Занятие № 9.

Продвинутое ООП

mutable, immutable объекты
Создание immutable объекта

Клонирование и копирование объекта

Equals(), hashCode()

Основы многопоточности

mutable, immutable объекты

Неизменяемый объект - это такой объект, чье внешнее видимое состояние не может измениться после его создания. Классы String, Integer и BigDecimal в библиотеке классов Java являются примерами неизменяемых объектов - они представляют отдельное значение, которое не может измениться в течение жизненного цикла объекта.

Преимущества неизменяемости

Неизменяемые классы при правильном использовании могут значительно упростить программирование. Они могут находиться только в одном состоянии, поэтому если они правильно сконструированы, они никак не могут быть в несогласованном состоянии. Вы можете свободно делать общими и кэшировать ссылки на неизменяемые объекты без необходимости копировать или клонировать их; вы можете кэшировать их поля или результаты их методов, не беспокоясь о том, что значения устареют или станут несогласованными с остальными состояниями объекта. Из неизменяемых классов обычно получаются лучшие map key. По существу они поточно-ориентированы, поэтому вам не нужно синхронизировать доступ к ним через потоки.

Возможность кэширования

Поскольку нет опасности, что неизменяемые объекты изменят свое значение, можно свободно кэшировать ссылки на них и быть уверенным, что эта ссылка обратится к этому же значению позже. Соответственно, поскольку их свойства не могут измениться, можно кэшировать их поля и результаты их методов.
Если объект изменяемый, вам нужно быть осторожными, сохраняя ссылку на него. Рассмотрим код, где в очередь на выполнение планировщиком поставлены два задания . Цель в том, чтобы первая задача начала выполняться сейчас, а вторая - через один день.

Потенциальная проблема с изменяемым объектом Date (дата)

```
Date d = new Date();
Scheduler.scheduleTask(task1, d);
d.setTime(d.getTime() + ONE_DAY);
scheduler.scheduleTask(task2, d);
```

Поскольку Date - изменяемый объект, метод scheduleTask должен на всякий случай скопировать параметр даты (возможно через clone()) во внутреннюю структуру данных. Иначе как task1, так и task2 могут выполниться завтра, что не соответствует нашей цели. Хуже того, внутренняя структура данных, использующаяся планировщиком задач, может быть повреждена. Очень легко забыть скопировать параметр даты при написании метода типа scheduleTask(). Если вы об этом все же забудете, то создадите неуловимую ошибку, которая некоторое время не будет проявляться, а когда проявится, потребуется долгое время, чтобы ее отследить. Неизменяемый класс Date сделал бы ошибку такого рода невозможной.

Внутренняя безопасность потоков

Многие вопросы, связанные с безопасностью потоков, возникают, когда множественные потоки пытаются параллельно модифицировать состояние объекта (конфликт по совпадению обращений для записи) или когда один поток пытается получить доступ к состоянию объекта, в то время как другой поток модифицирует его (конфликт по совпадению обращений при считывании и записи.) Для предупреждения таких конфликтов необходимо синхронизировать доступ к совместно используемым объектам, так, чтобы другие потоки не могли получить к ним доступ, пока они в несогласованном состоянии.

Возможно, это будет трудно осуществить правильно, требуется значительное количество документации для того, чтобы программа была правильно расширена, а также могут возникнуть отрицательные последствия, связанные с производительностью. Если неизменяемые объекты сконструированы правильно (это означает, что ссылка на объект покинула пределы конструктора), нет необходимости синхронизировать доступ, так как их состояние не может быть изменено, и поэтому конфликт по совпадению обращений для записи или по совпадению обращений при считывании и записи не может возникнуть.
Возможность делать общими ссылки на неизменяемые объекты в потоках без синхронизации может значительно облегчить процесс синхронизации написания параллельных программ и снижает количество потенциальных ошибок в них.

Безопасность при работе с "плохим" кодом

Методы, которые принимают объекты в качестве аргументов, не должны изменять состояние этих объектов, кроме тех случаев, когда им это документально предписано или они становятся владельцами этого объекта. Когда мы передаем объект обычному методу, мы обычно не ожидаем его возвращения в неизмененном виде. Однако по отношению к изменяемым объектам такая уверенность просто неразумна. Если мы передаем java.awt.Point такому методу, как Component.setLocation(), ничто не мешает setLocation модифицировать местоположение передаваемого Point или сохранить ссылку на этот point и изменить его позже в другом методе. (Конечно, Component этого не делает, потому, что это было бы "грубо", но не все классы такие "вежливые".) Теперь состояние нашего объекта Point изменилось без нашего ведома, и результаты потенциально опасны - мы все еще думаем, что point в одном месте, а он, фактически, находится в другом. Однако, если бы объектPoint был неизменяемым, такой "враждебный" код не смог бы модифицировать состояние нашей программы таким сбивающим с толку и опасным способом.

Хорошие ключи

Из неизменяемых объектов получаются лучшие ключи HashMap или HashSet.
Некоторые изменяемые объекты изменяют значение hashcode() в зависимости от своего состояния.
Если вы используете такой изменяемый объект, как ключ HashSet, а затем объект изменяет свое состояние, при реализации HashSet может возникнуть путаница - объект все еще будет присутствовать, если пронумеровать набор (set), но может оказаться, что его нет, если обратиться с запросом к набору при помощи contains().
Нет необходимости говорить, что это могло бы вызвать непредсказуемое поведение.

Код ниже, где это демонстрируется, напечатает "false", "1," и "moo."

Изменяемый класс StringHolder, неподходящий для использования в качестве ключа

```
public class StringHolder  {
    private String string

    public StringHolder(string's) {
        this.string = s;
    }

    public string getstring() {
        return string;
    }

    public void setstring(string string) {
        this.string = string;
    }
```

```
    public boolean equals(Object o) {
        if (this == o)
            return true;
        else if (o == null || !(o instanceof stringHolder))
            return false;
        else {
            final stringHolder other = (stringHolder) o;
            if (string == null)
                return (other.string == null);
            else
                return string);
        }
    }

    public int hashcode() {
        return (string != null ? string.hashcode() : 0);
    }

    public string tostring() {
        return string;
    }

    ...

    stringHolder sh = new stringHolder(";blert";);
    HashSet h = new HashSet();
    h.add(sh);
    sh.setstring(";moo";);
    System.out.println(h.contains(sh));
    System.out.println(h.size());
    System.out.println(h.iterator().next());
}
```

Когда использовать неизменяемые классы

Неизменяемые классы идеальны для представления значения абстрактных типов данных, таких как числа, перечислимые типы или цвета. Основные числовые классы в библиотеке классов Java, такие как Integer, Long и Float - неизменяемые, так же как и стандартные числовые типы, такие как BigInteger и BigDecimal. Классам для представления сложных чисел или рациональных чисел произвольной точности лучше обладать неизменяемостью. В зависимости от вашего приложения даже абстрактные типы, которые содержат много дискретных значений, таких как векторы или матрицы, могли бы применяться как неизменяемые классы.

Еще один пример неизменяемости в библиотеке классов Java - java.awt.Color. Поскольку цвета обычно представляются в виде упорядоченного набора числовых значений в некоторых способах цветового представления (таких как RGB, HSB или CMYK), кажется более разумным рассматривать цвет как выделенное значение в пространстве цветов, а не как упорядоченный набор индивидуально адресуемых значений, и поэтому есть смысл реализовать Color как неизменяемый класс.

Следует ли представлять объекты, являющиеся контейнерами для множественных элементарных значений, таких как точки, векторы, матрицы или цвета RGB, при помощи изменяемых или неизменяемых объектов? Ответ - когда как... Как они будут использованы? Используются ли они главным образом для представления многомерных значений (например, цвета пиксела), или просто как контейнеры для совокупности связанных друг с другом свойств некоторого другого объекта (например, высоты и ширины окна)? Как часто эти свойства будут изменяться? Если они изменяются, имеют ли значения индивидуального компонента собственный смысл в приложении?

События - еще один яркий пример возможной реализации неизменяемых классов. События кратковременны, и часто используются не в том потоке, где были созданы, поэтому использование их как неизменяемых имеет больше преимуществ, чем недостатков. Большая часть классов событий AWT реализуются не как строго неизменяемые, а с небольшими модификациями. Соответственно, в системе, которая использует какую-либо форму обмена сообщениями для общения между компонентами, использование пересылаемых объектов в качестве неизменяемых, вероятно, является разумным.

Создание immutable объекта

Immutable объект - это объект, состояние которого после создания невозможно изменить. В случае Java это значит, что все поля у класса отмечены как final и являются примитивами или тоже immutable типами.

Пример:

```
public class ImmutablePoint {
    private final int x;
    private final int y;
    private final String description;

    public ImmutablePoint(int x, int y, String description) {
        this.x = x;
        this.y = y;
        this.description = description;
    }
}
```

После создания экземпляра ***ImmutablePoint*** его модификация невозможна.

Простейший пример immutable класса из JDK - это ***String***. Любые методы, которые вы вызовите на строке (например ***description.toLowerCase()***) ,вернут новую строку, а не модифицируют исходную.

Пример mutable класса из JDK - ***Date***. Например, ***myDate.setHours(x)*** модифицирует сам экземпляр myDate.

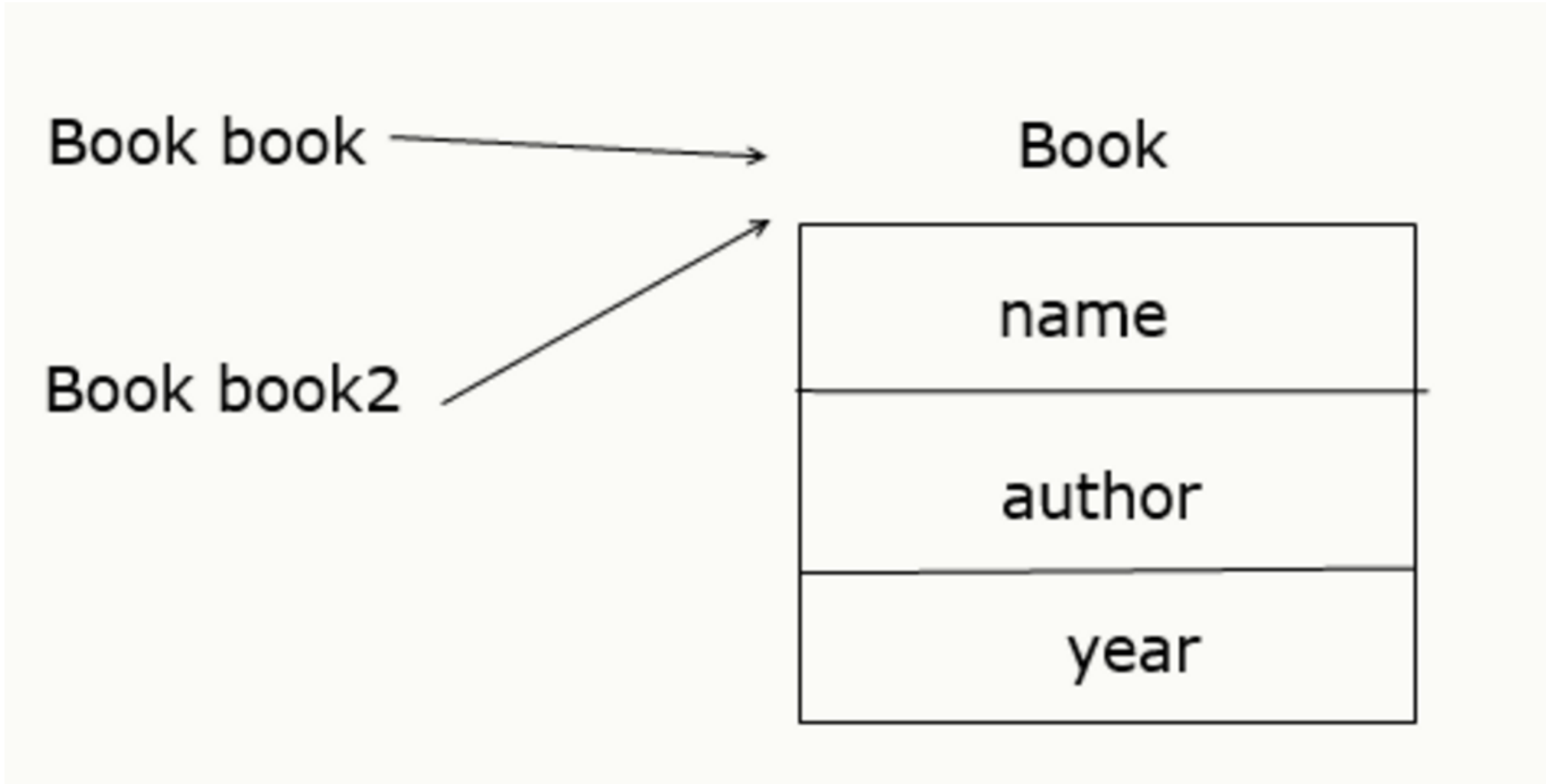
В случае многопоточного программирования преимущества immutable классов очевидны: после создания объекты можно передавать другим потокам и они всегда будут в актуальном состоянии. Т.е. вам не надо проверять, не устарело ли состояние вашего экземпляра и не модифицировал ли его другой поток пока вы с ним работаете. Например, у вас есть метод ***bill(Date endDate)***, в нём вы наивно проверяете соответствие endDate каким-то предварительным условиям и начинаете с ней работать. В этот момент другой поток может изменить endDate, например установит её глубоко в прошлое. Последствия могут быть самыми удивительными.

Ссылочные типы и клонирование объектов

При работе с объектами классов надо учитывать, что они все представляют ссылочные типы, то есть указывают на какой-то объект, расположенный в памяти. Чтобы понять возможные трудности, с которыми мы можем столкнуться, рассмотрим пример:

```
Book book = new Book("Война и мир", "Л. Толстой", 1863);
Book book2 = book;
book2.setName("Отцы и дети");
System.out.println(book.getName());
```

Здесь создаем два объекта Book и один присваиваем другому. Но, несмотря на то, что мы изменяем только объект book2, вместе с ним изменяется и объект book. Потому что после присвоения они указывают на одну и ту же область в памяти, где собственно данные об объекте Book и его полях и хранятся.


Book book
Book book2
Book
name
author
year

Чтобы избежать этой проблемы, необходимо создать отдельный объект для переменной book2, например, с помощью метода `clone`:

```
class Book implements Cloneable{

        //остльной код класса

        public Book clone() throws CloneNotSupportedException{

        return (Book) super.clone();
    }
}
```

Для реализации клонирования класс Book должен применить интерфейс **Cloneable**, который определяет метод clone. Реализация этого метода просто возвращает вызов метода clone для родительского класса - то есть класса Object с преобразованием к типу Book.

Кроме того, на случай если класс не поддерживает клонирование, метод должен выбрасывать исключение **CloneNotSupportedException**, что определяется с помощью оператора **throws**.

Затем с помощью вызова этого метода мы можем осуществить копирование:

```
try{
    Book book = new Book("Война и мир", "Л. Толстой", 1863);
    Book book2 = book.clone();
}
catch(CloneNotSupportedException ex){

    System.out.println("Не поддерживается клонирование");
}
```

Однако данный способ осуществляет **неполное копирование** и подойдет, если клонируемый объект не содержит сложных объектов. Например, пусть класс Book имеет следующее определение:

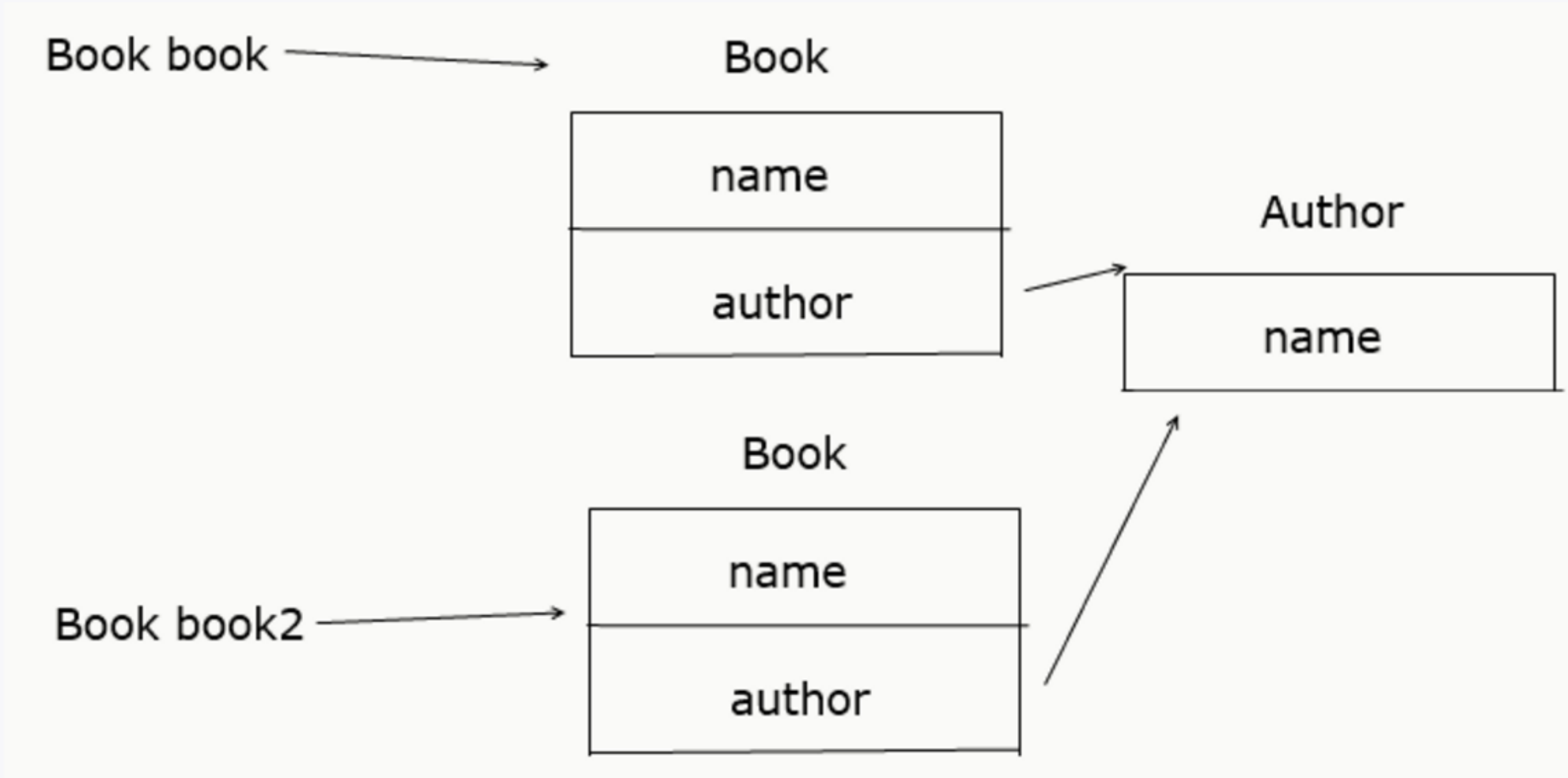
```
class Book implements Cloneable{

    private String name;
    private Author author;

    public void setName(String n){ name=n;}
    public String getName(){ return name;}

    public void setAuthor(String n){ author.setName(n);}
    public String getAuthor(){ return author.getName();}
```

```
    Book(String name, String author){

        this.name = name;
        this.author = new Author(author);
    }

    public String toString(){

        return "Книга '" + name + "' (автор " +  author + ")";
    }

    public Book clone() throws CloneNotSupportedException{

        return (Book) super.clone();
    }
}

class Author{

    private String name;

    public void setName(String n){ name=n;}
    public String getName(){ return name;}

    public Author(String name){

        this.name=name;
    }
}
```

Если мы попробуем изменить автора книги, нас последует неудача:

```
try{
    Book book = new Book("Война и мир", "Л. Толстой");
    Book book2 = book.clone();
    book2.setAuthor("И. Тургенев");
    System.out.println(book.getAuthor());
}
catch(CloneNotSupportedException ex){

    System.out.println("Не поддерживается клонирование");
}
```

В этом случае, хотя переменные book и book2 будут указывать на разные объекты в памяти, но эти объекты при этом будут указывать на один объект Author.



И в этом случае нам необходимо выполнить полное копирование. Для этого, во-первых, надо определить метод клонирования у класса Author:

```
class Author implements Cloneable{

    // остальной код класса

    public Author clone() throws CloneNotSupportedException{

        return (Author) super.clone();
    }
}
```

И затем исправим метод clone в классе Book следующим образом:

```
public Book clone() throws CloneNotSupportedException{

    Book newBook = (Book) super.clone();
    newBook.author=(Author) author.clone();
    return newBook;
}
```

Более безопасным способом является использование конструктора копирования, которое позволяет избежать многих проблем. Например:

```
class Foo {
  private int someInt;
  private String someString;

  // Getters & Setters

  public Foo(int val, String str) {
    someInt = val;
    someString = str;
  }

  // Copy constructor
  public Foo(Foo aFoo) {
    Foo(aFoo.getSomeInt(), aFoo.getSomeString());
  }

  // Another style
  public static Foo newInstance(Foo aFoo) {
    return new Foo(aFoo.getSomeInt(), aFoo.getSomeString());
  }
}
```

Другим безопасным вариантом является фабричный метод (Factory method), который представляет собой статический метод, возвращающий экземпляр своего класса.
Фабричный метод имеет следующие преимущества перед конструктором копирования:
- Имеет имя (чаще всего getInstance или valueOf), что делает код более понятным.
- Необязательно создавать новый объект в результате вызова: Объекты могут быть кэшированы и реиспользованы.
- Могут возвращать подтип своего возвращаемого типа. В частности, могут возвращать объект, у которого неизвестен класс реализации.

Еще одним распространенным способом клонирования является сериализация, которая реализуется сложнее и, в общем, имеет те же проблемы, что и метод clone() , и кроме того связывает класс контрактом интерфейса Serializable.

Equals(), hashCode()

Методы **hashCode**() и **equals**() были определены в классе Object, который является родительским классом для объектов java. Поэтому все java объекты наследуют от этих методов реализацию по умолчанию.

Использование hashCode() и equals()

Метод hashCode() используется для получения уникального целого номера для данного объекта. Когда необходимо сохранить объект как структуру данных в некой хэш-таблице (такой объект также называют корзиной — bucket), этот номер используется для определения его местонахождения в этой таблице. По умолчанию, метод hashCode() для объекта возвращает номер ячейки памяти, где объект сохраняется.
Метод equals(), как и следует из его названия, используется для простой проверки равенства двух объектов. Реализация этого метода по умолчанию просто проверяет по ссылкам два объекта на предмет их эквивалентности.

Переопределение поведения по умолчанию.

Все работает отлично до тех пор, пока вы не переопределяете ни один из этих методов в своих классах. Но иногда в приложениях необходимо изменять поведение по умолчанию некоторых объектов.
Давайте возьмем пример, где в вашем приложении имеется объект Employee. Давайте напишем минимально возможную структуру такого класса.

```java
public class Employee
{
    private Integer id;
    private String firstname;
    private String lastName;
    private String department;

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getFirstname() {
        return firstname;
    }
    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getDepartment() {
        return department;
    }
    public void setDepartment(String department) {
        this.department = department;
    }
}
```

Описанный выше класс Employee имеет некоторые основные атрибуты и методы доступа.
Сейчас рассмотрим простую ситуацию, где необходимо сравнить два объекта класса Employee.

```
public class EqualsTest {
     public static void main(String[] args) {
          Employee e1 = new Employee();
          Employee e2 = new Employee();

          e1.setId(100);
          e2.setId(100);
          //Печатает false в консоли
          System.out.println(e1.equals(e2));
     }
}
```

Видно, что написанный выше метод вернет “false”. Но правильно ли это на самом деле, учитывая, что эти оба объекта одинаковые? В real time application метод должен вернуть true. Чтобы достигнуть корректного поведения, нам нужно переопределить метод equals(), как и сделано ниже:

```
public boolean equals(Object o) {
        if(o == null)
        {
            return false;
        }
        if (o == this)
        {
           return true;
        }
        if (getClass() != o.getClass())
        {
            return false;
        }
        Employee e = (Employee) o;
        return (this.getId() == e.getId());
}
```

Добавьте этот метод в свой класс Employee, и проверка на эквивалентность вернет “**true**”. Однако, всё ли мы сделали? Пока нет. Давайте проверим наш модифицированный класс еще одним способом.

```
import java.util.HashSet;
import java.util.Set;

public class EqualsTest
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee();
        Employee e2 = new Employee();

        e1.setId(100);
        e2.setId(100);

        //Печатает 'true'
        System.out.println(e1.equals(e2));

        Set employees = new HashSet();
        employees.add(e1);
        employees.add(e2);
        //Печатает два объекта
        System.out.println(employees);
    }
}
```

Команда System.out.println(employee) распечатывает два объекта. Если оба объекта были эквивалентны, а в Set содержатся только уникальные объекты, то внутри HashSet должен быть только один экземпляр, т.е. оба объекта ссылаются на одинаковые экземпляры класса Employee.
Что же мы упустили?
Мы упустили второй важный метод hashCode(). Как сказано в документации java, если вы переопределяете метод equals(), то вы **обязаны** переопределить метод hashCode(). Итак, давайте добавим ещё один метод в наш класс Employee.

```
@Override
 public int hashCode()
 {
    final int PRIME = 31;
    int result = 1;
    result = PRIME * result + getId();
    return result;
 }
```

Мы добавили один раз этот метод в наш класс, и на печать будет выведен только один объект, и, таким образом, проверка эквивалентности e1 и e2 показала true.

То, что важно помнить.

1) Всегда используйте те же атрибуты объекта для вызова и hashCode() и equals(). Как раз в нашем случае, мы использовали employee id.

2) Метод equals() должен быть устойчивым (если объект не изменялся, метод должен возвращать то же самое значение).

3) Всякий раз, когда **a.equals(b)**, то a.hashCode() должно быть таким же, как b.hashCode().

4) Если вы переопределили один метод, то обязательно должны переопределить второй.

Основы многопоточности

Наиболее очевидная область применения многопоточности – это программирование интерфейсов.

Многопоточность незаменима тогда, когда необходимо, чтобы графический интерфейс продолжал отзываться на действия пользователя во время выполнения некоторой обработки информации. Например, поток, отвечающий за интерфейс, может ждать завершения другого потока, загружающего файл из интернета, и в это время выводить некоторую анимацию или обновлять прогресс-бар. Кроме того он может остановить поток, загружающий файл, если была нажата кнопка «отмена».

Еще одна популярная и, пожалуй, одна из самых хардкорных областей применения многопоточности – игры. В играх различные потоки могут отвечать за работу с сетью, анимацию, расчет физики и т.п.

Процессы

Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса, но бывают и исключения (например, браузер Chrome создает отдельный процесс для каждой вкладки, что дает ему некоторые преимущества, вроде независимости вкладок друг от друга). Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств).
Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

Операционная система оперирует так называемыми страницами памяти, которые представляют собой просто область определенного фиксированного размера. Если процессу становится недостаточно памяти, система выделяет ему дополнительные страницы из физической памяти. Страницы виртуальной памяти могут проецироваться на физическую память в произвольном порядке.
При запуске программы операционная система создает процесс, загружая в его адресное пространство код и данные программы, а затем запускает главный поток созданного процесса.

Потоки

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

Запуск потоков

Каждый процесс имеет хотя бы один выполняющийся поток. Тот поток, с которого начинается выполнение программы, называется главным. В языке Java, после создания процесса, выполнение главного потока начинается с метода main(). Затем, по мере необходимости, в заданных программистом местах, и при выполнении заданных им же условий, запускаются другие, побочные потоки.

В языке Java поток представляется в виде объекта-потомка класса Thread. Этот класс инкапсулирует стандартные механизмы работы с потоком.

Создать новый поток можно двумя способами:

Способ 1

Создать объект класса Thread, передав ему в конструкторе нечто, реализующее интерфейс Runnable. Этот интерфейс содержит метод run(), который будет выполняться в новом потоке. Поток закончит выполнение, когда завершится его метод run().

Выглядит это так:

```
class SomeThing                 //Нечто, реализующее интерфейс Runnable
implements Runnable             //(содержащее метод run())
{
        public void run()               //Этот метод будет выполняться в побочном потоке
        {
                System.out.println("Привет из побочного потока!");
        }
}

public class Program                    //Класс с методом main()
{
        static SomeThing mThing;        //mThing - объект класса, реализующего интерфейс Runnable

        public static void main(String[] args)
        {
                mThing = new SomeThing();

                Thread myThready = new Thread(mThing);  //Создание потока "myThready"
                myThready.start();                              //Запуск потока

                System.out.println("Главный поток завершён...");
        }
}
```

Способ 2

Создать потомка класса Thread и переопределить его метод run():

```
class AffableThread extends Thread
{
        @Override
        public void run()       //Этот метод будет выполнен в побочном потоке
        {
                System.out.println("Привет из побочного потока!");
        }
}

public class Program
{
        static AffableThread mSecondThread;

        public static void main(String[] args)
        {
                mSecondThread = new AffableThread();    //Создание потока
                mSecondThread.start();                                  //Запуск потока

                System.out.println("Главный поток завершён...");
        }
}
```

Для демонстрации параллельной работы потоков давайте рассмотрим программу, в которой два потока спорят на предмет философского вопроса «что было раньше, яйцо или курица?». Главный поток уверен, что первой была курица, о чем он и будет сообщать каждую секунду. Второй же поток раз в секунду будет опровергать своего оппонента. Всего спор продлится 5 секунд. Победит тот поток, который последним изречет свой ответ на этот, без сомнения, животрепещущий философский вопрос. В примере используются средства, о которых пока не было сказано (isAlive(), sleep() и join()). К ним даны комментарии.

```
class EggVoice extends Thread
{
        @Override
        public void run()
        {
                for(int i = 0; i < 5; i++)
                {
                        try{
                                sleep(1000);            //Приостанавливает поток на 1 секунду
                        }catch(InterruptedException e){}

                        System.out.println("яйцо!");
                }
                //Слово «яйцо» сказано 5 раз
        }
}
```

```
public class ChickenVoice       //Класс с методом main()
{
        static EggVoice mAnotherOpinion;        //Побочный поток

        public static void main(String[] args)
        {
                mAnotherOpinion = new EggVoice();       //Создание потока
                System.out.println("Спор начат...");
                mAnotherOpinion.start();                        //Запуск потока

                for(int i = 0; i < 5; i++)
                {
                        try{
                                Thread.sleep(1000);             //Приостанавливает поток на 1 секунду
                        }catch(InterruptedException e){}

                        System.out.println("курица!");
                }

                //Слово «курица» сказано 5 раз

                if(mAnotherOpinion.isAlive())   //Если оппонент еще не сказал последнее слово
                {
                        try{
                                mAnotherOpinion.join(); //Подождать пока оппонент закончит высказываться.
                        }catch(InterruptedException e){}

                        System.out.println("Первым появилось яйцо!");
                }
                else    //если оппонент уже закончил высказываться
                {
                        System.out.println("Первой появилась курица!");
                }
                System.out.println("Спор закончен!");
        }
}
```

```
Консоль:
Спор начат...
курица!
яйцо!
яйцо!
курица!
яйцо!
курица!
яйцо!
курица!
яйцо!
курица!
Первой появилась курица!
Спор закончен!
```

В приведенном примере два потока параллельно в течение 5 секунд выводят информацию на консоль. Точно предсказать, какой поток закончит высказываться последним, невозможно. Можно попытаться, и можно даже угадать, но есть большая вероятность того, что та же программа при следующем запуске будет иметь другого «победителя». Это происходит из-за так называемого «асинхронного выполнения кода». Асинхронность означает то, что нельзя утверждать, что какая-либо инструкция одного потока, выполнится раньше или позже инструкции другого. Или, другими словами, параллельные потоки независимы друг от друга, за исключением тех случаев, когда программист сам описывает зависимости между потоками с помощью предусмотренных для этого средств языка.