Java training

Java OOP concepts

Session overview

Object-oriented programming (OOP) concepts:

a. Abstraction
b. Encapsulation
c. Inheritance
d. Polymorphism

OOP concepts / principles overview

- Providing the support for scalable & flexible architectures
- Summary:
  - **Abstraction**: exposing just the essential details of an entity / service
  - **Encapsulation**: mixing processing and data in a single class
  - **Inheritance**: creating new types of objects from existing ones
  - **Polymorphism**: a method can have different behavior in different contexts

Abstraction

- Designing the representation of programs similar in form to its meaning, while hiding the implementation details
  - Hiding information that is not relevant → expose only relevant information
  - The classes, method names and signature exposes the ***what***, not the ***how***
- Two main forms:
  - *Data abstraction* - creating / modeling data types (using classes)
  - *Processing abstraction* - abstracting the processing (using methods)

Abstraction examples

- Data abstraction:

```
public class Section {
    private String name;
    private Location location;
    private List<Product> products;
}
```

- Processing abstraction:

```
public boolean buy(Product product) {...}

public void register(User user) {...}
```

Encapsulation

- Combining data and processing logic, while hiding it from the external usage
- Implemented via *information* and ***implementation*** hiding
  - Information hiding: using access control keywords:
    - `public, private, protected, final`
  - Implementation hiding: using interfaces and abstract classes
- Especially useful in places where the implementation is likely to change

  "Whatever changes, encapsulate it" – a good design principle / good practice

Encapsulation - examples

- *Information* hiding:

```
public List<Product> processProducts() {...}
```

→ the method returns a `List` of products, but does not specify *how* it is computed

- *Implementation* hiding:

```
public interface UserRepository {}

public class DatabaseRepository implements UserRepository {}

public class FileSystemRepository implements UserRepository {}
```

Inheritance

- Mechanism through which +1 child classes inherit properties of a parent class
  - The resulted classes structure → class hierarchy
- Benefits:
  - Open / closed principle - ‘software entities should be open for extension, but closed for modification’
  - Code reuse
  - Maintainability
  - Clean code structure

Inheritance - characteristics

- Keyword used for creating class hierarchies - `extends`
- Inherited parts - all non private entities:
  - Methods
  - Fields
  - Inner classes
- Terminology:
  - Parent class: superclass (parent class)
  - Extending class(es): subclass (child class)
- Multiple inheritance (extending more than a class) - not allowed

Inheritance - example

- Parent class (super-class):

```
public class Product {
    private String name; // + getter and setter
}
```

→ ***'has a'*** relationship

- Extending class (sub-class):

```
public class Tablet extends Product {
    private String producer;
    private double diagonal; // + getters and setters
}
```

→ ***'is a'*** relationship

Polymorphism

- Polymorphism → taking many forms, based on the context / usage
- Types:
  - **Static** → method *overloading*: using methods with the same name, but with different arguments (order, number or type):

```
public void processProduct(int index);

public void processProduct(Product product);
```

  - **Dynamic** → method *overriding*: dynamically invoking different methods, based on the invoking object type:

```
public void process() // from the Product class

public void process() // from the extending Tablet class
```

Upcasting and downcasting

- **Casting** - converting an object into another type *from the same hierarchy*
- **Up- and down-casting** - converting into a super or sub-type

```
class Product {...}
class Tablet extends Product {...}

Product product = new Product();
Tablet tablet = new Tablet();

Product another = (Product) tablet;        → up-casting
Tablet newTablet = (Tablet) product;       → down-casting
```

Hands-on →

Upcasting and downcasting

- **Up-casting** - casting a subclass to a superclass (upward in the inheritance tree):

```
Tablet tablet = new Tablet();
Product product = (Product) tablet;
```

  - Upcasting is always safe → we treat a type as a more general one

- **Down-casting** - casting a superclass to a subclass (downward in the hierarchy):

```
Product product = new Product();
Tablet tablet = (Tablet) product;
```

  - Downcasting can fail if the actual object type is not of the intended target object type
    → throws a `ClassCastException`

Inheritance & composition - ‘is-a’ & ‘has-a’

- **Inheritance** - classes hierarchies form a ‘***is-a***’ relationship:

```
class Product {...}

class Tablet extends Product {...}
```

→ a `Tablet` *is a* `Product`

- **Composition** - class fields use a ‘***has-a***’ relationship:

```
class Store {
    private Set<Section> sections;
}
class Section {
    private Set<Product> products;
}
```

→ a `Store` ***has*** `Sections`

→ a `Section` ***has*** *a set of* `Products`

‘instanceof’ operator

- ‘instanceof’ operator - test if an object is an instance of a type
  - Class
  - Interface
- Example:

```
class Product {...}

class Tablet extends Product {...}

Tablet tablet = new Tablet();
System.out.println(tablet instanceof Product);
```

→ output: true

Q & A session

1. You ask, I answer
2. I ask, you answer
    a. What is inheritance?
    b. What is polymorphism?
    c. What is encapsulation?