Cloneable

Описание
Как реализовывать
Случай без наследования
Полиморфное клонирование при наследовании
Тонкость: final-поля
Полиморфное клонирование и подклассы

Интерфейс: `java.lang.Cloneable`

Описание

Интерфейс без методов, обозначающий, что объект поддерживает клонирование — глубокое копирование, при котором создаётся новый объект с копией состояния текущего объекта, желательно полностью независимый от исходного.

Вот полное определение этого интерфейса:

```
public interface Cloneable { }
```

Клонирование используется в первую очередь затем, чтобы после него работать с двумя копиями данных независимо, так, чтобы они не портили друг друга. Это полезно, например, при *защитном копировании* (defensive copying) данных, передаваемых во внешний код. (См. Effective Java.)

Как реализовывать

Лучше — никак. Неизменяемые объекты клонировать нет смысла, для изменяемых же предпочтительнее идиома конструктора копии. Даже стандартные классы, поддерживающие клонирование, далеко не всегда удовлетворяют принципу глубокого копирования. Массивы и стандартные контейнеры при клонировании просто копируют ссылки на свои элементы в новый объект, не применяя к ним глубокого копирования.

Сам по себе интерфейс `Cloneable` не содержит методов, а метод `Object.clone`, который переопределяют реализующие `Cloneable` классы, объявлен как `protected`. Поэтому классы, реализующие `Cloneable`, нельзя даже клонировать единообразным образом без использования рефлексии.

Кроме того, механизм клонирования был спроектирован в Java 1.0. Как и многие компоненты стандартной библиотеки Java 1.0, он во многом спроектирован ужасно, и его не стоит рассматривать как пример для подражания.

Вы точно хотите продолжать?

Если всё-таки хотите — читайте дальше.

Случай без наследования

Поскольку в интерфейсе `Cloneable` по историческим причинам отсутствует публичный метод `clone`, нам придётся объявить его самостоятельно.

Сложность клонирования заключается в том, что оно должно возвращать объект того же класса, что и клонируемый объект. В классах, спроектированных для наследования, это требование обеспечить не так просто, ведь производные классы должны возвращать экземпляры производного класса, а не базового.

Для классов, не предназначенных для наследования, метод `clone` гарантированно будет возвращать экземпляр самого класса, а не какого-то из его (отсутствующих!) подклассов, поэтому для них легко реализовать метод `clone` через конструктор копии:

```
public final class Point implements Cloneable {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    // Конструктор копии
    public Point(Point p) {
        this(p.x, p.y);
    }

    @Override
    public Point clone() {
        return new Point(this);
    }
}
```

(Почему `@Override`? Что мы здесь переопределяем? До этого мы доберёмся.)

Поскольку класс `Point` объявлен как `final`, он гарантирует, что любой объект, возвращаемый его методом `clone`, будет экземпляром самого класса `Point`, и что сам метод `clone` не будет переопределён в подклассах. Такой контроль над иерархией классов значительно упрощает и проектирование, и реализацию. Ещё лучше, кстати, было бы определить класс `Point` как неизменяемый, после чего отпала бы даже необходимость в копировании.

Если же класс проектируется для наследования, реализация метода `clone` становится сложнее и обрастает нюансами.

Полиморфное клонирование при наследовании

В случае наследования нам нужен способ создать внутри метода `clone` экземпляр *именно того класса, для которого вызван метод* — это может быть либо наш класс, либо какой-то его подкласс. Эта задача — полиморфное клонирование — решается с помощью *псевдоконструктора* `Object.clone`. Этот метод объявлен в классе `Object` следующим образом:

```
protected native Object clone() throws CloneNotSupportedException;
```

На самом деле наш публичный метод `clone`, объявленный выше в классе `Point`, переопределяет метод `Object.clone`, поэтому он и помечен аннотацией `@Override`.

Метод `Object.clone` ужасен. Нет, правда, он действительно ужасен. Он сочетает в себе сразу

несколько плохих архитектурных решений, которые не стоит воспроизводить в своём коде.

- `Object.clone` создаёт экземпляр того же класса, что и вызывающий класс, не вызывая никакой его конструктор. Поэтому он и называется псевдоконструктором.
- `Object.clone` возвращает *поверхностную копию* (shallow copy) вызывающего объекта с копиями всех его полей. Поля объектного типа копируются по ссылке и продолжают ссылаться на те же объекты, что и поля исходного объекта. Это может нарушить некоторые предположения, которые объект делает о своём состоянии.
- Если вызывающий класс не реализует интерфейс `Cloneable`, то `Object.clone` бросает исключение `CloneNotSupportedException`. Это грубое нарушение основной цели интерфейсов — описание контракта, который должны поддерживать реализующие классы. В интерфейсе `Cloneable` вообще нет методов — он не определяет контракт, а изменяет поведение базового класса, и это единственный пример такого использования интерфейсов во всей стандартной библиотеке. Здесь была бы уместнее аннотация, но, к сожалению, в Java 1.0 аннотаций ещё не было.
- Исключение `CloneNotSupportedException` является проверяемым — несмотря на то, что метод `Object.clone` является `protected`, доступен только из подклассов, и выброс этого интерфейса сигнализирует об ошибке в коде, а не о нештатной ситуации в среде исполнения. Здесь уместнее было бы непроверяемое исключение.
- Поскольку публичные методы `clone` переопределяют `Object.clone`, к методу `Object.clone` нельзя непосредственно доступиться из классов, производных от классов с публичным методом `clone`. Если в суперклассе публичный метод `clone` реализован неправильно — tough cookies.

> 
> **Важно!** В языке Java есть два псевдоконструктора: `clone` и `readObject`. Со вторым из них мы познакомимся позже, при рассмотрении встроенного механизма сериализации.

Тем не менее в реализации `clone` для классов, спроектированных для наследования, нам придётся использовать `Object.clone`, потому что лучше всё равно ничего нет: мы не можем обязать каждый подкласс переопределять метод `clone`, как и не можем обязать каждый подкласс реализовывать конструктор копии, чтобы вызывать его рефлексией.

Итак, как реализовать свой метод `clone` через `Object.clone`?

```
public class Line implements Cloneable {
    private Point start;
    private Point end;

    @Override
    public Line clone() {
        // что-то там с Object.clone()
    }
}
```

По шагам:

- Сначала разберёмся с типом возвращаемого значения. Метод `Object.clone` возвращает `Object`, но объект, возвращаемый `Line.clone`, является как минимум экземпляром `Line`

(возможно — одного из подклассов `Line`), поэтому можно объявить наш метод `clone` как возвращающий объект типа `Line`. Вообще считается хорошим тоном при реализации `clone` объявлять его как возвращающий объект того же класса. (К сожалению, по историческим причинам в стандартных классах `Date`, `ArrayList` и т.д. этого не сделано.)

- Следующая неприятность — исключение `CloneNotSupportedException`. Поскольку наш класс реализует `Cloneable`, то оно никогда не выбросится, поэтому имеет смысл обернуть его в идиому "невозможное исключение" (см. статью по `Throwable`):

```
@Override
public Line clone() {
    Line result;

    try {
        // Вызываем Object.clone() через синтаксис super
        result = (Line) super.clone();
    } catch (CloneNotSupportedException e) {
        // Идиома "невозможное исключение"
        throw new AssertionError(e);
    }

    // Что-то ещё?
    return result;
}
```

- Если объект не содержит ссылок на изменяемые объекты, на этом можно и остановиться — поверхностное копирование, выполняемое по умолчанию методом `Object.clone`, нам подходит. В противном случае, чтобы сделать состояние оригинала и копии полностью независимым, нужно скопировать и эти объекты тоже — методом `clone`, конструктором копии, либо, в худшем случае, ручным копированием их состояния. В нашем случае поля класса `Line` включают две ссылки на объекты класса `Point`, который сам является изменяемым, поэтому нужно скопировать и их:

```
@Override
public Line clone() {
    Line result;

    try {
        // Вызываем Object.clone() через синтаксис super
        result = (Line) super.clone();
    } catch (CloneNotSupportedException e) {
        // Идиома "невозможное исключение"
        throw new AssertionError(e);
    }

    // В этом месте пока ещё start и result.start ссылаются на один и тот же объект,
    // аналогично end и result.end, поэтому копируем

    result.start = start.clone();
    result.end = end.clone();
    return result;
}
```

Тонкость: final-поля

Что было бы, если бы поля класса `Line` были объявлены как `final`? Как бы мы реализовали в этом случае полиморфное клонирование?

```
public class Line implements Cloneable {
    private final Point start;
    private final Point end;

    @Override
    public Line clone() {
        <...>

        // Ошибка компиляции!
        result.start = start.clone();
        result.end = end.clone();
        return result;
    }
}
```

...А никак.

То есть, серьёзно, никак.

Это фундаментальное ограничение полиморфного клонирования: оно несовместимо со стандартными языковыми средствами обеспечения неизменяемости. Псевдоконструктор `Object.clone` является *внеязыковым* средством создания новых экземпляров класса, и компилятор ничего не знает про особенности его работы. Компилятор видит только попытку перезаписать ссылки, помеченные как `final`, и отказывается компилировать такой код.

К сожалению, единственный способ заставить этот код компилироваться — это убрать с полей ограничение `final`. Это ещё одна причина, мешающая сочетать полиморфное клонирование с хорошим проектированием иерархии классов.

Полиморфное клонирование и подклассы

Одно хорошее свойство полиморфного клонирования состоит в том, что в подклассах, не добавляющих новых ссылок на изменяемые объекты, мы "бесплатно" получаем правильную поддержку клонирования.

Допустим, мы объявили подкласс класса `Line` с дополнительным полем — цветом линии:

```
public class ColorLine extends Line {
    private Color color;
}
```

Если класс `Color` является неизменяемым, то полиморфное клонирование вернёт объект типа `ColorLine`, а не `Line`:

```
ColorLine original = <...>;
Line copy = original.clone();
System.out.println(copy.getClass());  // class ColorLine
```

Здесь есть одна тонкость. Класс `ColorLine` наследует метод `clone` от класса `Line`, поэтому он объявлен как возвращающий `Line`, а не `ColorLine`. Чтобы пользователю не приходилось каждый раз делать приведение типа, имеет смысл переопределить его тривиальной реализацией:

```
public class ColorLine extends Line {
    private Color color;

    @Override
    public ColorLine clone() {
        // Вызываем Line.clone() через синтаксис super
        return (ColorLine) super.clone();
    }
}
```

Конечно, любые ссылки на изменяемые объекты, добавленных в подклассе, нужно будет дополнительно скопировать явно, потому что реализация метода `clone` в суперклассе этого не сделает. Так, если бы класс `Color` был изменяемым, нам нужно было бы дополнительно скопировать объект `color`:

```
public class ColorLine extends Line {
    private Color color;

    @Override
    public ColorLine clone() {
        // Вызываем Line.clone() через синтаксис super
        ColorLine result = (ColorLine) super.clone();
        result.color = color.clone();
        return result;
    }
}
```

Обратите внимание, что при переопределении публичного метода `clone` нам не приходится ловить исключение `CloneNotSupportedException`, потому что ранее при переопределении метода `Object.clone` в классе `Line` мы убрали это исключение из объявления метода.