Clases - Java 21

JoeDayz | 2025

Objectives

After completing this lesson, you should be able to:

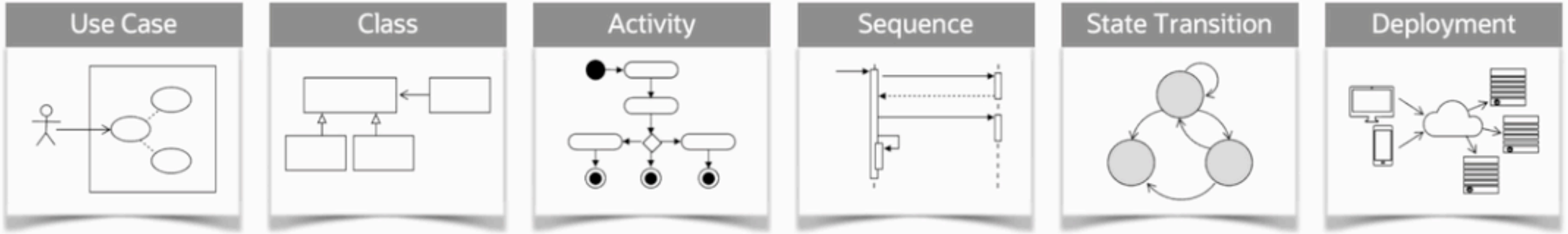
- Model business problems by using classes
- Define instance variables and methods
- Describe the `this` object reference
- Explain object instantiation
- Explain local variables and local variable type inference
- Define static variables and methods
- Invoke methods and access variables
- Describe IntelliJ IDE

UML: Introduction

Unified Modelling Language (UML) diagrams:

- Are used to graphically represent business requirements and design software
- Facilitate communications between analysts, designers, and developers
- Clearly and concisely represent analyses, design, and implementation requirements
- Are eventually implemented in actual Java code

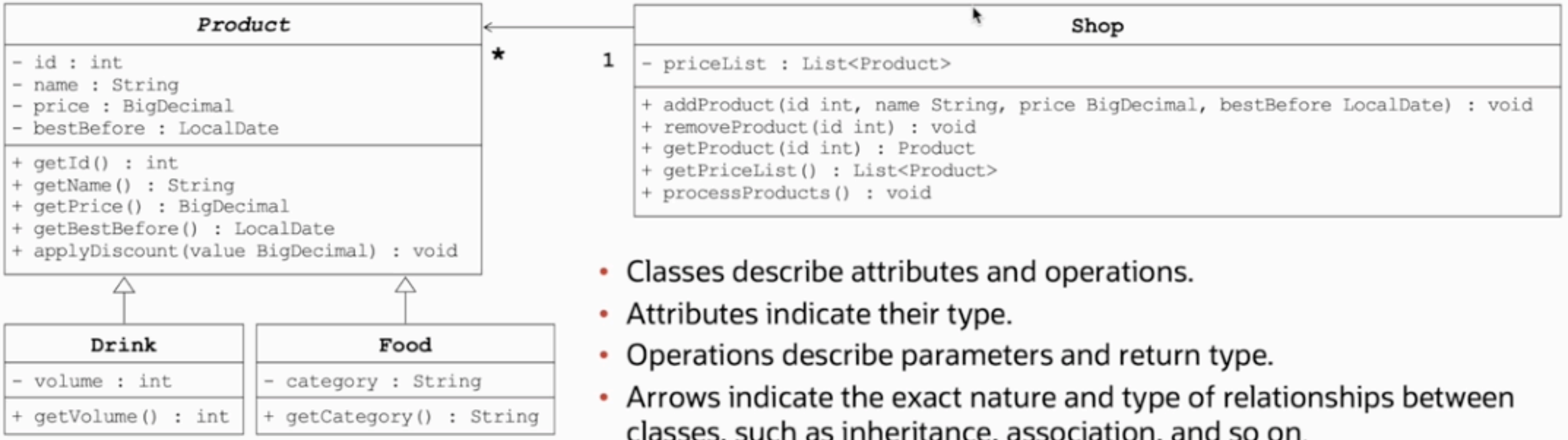
Examples of UML diagrams:



- This course provides only a very brief introduction to modelling. For more information, see the *Object-Oriented Analysis and Design Using UML* course.

Modeling Classes

This example shows a Class diagram, which represents classes and their relationships.

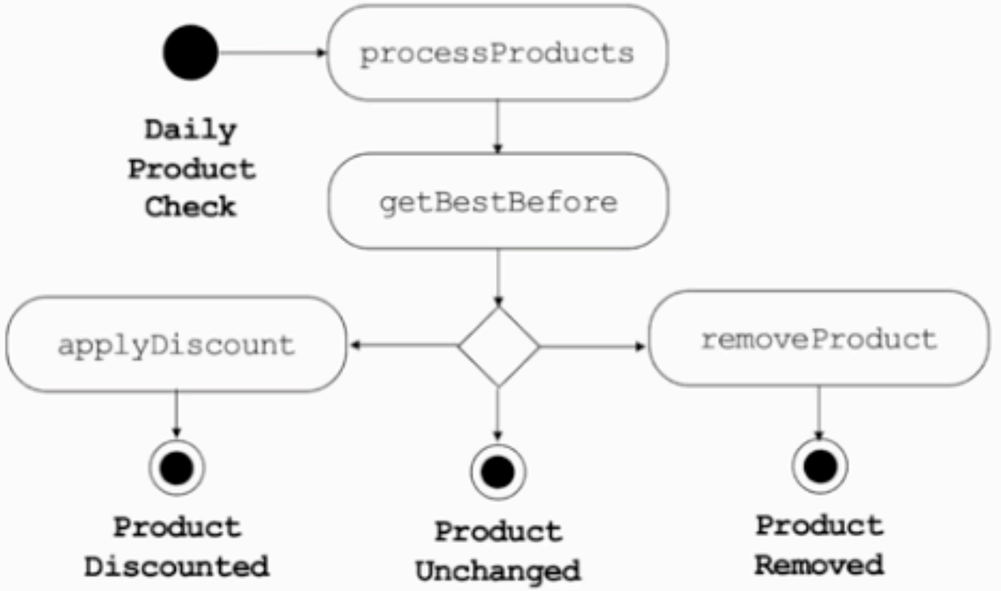


- Classes describe attributes and operations.
- Attributes indicate their type.
- Operations describe parameters and return type.
- Arrows indicate the exact nature and type of relationships between classes, such as inheritance, association, and so on.

- The example provides a model of the set of classes required to implement a product management system for a shop (see detailed description in notes).
- Also see notes for naming convention details.

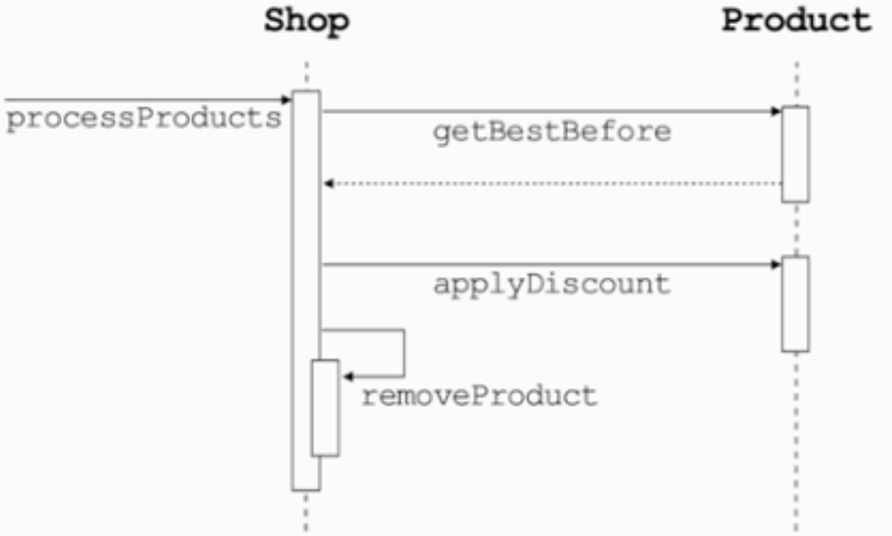
Modeling Interactions and Activities

An Activity diagram represents a flow of operations.



- Flow of activities is trigged by an event.
- Flow of activities may end in a number of outcomes.
- Steps in a flow describe program activities and decisions.
- Lines represent the order of activities.

A Sequence diagram represents interactions between program objects.



- Each object is represented by its lifeline.
- Interactions between objects may include sending messages and returning values.
- Objects can invoke operations (send messages) upon other objects or recursively upon themselves.

Designing Classes

Java class definition typically includes:

- Description of the package that this class is a member of
- Description of imports of classes from different packages that this class may need to reference
- This class access modifier (typically public)
- Keyword class followed by this class name
- Class and method bodies are enclosed with "{" and "}" symbols.
- A number of variable and method definitions within the class body

```
package <package name>;
import <package name>.<OtherClassName>;
<access modifier> class <ClassName> {
  // variables and methods
}
```

```
package demos.shop;
import java.math.BigDecimal;
public class Product {
  private BigDecimal price;
  public BigDecimal getPrice() {
    return price;
  }
  public void setPrice(double value) {
    price = BigDecimal.valueOf(value);
  }
}
```

Creating Objects

Java objects are instances of classes.

- The `new` operator creates an object (an instance of a class), allocating memory to store this object.
- Assign "reference" to the memory allocated for the object to be able to access it.
- Access a variable or methods of the object by using the " . " operator.

```
Product p1 = new Product();
p1.setPrice(1.99);
BigDecimal price = p1.getPrice();
```

Note: A reference is a typed variable that points to an object in memory.

```
package demos.shop;
import java.math.BigDecimal;
public class Product {
  private BigDecimal price;
  public BigDecimal getPrice() {
    return price;
  }
  public void setPrice(double value) {
    price = BigDecimal.valueOf(value);
  }
}
```

Defining Instance Variables

Classes may contain variable definitions to store state information about their instances (objects).

- Variable is defined with its type, which can be one of the eight primitive types or any class.
- Variable name is typically a noun written in lowercase.
- To protect data within the class, variables typically use private access modifier.
- Optionally, variables can be initialized (assigned a default value).
- Uninitialized primitives are defaulted to `0`, except `boolean`, which defaults to `false`.
- Uninitialized object references are defaulted to `null`.

```
package <package name>;
import <package name>.<ClassName>;
<access modifier> class <ClassName> {
  <access modifier> <variable type> <variable name> = <variable value>;
}
```

```
package demos.shop;
import java.mathBigDecimal;
import java.time.LocalDate;
public class Product {
  private int id;
  private String name;
  private BigDecimal price;
  private LocalDate bestBefore = LocalDate.now().plusDays(3);
}
```

Defining Instance Methods

Classes may contain method definitions to implement behaviors of their instances (objects).

- Method must declare its return type or use the `void` keyword if a method does not need to return a value.
- Nonvoid methods must contain a return statement, which must return a value of the corresponding type.
- Method name is typically a verb (like get or set) written in lowercase, followed by descriptive nouns.
- Access modifier determines from where the method can be invoked.
- Methods may describe a comma-separated list of parameters enclosed in "`()`" symbols.
- Method body is enclosed with "`{`" and "`}`" symbols.

```
package <package name>;
<access modifier> class <ClassName> {
  <access modifier> <return type> <method name>(<Type> <name>, <Type> <name>) {
    return <value>;
  }
}
```

Notes:

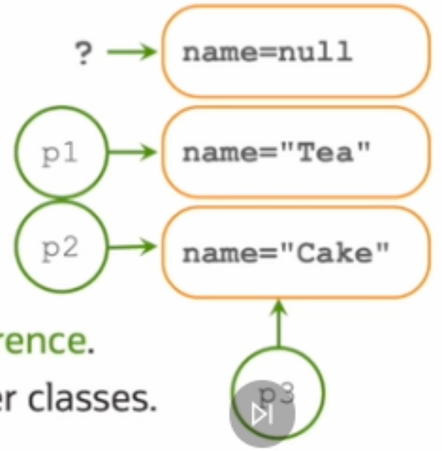
- A return statement terminates the method.
- A void method may be terminated using empty return.
- If no parameters are required, just put empty "`()`" brackets.

```
package demos.shop;
public class Product {
   private String name;
   public String getName() {
      return name;
   }
   public void setName(String newName) {
      name = newName;
   }
}
```

Object Creation and Access: Example

Object is an instance of a class.

- The **new** operator creates an object, allocating memory for it.
- You may assign an object reference to a variable of the appropriate type.
- You can assign the same object reference to more than one variable.
- Invoke operations or access variables of the object by using the " . " operator, via object reference.
- Access modifiers may restrict the ability to access classes, variables, and methods from other classes.




```
package demos.shop;
public class Shop {
  public static void main(String[] args) {
    new Product();
    Product p1 = new Product();
    Product p2 = new Product();
    Product p3 = p2;
    p1.setName("Tea");
    p2.setName("Cake");
    System.out.println(p1.getName()+" in a cup");
    System.out.println(p2.getName()+" on a plate");
    System.out.println(p3.getName()+" to share");
    p1.name = "Coffee";
  }
}
```

```
package demos.shop;
public class Product {
  private String name;
  public void setName(String newName) {
    name = newName;
  }
  public String getName() {
    return name;
  }
}
```

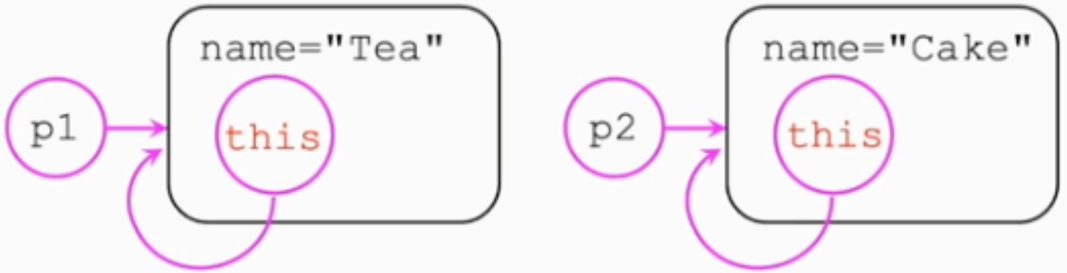
```
>java demos.shop.Shop
Tea in a cup
Cake on a plate
Cake to share
```

Local Variables and Recursive Object Reference

Local variables are declared inside methods.

- They cannot have an access modifier. They are not visible outside of the method anyway.
- Method parameters are essentially local variables.
- A local variable can "shadow" an instance variable if their names coincide.
- Use the **this** keyword (recursive reference to current object) to refer to an instance, rather than a local variable.
- Variables defined in inner code blocks (delimited by "{ }" symbols) are not visible outside of these blocks.

```
Product p1 = new Product();
Product p2 = new Product();
p1.setName("Tea");
p2.setName("Cake");
```



✣ **Note:** In the example, variables `p1` and `p2` are referencing different products, while `this` is referencing the current one.

```
public class Product {
  private String name;
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    if (name == null) {
      String dummy = "Unknown";
      return dummy;
    }
    return name;
  }
  public String consume() {
    String feedback = "Good!";
    return feedback;
  }
}
```

Local Variable Type Inference

There is no need to describe a variable type if it can be unambiguously inferred from the context.

- Infer types of local variables with initializers.
- No need to explicitly declare local variable type if it can be inferred from the assigned value.
- Type cannot be reassigned and polymorphism is not supported.
- This feature is limited to:
  - Local variables with initializers
  - Indexes in the enhanced for-loops
  - Local variables declared in a traditional for-loops
- Overuse can reduce code readability.

```
public void someOperation(int param) {
  var value1 = "Hello"; // infers String
  var value2 = param;   // infers int
}
```

Note: Polymorphism and loops are covered later in this course.

Defining Constants

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

- The keyword `final` is used to mark a variable as a **constant**; once it is initialized, it cannot be changed.
- Instance constants must be either initialized immediately or via all constructors.
- Local variables and parameters can also be marked as final.
- An attempt to reassign a constant will result in a compiler error.

```
public class Product {
  private final String name = "Tea";
  private final BigDecimal price = BigDecimal.ZERO;
  public BigDecimal getDiscount(final BigDecimal discount) {
    return price.multiply(discount);
  }
}
```

```
public class Shop {
  public static void main(String[] args) {
    Product p = new Product();
    BigDecimal percentage = BigDecimal.valueOf(0.2);
    final BigDecimal amount = p.getDiscount(percentage);
  }
}
```

- **Note:** In this example, product values are hardcoded. However, you can also dynamically assign them using a constructor, which is covered later.

Static Context

Each class has its own memory context.

- Class memory context (also known as static context) is shared by all instances of this class.
- The keyword `static` is used to mark variables or methods that belong to the class context.
- Objects can access shared static context.
- Current instance (this) is meaningless within the static context.
- Attempt to access current instance methods or variables from the static context will result in a compiler error.

```
public class Product {
  private static Period defaultExpiryPeriod = Period.ofDays(3);
  private String name;
  private BigDecimal price;
  private LocalDate bestBefore;
  public static void setDefaultExpiryPeriod(int days){
    defaultExpiryPeriod = Period.ofDays(days);
    String name = this.name;
  }
}
```

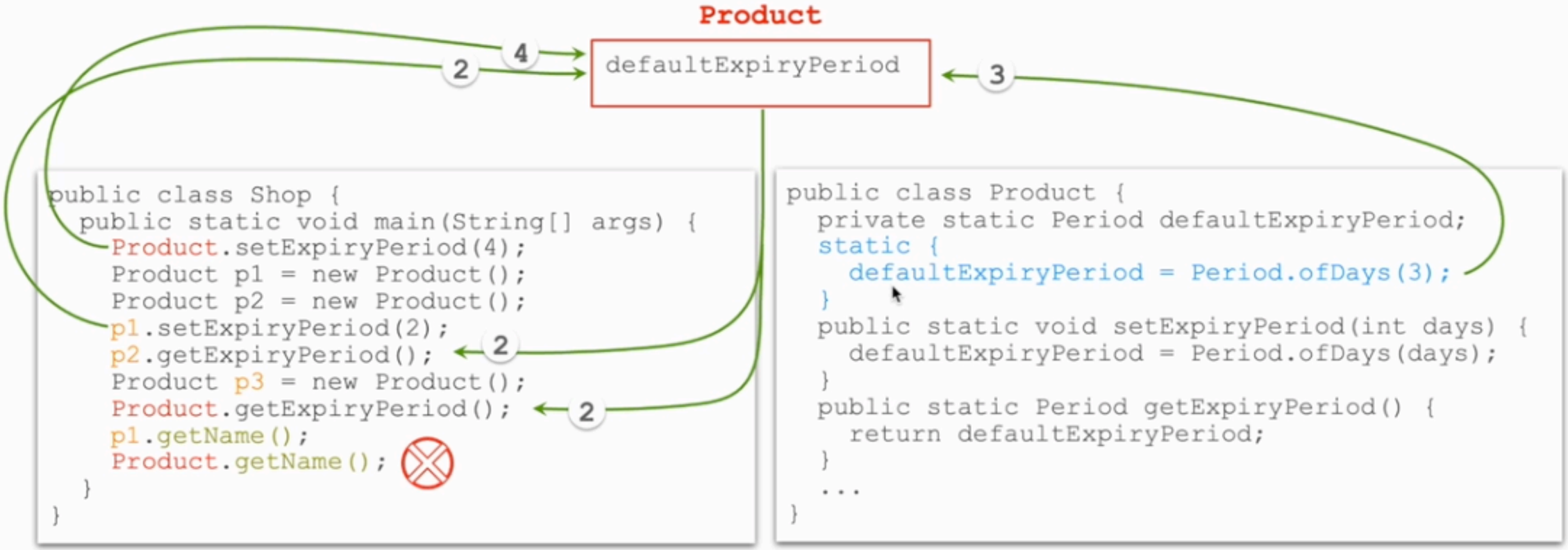
```
Product p1 = new Product();
Product p2 = new Product();
```

✣ **Reminder:** Java classes are types and objects are their instances.

Accessing Static Context

Rules for accessing static context:

- Object reference is not required (but can be used) to access static context.
- Static initializer runs once, before any other operation (when class is loaded).
- Instance variables and methods are not accessible through the static context.



Combining Static and Final

Shared constants can be defined as static and final variables.

- It provides a simple way of defining globally visible constants.
- Encapsulation (private access modifier) is not required because the value is read-only.

```
public class Product {
  public static final int MAX_EXPIRY_PERIOD = 5;
  ...
}
```

```
public class Shop {
  public static void main(String[] args) {
    Period expiry = Period.days(Product.MAX_EXPIRY_PERIOD);
    ...
  }
}
```

Other Static Context Use Cases

Examples of static methods and variables encountered earlier in this course:

- The `main` method is static.
- All operations of the `Math` class are static.
- Factory methods are static methods that create and return a new instance.
- The `System.out` is a static variable, which refers to the `PrintStream` object for standard output.

```
import static Math.random;
public class Shop {
  public static void main(String[] args) {
    Math.round(1.99);
    double value = random();
    BigDecimal.valueOf(1.99);
    LocalDateTime.now();
    ZoneId.of("Europe/London");
    ResourceBundle.getBundle("messages",Locale.UK);
    NumberFormat.getCurrencyInstance(Locale.UK);
    System.out.println("Hello World");
  }
}
```

- **Note:** Static import enables referencing static variables and methods of another class as if they are in this class.

```
public class BigDecimal {
  public static BigDecimal valueOf(double val) {
    return new BigDecimal(Double.toString(val));
  }
...
}
```

IntelliJ IDE: Introduction

IDE Windows:

1. Project
2. Editor
3. Structure
4. Edit Configurations
5. Output

Toolbar Actions:

- Build Project
- Run
- Debug
- Stop