
- It is like a plugin to enhance the functions for a system

HeavyWeaponSystem Example

- An example of making a robot, and we want that robot to install a weapon pack
- Other robots can decide whether or not to install this heavy weapon system
- We can design other interfaces for other weapon systems or defense systems

```
public interface HeavyWeaponSystem {
  public static final int totalWeapons = 3;
  public String getGun();
  public String getShield();
  public String getSword();
}
```

Robot Example

```
public class Robot implements HeavyWeaponSystem {
  public Robot() {
    System.out.println("Create a Robot.");
  }

  public String getName() {return "Golden Warrior";}

  public String getSize() {return "MG";}

  public void showInfo() {
    System.out.println("Robot Name is: " + getName());
    System.out.println("Robot Size is: " + getSize());
    System.out.println("Has " + HeavyWeaponSystem.totalWeapons + " weapons");
    System.out.println("Robot has: " + getGun());
    System.out.println("Robot has: " + getShield());
    System.out.println("Robot has: " + getSword());
 }
```

Robot Example (cont.)

```
  @Override
  public String getGun() {return "Hyper Bazooka";}

  @Override
  public String getShield() {return "Excalibur";}

  @Override
  public String getSword() {return "Beam Saber";}

  public static void main(String args[]) {
    Robot robot = new Robot();
    robot.showInfo();
  }
}
```

Making Abstraction

- When codes are repeated within a class, we can convert them to a ***method***
- When programs are repeated or similar within an application, we can make an abstract class or ***super class***
  - Top-down
  - Bottom-up approaches
- When applications are repeated or similar, we can develop a ***software framework***

Convert to Method

- Calculate the area of a rectangle

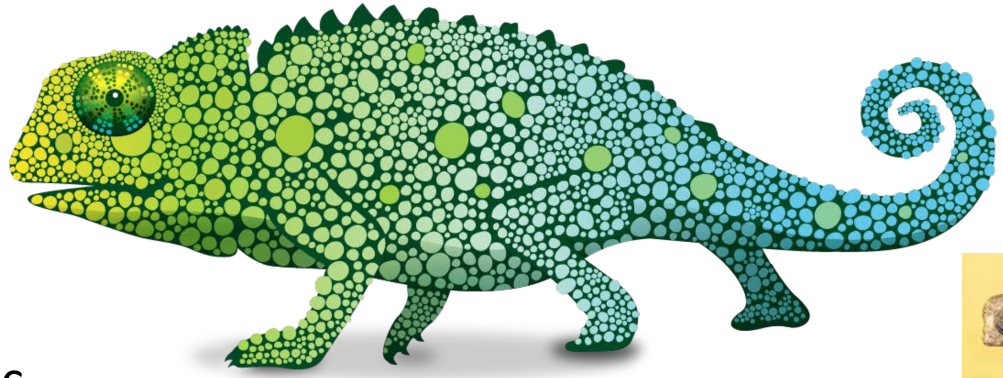
```
Rectangle a = new Rectangle(3, 5);
System.out.println("Area is " + a.getWidth() * a.getHeight());
Rectangle b = new Rectangle(4, 6);
System.out.println("Area is " + b.getWidth() * b.getHeight());
```

- Better to use a method to do it

```
Rectangle a = new Rectangle(3, 5);
calArea(a);
Rectangle b = new Rectangle(4, 6);
calArea(b);
…
public static void calArea(Rectangle r) {
  System.out.println("Area is " + r.getWidth() * r.getHeight());
}
```

Top-down Approach

- Top-down Approach
  - It starts with the big picture and breaks down from there into smaller segments
- Abstract class: *Reptile*
  - swim, legs, toxic
- Concrete classes
  - *Snake*: cannot swim, no leg, toxic
  - *Turtle*: can swim, 4 legs, non-toxic
  - *Chameleon*: cannot swim, 4 legs, non-toxic
- Add more characteristics to the super class for describing other reptile species
  - *Brookesia Micra*: cannot swim, 4 legs, non-toxic, smallest
  - *Meiolania*: can swim, 4 legs, non-toxic, has horn



Reptile Example

- Super Class: *Reptile*

```
public abstract class Reptile {
  public abstract boolean canSwim();
  public abstract int getLegs();
  public abstract boolean isToxic();
  public String getName() {return this.getClass().getSimpleName();}
}
```

- Concrete Class: *Chameleon*

```
public class Chameleon extends Reptile {
  public boolean canSwim() {return false;}
  public int getLegs() {return 4;}
  public boolean isToxic() {return false;}
}
```

Bottom-up Approach

- Bottom-up Approach
  - The individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems
  - It is more common to use in developing enterprise systems
- Concrete classes: CRUD Transaction
  - CreateTrans, ReadTrans, UpdateTrans, DeleteTrans
- Abstract class: *ESAPTrans*
- *ESAPTrans* is not a common things compared to *Reptile*, so we usually make the abstraction based on the similarities of the concrete classes (CRUD)

Software Framework

- A software framework is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications, products and solutions
- It helps us to build software quickly and more standardized
- Software designer create a software framework to abstract the system

How to Build Abstraction?

- There is no unique answer to build abstract classes, you can make it with your own style
- It is about craftsmanship and more like an written essay to have personal style. The bottom line is they all solve the problem
- However, it does follow the ***design principles*** on building abstraction
- Moreover, the ***design patterns*** help us to write programs in pattern which can make the programs to have *repeated codes* for building abstraction

Summary

- A Java object is an instance of a class
- JavaBean uses for information hiding
- Fundamental OOP
  - Polymorphism
  - Encapsulation
  - Implementation
  - Abstraction