Diseño de Clases

Mejorado

JoeDayz | 2024

Overload Methods

Define more than one version of a method within a given class:

- Two or more methods, within the same class, that have identical names
- They must have a different number, or different types, or different order of parameters.
- Invoker does not need to learn different method names, just set different parameters.

```
public class Product {
  private BigDecimal price;
  private BigDecimal discount = BigDecimal.ZERO;
  public void setPrice(double price) {
    this.price = BigDecimal.valueOf(price);
  }
  public void setPrice(BigDecimal price) {
    this.price = price;
  }
  public void setPrice(BigDecimal price,
                       BigDecimal discount) {
    this.price = price;
    this.discount = discount;
  }
}
```

Methods cannot be overloaded:

- Using different parameter names
- Using different return types

```
Product p = new Product();
p.setPrice(1.99);
p.setPrice(BigDecimal.valueOf(1.99));
p.setPrice(BigDecimal.valueOf(1.99),
           BigDecimal.valueOf(0.9));
```

Variable Number of Arguments

The `vararg` feature enables a variable number of arguments of the same type.

- It avoids creating too many overloaded versions of the same method.
- The `vararg` parameter is treated as an array, with the length constant indicating a number of values.
- An int index (starting at 0) is used to access array elements.
- In case where there are other parameters in the method, the `vararg` parameter must be defined last.
- Using `varargs` is convenient—the invoker simply sets a required number of values.

```
Product p = new Product();
p.setFiscalDetails(1.99);
p.setFiscalDetails(1.99, 0.9, 0.1);
p.setFiscalDetails(1.99, 0.9);
```

✤ **Note:** Arrays are covered later in the course.

```
public class Product {
  private BigDecimal price;
  private BigDecimal discount;
  private BigDecimal tax;
  public void setFiscalDetails(double... values) {
    switch (values.length) {
      case 3:
        tax = BigDecimal.valueOf(values[2]);
      case 2:
        discount = BigDecimal.valueOf(values[1]);
      case 1:
        price = BigDecimal.valueOf(values[0]);
    }
  }
}
```

Defining Constructors

A constructor is a special method that initializes the object:

- It is invoked using the `new` operator.
- It must be named after the class, must not have a return type, or be defined as void.
- Usually public, it may have parameters, and can be overloaded like any other method.
- A default constructor with no parameters is implicitly added to the class, only if no other constructors were added.
- You may explicitly add the `no-arg` constructor, as yet another overloaded version of the constructor, if you want to be able to create objects with or without providing constructor arguments.

```
public class Product {
   private String name;
   private BigDecimal price;
}
```

```
new Product();
```

```
public class Product {
   private String name;
   private BigDecimal price;
   public Product() {
   }
   public Product(String name) {
      this.name = name;
      this.price = BigDecimal.ZERO;
   }
}
```

```
new Product("Water");
new Product();
```

Reusing Constructors

A constructor can invoke another to reuse its logic.

- Invocation of another constructor is performed by using the following syntax:
  `this(<other constructor parameters>);`
- Such a call must be the first line of code in the invoking constructor.
- A cycle (loop) of constructor invocations is not allowed.

```
public class Product {
    private String name;
    private BigDecimal price;
    public Product(String name, double price) {
        this(name);
        this.price = BigDecimal.valueOf(price);
    }
    public Product(String name) {
        this.name = name;
        this.price = BigDecimal.ZERO;
    }
}
```

```
new Product("Water");
new Product("Tea",1.99);
new Product();
```

Access Modifiers: Summary

Access Modifiers control access to classes, methods, and variables:

- `public`: Visible to any other class
- `protected`: Visible to classes that are in the same package and to subclasses
- `<default>`: Visible to classes in the same package only
- `private`: Visible within the same class only

	Any class	Classes in the same package and subclasses	Classes in the same package only	Same class
`public`	✓	✓	✓	✓
`protected`	✗	✓	✓	✓
`<default>`	✗	✗	✓	✓
`private`	✗	✗	✗	✓

Notes:

- `<default>` means that no access modifier is explicitly set.
- The subclass-superclass relationship (use of the extends keyword) is covered later in the course.

Defining Encapsulation

Information contained within the object should normally be "hidden" inside it.

- Instance variables are typically `private`, so that they are not visible outside of this class.
- To access this information from other classes, you may provide methods with less restrictive access modifiers: `public`, `<default>`, `protected`.
- This allows you to control access to information, validate data, or modify data format.
- There are also some `private` methods that can be invoked only from other methods of the same class.

```
public class Product {
  private BigDecimal price;
  private BigDecimal tax;
  private BigDecimal rate = BigDecimal.valueOf(0.1);
  public void setPrice(BigDecimal price) {
    this.price = price;
    calculateTax();
  }
  public BigDecimal getPrice() {
    return price.add(tax);
  }
  private void calculateTax() {
    tax = price.multiply(rate);
  }
}
```

- **Note:** Attempting to access a "hidden" variable or method from another class will result in a compilation error.

```
public class Shop {
  public static void main(String[] args) {
    Product p1 = new Product();
    p1.setPrice(BigDecimal.valueOf(1.99));
    BigDecimal total = p1.getPrice();
    p1.tax = 3.20;
    p1.calculateTax();
  }
}
```

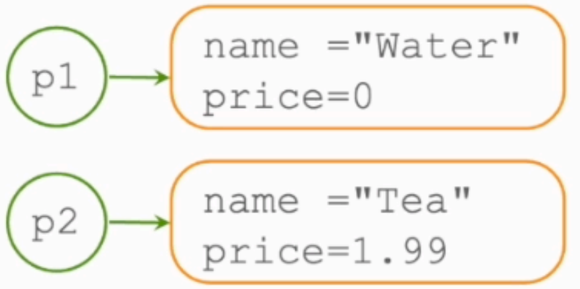
Defining Immutability

Immutable objects present read-only data that is not modifiable after object construction.

- Instance variables must be encapsulated (`private`) to prevent direct access.
- Instance variables are initialized immediately or via constructors.
- No setter methods are provided.
- Immutable objects are thread-safe, without the overhead cost of coordinating synchronized access.
- Many JDK classes are designed this way: primitive wrappers, local date and time, string, and so on.

```
public class Product {
  private String name;
  private BigDecimal price = BigDecimal.ZERO;
  public Product(String name) {
    this.name = name;
  }
  public Product(String name, BigDecimal price) {
    this(name);
    this.price = price;
  }
  public String getName() {
    return name;
  }
  public BigDecimal getPrice() {
    return price;
  }
}
```

```
public class Shop{
  public static void main(String[] args) {
    Product p1 = new Product("Water");
    String name = p1.getName();
    BigDecimal price = p1.getPrice();
    price = BigDecimal.valueOf(1.99);
    Product p2 = new Product("Tea", price);
  }
}
```



Constants and Immutability

Constants represent data that is assigned once and cannot be changed.

- The `final` keyword is used to mark a variable as a constant. Once it is initialized, it cannot be changed.
- Instance `final` variables must be either initialized immediately or using instance initializer or all constructors.

```
public class Product {
  private static int maxId = 0;
  private final int id;
  private final String name;
  private final BigDecimal price;
  { id = ++maxId; }
  public Product(String name) {
    this.name = name;
    this.price = BigDecimal.ZERO;
  }
  public Product(String name, BigDecimal price) {
    this.name = name;
    this.price = price;
  }
  public BigDecimal getDiscount(final BigDecimal discount) {
    return price.multiply(discount);
  }
}
```

- **Note:** An instance initializer is a block of code that is triggered before the invocation of the constructor.

ENUMS

Enumerations

Enumeration (`enum`) provides a fixed set of instances of a specific type.

- Enum values are instances of this enum type. (`HOT`, `WARM`, and `COLD` are instances of the `Condition`.)
- Enum values are implicitly public, static, and final.
- Enums can be used as:
  - Variable types
  - Cases in switch constructs

```
public enum Condition {
  HOT, WARM, COLD;
}
```

```
import static Condition.*;
public class Shop {
  public static void main(String[] args) {
    Product tea = new Product("Tea",HOT);
    Person joe = new Person("Joe");
    joe.consume(tea.serve());
  }
}
```

```
public class Product {
  private String name;
  private String caution;
  private Condition condition;
  public Product(String name,
                 Condition condition){
    this.name = name;
    this.condition = condition;
  }
  public Product serve() {
    switch(condition) {
      case Condition.COLD:
        this.addCaution("Warning COLD!");
        break;
      case Condition.WARM:
        this.addCaution("Just right");
        break;
      case Condition.HOT:
        this.addCaution("Warning HOT!");
    }
    return this;
  }
}
```

Java Memory Allocation

Java has the following memory contexts: Stack and heap.

- Stack is a memory context of a thread, storing local method variables.
- Heap is a shared memory area, accessible from different methods and thread contexts.
- Stack can hold only primitives and object references.
- Classes and objects are stored in the heap.


```
public class Shop {
  public static void main(String[] args) {
    int x = 10;
    Product p = new Product();
  }
}
```

```
public class Product {
  private int id;
  private String name;
  private BigDecimal price;
  private LocalDate bestBefore = LocalDate.of(2019,1,21);
}
```

✣ **Note:** Creation of a new object does not necessarily initialize all of its values.

Parameter Passing

Just like any other local method variables, parameters are stored in a stack.

Passing parameters means copying stack values:

- A copy of an object reference value
- A copy of a primitive value

```
public void manageProduct() {
  Product p1 = new Product();
  orderProduct(p1);
  double v1 = p1.getPrice();
  changePrice(v1);
}
public void orderProduct(Product p2){
  p2.setPrice(1.99);
}
public void changePrice(double v2){
  v2 = 2.99;
}
```

- The p2 parameter is initialized by copying the value of the p1 variable, which is a reference pointing to the same object in the heap, making this object accessible to both methods.
- The v2 parameter is initialized by copying the value of the v1 variable. Modification of the v2 variable has no effect on the v1 variable.

Java Memory Cleanup

Objects remain in the heap as long as they are still referenced.

- An object reference is `null` until it is initialized.
- Assigning `null` does not destroy the object; it just indicates the absence of a reference.
- When a method returns, its local variables go out of scope and their values are destroyed.

```
public void manageProduct() {
  int x = 10;
  Product p1;
  p1 = new Product();
  orderProduct(p1);
  p1 = null;
}
public void orderProduct(Product p2) {
  // continue to use p2 reference
}
```

Garbage collection is a background process that cleans unused memory within the Java run time.

- Garbage collection is deferred.
- An object becomes eligible for garbage collection when there are no references pointing to it.

Quiz

1. Which two are rules for method overloading?

- ☐ Must have different parameter names
- ☐ Must not be void
- ☐ Must have the same return type
- ☐ Must have a different number or types of parameters, or both
- ☐ Must be in different classes
- ☐ Must not have identical names

C, D

1. Which two are characteristics of a constructor method?

- [ ] Must be named after the class
- [ ] Must be void
- [ ] Is a special method that initializes the object
- [ ] Must have one or more parameters
- [ ] Must not have an access modifier

A, C

2. Which two statements are true?

- [ ] Objects are allocated in the heap.
- [ ] Each thread has its own stack.
- [ ] Primitives cannot be placed into a heap.
- [ ] An object is eligible for garbage collection when it is null.
- [ ] A primitive is eligible for garbage collection when it is 0.

A B

3. What is the concept of enclosing an object's internal information called?

- ○ Encapsulation
- ○ Shadowing
- ○ Abstraction
- ○ Polymorphism
- ○ Inheritance

A

5. Which statement is false?

- ○ Enum provides a fixed set of instances of a specific type.
- ○ Enum values are instances of this enum type.
- ○ Enum cannot be used in the switch constructs.
- ○ Enum cannot be used as variable value.
- ○ Enum cannot be used as variable type.
- ○ Enum values are implicitly protected.

C