Programowanie obiektowe w Javie - teoria 5

Klonowanie obiektów

Klonowanie obiektów to proces tworzenia dokładnej kopii istniejącego obiektu. W kontekście programowania obiektowego, gdzie obiekt jest instancją klasy, klonowanie obiektu oznacza tworzenie nowego obiektu, który ma te same wartości dla wszystkich swoich pól, co obiekt oryginalny.

Klonowanie obiektów jest stosowane w różnych sytuacjach, takich jak:

1. **Niezmienność oryginalnego obiektu**: Jeżeli chcemy dokonać pewnych operacji na obiekcie, ale nie chcemy wpływać na oryginalny obiekt, możemy sklonować ten obiekt i pracować na jego kopii.
2. **Optymalizacja wydajności**: W niektórych przypadkach, tworzenie nowego obiektu przez sklonowanie istniejącego obiektu może być szybsze niż tworzenie obiektu od zera, zwłaszcza gdy inicjalizacja obiektu wymaga kosztownych operacji, takich jak zapytania do bazy danych.
3. **Wzorzec prototypu**: Klonowanie obiektów jest kluczowym elementem wzorca projektowego prototypu, który polega na kopiowaniu istniejących obiektów zamiast tworzenia nowych.

Jednak klonowanie obiektów nie jest procesem prostym, a metody klonowania mogą mieć różne skutki w zależności od struktury obiektu. W języku Java istnieją dwa główne typy klonowania: płytkie (shallow) i głębokie (deep).

Przegląd różnych metod

1. **Implementacja interfejsu Cloneable i użycie metody `clone()`**: Ta metoda jest najbardziej standardowym podejściem do klonowania obiektów w Javie. Interfejs `Cloneable` jest pusty i służy jako wskaźnik dla JVM, że klasa może być klonowana. Następnie musisz nadpisać metodę `clone()` z klasy `Object` w swojej klasie. Pamiętaj jednak, że ta metoda domyślnie wykonuje klonowanie płytkie.

2. **Klonowanie głębokie poprzez serializację**: Klonowanie głębokie można osiągnąć poprzez serializację obiektu do strumienia bajtów, a następnie deserializację strumienia bajtów z powrotem do obiektu. Ta metoda jest dosyć kosztowna pod względem wydajności, ale umożliwia dokładne skopiowanie obiektu wraz z jego zagnieżdżonymi obiektami.
3. **Klonowanie głębokie poprzez kopiowanie pola po polu**: Inny sposób na wykonanie klonowania głębokiego polega na ręcznym kopiowaniu każdego pola z oryginalnego obiektu do nowego obiektu. To podejście może być czasochłonne i łatwe do pomyłki, ale daje pełną kontrolę nad procesem klonowania.
4. **Użycie bibliotek zewnętrznych**: Istnieją biblioteki, takie jak Apache Commons Lang i Google Guava, które oferują metody do klonowania obiektów. Te biblioteki mogą zaoferować różne strategie klonowania i mogą być bardziej elastyczne niż wbudowane metody Javy.
5. **Konstruktor kopiujący**: To jest inny sposób tworzenia duplikatu obiektu. W tym przypadku tworzymy nowy konstruktor w klasie, który przyjmuje istniejący obiekt tej samej klasy jako argument i kopiujemy wartości z obiektu do nowego obiektu.

Każda z tych metod ma swoje zalety i wady, a wybór między nimi zależy od konkretnych wymagań i kontekstu.

Czemu operator przypisania nie działa?

Oto przykład prostej klasy `Person` w Javie, który pokazuje, dlaczego sam operator przypisania nie jest wystarczający do kopiowania obiektów:

```
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

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

    public String getName() {
        return name;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        // Tworzymy obiekt klasy Person
        Person person1 = new Person("Jan");
        // Próbujemy skopiować obiekt za pomocą operatora przypisania
        Person person2 = person1;

        // Zmieniamy imię osoby 1
        person1.setName("Kasia");

        // Wyświetlamy imię osoby 2
        System.out.println(person2.getName());  // Wyświetli "Kasia", a nie "Jan"
    }
}
```

Kopiowanie płytkie i głębokie w Java

Klonowanie płytkie polega na tworzeniu nowego obiektu i skopiowaniu do niego bezpośrednich wartości z oryginalnego obiektu. Jeżeli oryginalny obiekt zawiera odniesienia do innych obiektów, tylko referencje są kopiowane. Oznacza to, że oryginalny obiekt i jego klon będą wskazywać na te same obiekty, do których prowadzą te referencje.

```java
class ShallowExample implements Cloneable {
    int[] nums = {1, 2, 3, 4, 5};

    @Override
    public ShallowExample clone() {
        try {
            return (ShallowExample) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Klonowanie głębokie to proces tworzenia nowego obiektu i rekursywnego skopiowania wszystkich obiektów, na które wskazują pola oryginalnego obiektu. W rezultacie, klon nie zawiera żadnych odniesień do tych samych obiektów, co oryginalny obiekt.

```
class DeepExample implements Cloneable {
    int[] nums = {1, 2, 3, 4, 5};

    @Override
    public DeepExample clone() {
        try {
            DeepExample result = (DeepExample) super.clone();
            result.nums = nums.clone();  // głębokie klonowanie tablicy
            return result;
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Sprawdzanie

W Javie możemy używać operatora `==` do porównania dwóch referencji do obiektów. Jeżeli dwie referencje wskazują na ten sam obiekt, operator `==` zwróci `true`. Możemy użyć tego operatora do sprawdzenia, czy skopiowany obiekt ma odniesienia do tych samych obiektów co oryginalny obiekt.

Oto przykład:

```
class SomeClass {
    // jakaś zmienna
}

class Test {
    SomeClass someClassField;

    // konstruktor, gettery, settery
}

public class Main {
    public static void main(String[] args) {
        Test original = new Test();
        original.setSomeClassField(new SomeClass());

        Test copy = new Test();
        copy.setSomeClassField(original.getSomeClassField());
```

```
        System.out.println(original.getSomeClassField() == copy.getSomeClassField());
        // true
    }
}
```

W powyższym kodzie, `original` i `copy` mają `someClassField`, które wskazują na ten sam obiekt `SomeClass`. Dlatego porównanie `original.getSomeClassField() == copy.getSomeClassField()` zwraca `true`.

Jednak pamiętaj, że ta metoda jest użyteczna tylko do sprawdzenia, czy dwa obiekty mają odniesienia do tych samych obiektów. Nie sprawdzi ona, czy dwa obiekty są strukturalnie identyczne. Do porównania strukturalnego obiektów można użyć metody `equals()`, ale musi być ona odpowiednio zaimplementowana w klasie obiektu.

Zasady implementacji interfejsu `Cloneable`

Implementacja interfejsu `Cloneable` w Javie wymaga przestrzegania pewnych zasad i dobrych praktyk:

1. **Interfejs `Cloneable`**: Pierwszym krokiem jest oznaczenie klasy jako `Cloneable`, implementując ten interfejs. Jest to interfejs znacznikowy, co oznacza, że nie ma żadnych metod do zaimplementowania.

   ```
   public class MyClass implements Cloneable {
       //...
   }
   ```

2. **Nadpisanie metody `clone()`**: Następnie trzeba nadpisać metodę `clone()` z klasy `Object`. Metoda `clone()` jest chroniona, co oznacza, że jest dostępna tylko w obrębie pakietu i dla klas pochodnych. Powinieneś zadeklarować ją jako `public`, aby była dostępna z zewnątrz.

   ```
   @Override
   public Object clone() throws CloneNotSupportedException {
       return super.clone();
   }
   ```

3. **Obsługa wyjątku `CloneNotSupportedException`**: Metoda `clone()` może rzucić wyjątek `CloneNotSupportedException`, jeśli klasa nie implementuje interfejsu `Cloneable`. Powinieneś obsłużyć ten wyjątek w metodzie `clone()`.

4. **Prawidłowe klonowanie pól**: Domyślna metoda `clone()` wykonuje klonowanie płytkie, co oznacza, że dla pól będą skopiowane tylko wartości. Jeśli masz pola, które są

referencjami do innych obiektów, musisz zaimplementować klonowanie głębokie dla tych pól.

5. **Typ zwrotny**: Zamiast zwracania `Object`, dobrą praktyką jest zwrócenie faktycznego typu klasy. To eliminuje konieczność rzutowania typów po wywołaniu metody `clone()`.

```
@Override
public MyClass clone() throws CloneNotSupportedException {
    return (MyClass) super.clone();
}
```

6. **Finalne pola**: Jeśli klasa ma finalne pola, klonowanie może być trudne, ponieważ wartości finalnych pól nie można zmienić po inicjalizacji. W takim przypadku powinieneś rozważyć użycie innego podejścia do klonowania, takiego jak konstruktor kopiujący.

Przykłady:

```
public class Main {

    public static void main(String[] args) throws CloneNotSupportedException {
        Book book = new Book("Pan Tadeusz", "Adam Mickiewicz");
        Book bookClone = (Book) book.clone();
        bookClone.setTitle("Dziady");
        System.out.println(book.getTitle());
        System.out.println(bookClone.getTitle());
    }
}

class Book implements Cloneable {


    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getTitle() {
```

```
        return title;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public String getAuthor() {
        return author;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    private String title;
    private String author;
}
```

```
public class Main {

    public static void main(String[] args) throws CloneNotSupportedException {
        Book book = new Book("Wiedźmin", new Author("Andrzej Sapkowski"));
        Book bookClone = book.clone();
        System.out.println(book.getAuthor().getName());
        System.out.println(bookClone.getAuthor().getName());
        bookClone.getAuthor().setName("Jan Kowalski");
        System.out.println(book.getAuthor().getName());
        System.out.println(bookClone.getAuthor().getName());

    }
}

class Author implements Cloneable
{
    private String name;

    public Author(String name)
```

```
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

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

    public Author clone() throws CloneNotSupportedException
    {
        return (Author) super.clone();
    }
}

class Book implements Cloneable
{
    private String title;
    private Author author;

    public Book(String title, Author author)
    {
        this.title = title;
        this.author = author;
    }

    public String getTitle()
    {
        return title;
    }

    public Author getAuthor()
    {
        return author;
    }

    public Book clone() throws CloneNotSupportedException
```

```
    {
        Book cloned = (Book) super.clone();
        cloned.author = cloned.author.clone();
        return cloned;
    }
}
```

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {
        Library library = new Library();
        library.addBook(new Book("Wiedźmin", new Author("Andrzej Sapkowski")));
        library.addBook(new Book("Władca Pierścieni", new Author("J.R.R. Tolkien")));
        library.addBook(new Book("Harry Potter", new Author("J.K. Rowling")));
        library.removeBook(library.getBook(0));
        System.out.println(library);
        library.getBook(1).getAuthor().setName("Jan Kowalski");
        System.out.println(library);
    }
}


class Author
{
    public Author(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

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

```
    }

    @Override
    public String toString()
    {
        return "Author{" + "name=" + name + '}';
    }

    @Override
    public Author clone() throws CloneNotSupportedException
    {
        return (Author) super.clone();
    }

    private String name;
}


class Book
{
    public Book(String title, Author author)
    {
        this.title = title;
        this.author = author;
    }

    public String getTitle()
    {
        return title;
    }

    public Author getAuthor()
    {
        return author;
    }

    public void setTitle(String title)
    {
        this.title = title;
    }
```

```
    public void setAuthor(Author author)
    {
        this.author = author;
    }

    @Override
    public String toString()
    {
        return "Book{" + "title=" + title + ", author=" + author + '}';
    }

    @Override
    public Book clone() throws CloneNotSupportedException
    {
        Book cloned = (Book) super.clone();
        cloned.author = this.author.clone();
        return cloned;
    }

    private String title;
    private Author author;
}

class Library
{
    public Library()
    {
        this.books = new ArrayList<>();
    }

    public void addBook(Book book)
    {
        books.add(book);
    }

    public void removeBook(Book book)
    {
        books.remove(book);
    }

    @Override
```

```
    public String toString()
    {
        return "Library{" + "books=" + books + '}';
    }

    public Book getBook(int index)
    {
        return books.get(index);
    }

    @Override
    public Library clone() throws CloneNotSupportedException
    {
        Library cloned = (Library) super.clone();
        cloned.books = new ArrayList<>();
        for (Book book : this.books)
        {
            cloned.books.add(book.clone());
        }
        return cloned;
    }

    private ArrayList<Book> books;
}
```