**5. Basics of OOP**



# OOP, Classes and Objects, Inheritance, Class Object, Interfaces

1. Дайте развернутое объяснение трем концепциям ООП.

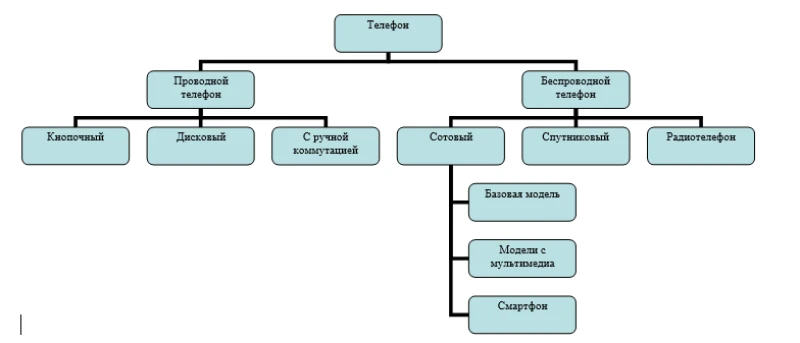
Answer

1. ***Полиморфизм*** – это принцип ООП, позволяющий переопределять методы.

При переопределении метода, используется аннотация **@Override**, которая подсказывает компилятору о необходимости проверить сигнатуры переопределяемого и переопределяющего методов.

1. ***Абстракция*** – эта концепция позволяет абстрактно создавать и описывать объект.

Пример:



C помощью абстракции можно выделить в этой иерархии объектов общую информацию: общий абстрактный тип объектов — телефон, общую характеристику телефона — год его создания, и общий интерфейс — все телефоны способны принимать и посылать вызовы.

Код

**public** **abstract** **class** AbstractPhone {

**private** **int** year;

**public** AbstractPhone(**int** year) {

**this**.year = year;

}

**public** **abstract** **void** call(**int** outputNumber);

**public** **abstract** **void** ring (**int** inputNumber);

}

С помощью абстракции мы выделяем общее для всех объектов

1. ***Инкапсуляция*** – это сокрытие элементов в коде программы для исключения ситуаций с изменением (класса, поля класса, метода) из вне с помощью атрибутов доступа.

Атрибуты доступа:

* **public** – это модификатор доступа, применяющийся к классам, интерфейсам, полям класса и методам. Даёт видимость на протяжении проекта.
* **protected** – это модификатор доступа, применяющийся к классам, интерфейсам, полям класса и методам. Даёт видимость на протяжении пакета и у наследников.
* **default** (по умолчанию) – это модификатор доступа, применяющийся к классам, интерфейсам, полям класса и методам. Даёт видимость на протяжении пакета.
* **private** – это модификатор доступа, применяющийся к внутренним классам, полям класса и методам. Даёт видимость в классе.

1. ***Наследование*** – это принцип ООП, позволяющий создавать иерархию классов, а значит значительно уменьшить объем кода, следовательно, и облегчить работу программисту.

**Объяснение:**

Посмотрев на диаграмму телефонов (из картинки выше), можно заметить, что она представляет собой иерархию, в которой модель, расположенная ниже обладает всеми признаками моделей, расположенных выше по ветке, плюс своими собственными. Например, смартфон, использует сотовую сеть для связи (обладает свойствами сотового телефона), является беспроводным и переносным (обладает свойствами беспроводного телефона) и может принимать и делать вызовы (свойствами телефона). В этом случае мы можем говорить о наследовании свойств объекта.

**В программировании наследование заключается в использовании уже существующих классов для описания новых.**

Рассмотрим пример создания класса смартфон с помощью наследования. Все беспроводные телефоны работают от аккумуляторных батарей, которые имеют определенный ресурс работы в часах. Поэтому добавим это свойство в класс беспроводных телефонов:

**public** **abstract** **class** WirelessPhone **extends** AbstractPhone {

**private** **int** hour;

**public** WirelessPhone(**int** year, **int** hour) {

**super**(year);

**this**.hour = hour;

}

}

Сотовые телефоны наследуют свойства беспроводного телефона, мы также добавили в этот класс реализацию методов call и ring:

**public** **class** CellPhone **extends** WirelessPhone {

**public** CellPhone(**int** year, **int** hour) {

**super**(year, hour);

}

@Override

**public** **void** call(**int** outputNumber) {

System.***out***.println("Вызываю номер " + outputNumber);

}

@Override

**public** **void** ring(**int** inputNumber) {

System.***out***.println("Вам звонит абонент " + inputNumber);

}

}

И, наконец, класс смартфон, который в отличие от классических сотовых телефонов имеет полноценную операционную систему. В смартфон можно добавлять новые программы, поддерживаемые данной операционной системой, расширяя, таким образом, его функциональность. С помощью кода класс можно описать так:

**public** **class** Smartphone **extends** CellPhone {

**private** String operationSystem;

**public** Smartphone(**int** year, **int** hour, String operationSystem) {

**super**(year, hour);

**this**.operationSystem = operationSystem;

}

**public** **void** install(String program) {

System.***out***.println("Устанавливаю " + program + "для" + operationSystem);

}

}

Как видите, для описания класса Smartphone мы создали совсем немного нового кода, но получили новый класс с новой функциональностью. Использование принципа наследование ООП позволяет значительно уменьшить объем кода, а значит, и облегчить работу программисту.

1. Опишите процедуру инициализации полей класса и полей экземпляра класса. Когда инициализируются поля класса, а когда – поля экземпляров класса. Какие значения присваиваются полям по умолчанию? Где еще в классе полям могут быть присвоены начальные значения?

Answer

Смотреть 4 раздел вопрос 1

1. Приведите правила, которым должен следовать компонент java-bean.

Answer

***Java-bean*** – это обычный java-класс, который создан согласно следующим правилам:

* Все поля класса с модификатором доступа private;
* Ко всем полям есть getter и setter;
* У класса есть публичный default (пустой) конструктор.

1. Дайте определение перегрузке методов. Как вы думаете, чем удобна перегрузка методов? Укажите, какие методы могут перегружаться, и какими методами они могут быть перегружены? Можно ли перегрузить методы в базовом и производном классах? Можно ли private метод базового класса перегрузить public методов производного? Можно ли перегрузить конструкторы, и можно ли при перегрузке конструкторов менять атрибуты доступа у конструкторов?

Answer

Смотреть раздел 4 вопрос 2

1. Объясните, что такое раннее и позднее связывание? Перегрузка – это раннее или позднее связывание? Объясните правила, которым следует компилятор при разрешении перегрузки; в том числе, если методы перегружаются примитивными типами, между которыми возможно неявное приведение или ссылочными типами, состоящими в иерархической связи.

Answer

Смотреть раздел 4 вопрос 3

1. Объясните, как вы понимаете, что такое неявная ссылка this? В каких методах эта ссылка присутствует, а в каких – нет, и почему?

Answer

Смотреть раздел 4 вопрос 4

1. Что такое финальные поля, какие поля можно объявить со спецификатором final? Где можно инициализировать финальные поля?

Answer

Смотреть раздел 4 вопрос 5

1. Что такое статические поля, статические финальные поля и статические методы. К чему имеют доступ статические методы? Можно ли перегрузить и переопределить статические методы? Наследуются ли статические методы?

Answer

Смотреть раздел 4 вопрос 6

1. Что такое логические и статические блоки инициализации? Сколько их может быть в классе, в каком порядке они могут быть размещены и в каком порядке вызываются?

Answer

Смотреть раздел 4 вопрос 7

1. Что представляют собой методы с переменным числом параметров, как передаются параметры в такие методы и что представляет собой такой параметр в методе? Как осуществляется выбор подходящего метода, при использовании перегрузки для методов с переменным числом параметров?

Answer

Смотреть раздел 4 вопрос 8

1. Чем является класс Object? Перечислите известные вам методы класса Object, укажите их назначение.

Answer

Смотреть раздел 4 вопрос 9

1. Что такое хэш-значение? Объясните, почему два разных объекта могут сгенерировать одинаковые хэш-коды?

Answer

Смотреть раздел 4 вопрос 10

1. Как вы думаете, для чего используется наследование классов в java-программе? Приведите пример наследования. Как вы думаете, поля и методы, помеченными модификатором доступа private, наследуются?

Answer

Смотреть вопрос 1

1. Укажите, как вызываются конструкторы при создании объекта производного класса? Что в конструкторе класса делает оператор super()? Возможно ли в одном конструкторе использовать операторы super() и this()?

Answer

Объект создается в несколько этапов.

1. Сначала выделяется память под все переменные – поля класса.
2. Затем идет инициализация базового класса.
3. Потом всем переменным присваиваются значения, если они указаны.
4. И наконец, вызывается конструктор.

Итого: сначала базовый класс, затем переменные вне конструктора, затем вызов кода конструктора.

Оператор **super()** вызывает конструктор родительского класса. Если класс не наследуется «явно», он по умолчанию наследуется от класса Object и вызывается конструктор у класса Object оператором super().

Внутри конструктора this и super должны стоять выше всех других выражений, в самом начале, иначе компилятор выдаст сообщение об ошибке. Из чего следует, что в одном конструкторе не может быть одновременно и this(), и super().

1. Объясните, как вы понимаете утверждения: “ссылка базового класса может ссылаться на объекты своих производных типов” и “объект производного класса может быть использован везде, где ожидается объект его базового типа”. Верно ли обратное и почему?

Answer

***“Ссылка базового класса может ссылаться на объекты своих производных типов”.***

Есть класс Person, от него наследует класс Employee. Так как Employee наследует от Person, то объект Employee одновременно является и объектом Person.

Person sam = new Employee("Sam", "Oracle");

Но данный объект будет иметь доступ к методам только от класса Person, а если нам нужен доступ к методам класса Employee, нам необходимо понизить ссылку до класса Employee.

***“Объект производного класса может быть использован везде, где ожидается объект его базового типа”.***

Это верно.

Объекты дочерних классов имеют все методы классов родителей. Поэтому дочерние классы имеют все возможности родительских классов. А родители не могут использовать те возможности, которые имеют наследники.

1. Что такое переопределение методов? Как вы думаете, зачем они нужны? Можно ли менять возвращаемый тип при переопределении методов? Можно ли менять атрибуты доступа при переопределении методов? Можно ли переопределить методы в рамках одного класса?

Answer

***Переопределение метода*** – это изменение реализации метода из класса родителя.

Переопределение метода нужно если у нас есть, для примера абстрактный класс Animal и у него есть метод “издавать звук”. От этого класса наследуются (extends) классы Cat и Dog, в них мы пишем разную реализацию метода, так как они издают разные звуки кот и собака, следовательно, у них будет разные реализации методов.

***Возвращаемый тип*** и ***атрибуты доступа*** при переопределении методов менять нельзя, так как будет ошибка компиляции.

Переопределять методы в рамках одного класса можно**, но** это будет называться перегрузкой методов.

1. Определите правило вызова переопределенных методов. Можно ли статические методы переопределить нестатическими и наоборот?

Answer

Вызов переопределённых методов производится следующим образом:

* Если метод статический – мы обращаемся к классу, у которого мы вызываем метод и через точку пишем название метода.
* Если метод не статический – мы обращаемся к экземпляру класса, у которого мы хотим вызвать метод и через точку пишем название метода.

При переопределении метода все атрибуты доступа, ключевые слова, возвращаемый тип, название метода и входные аргументы **не меняются**. Иначе говоря, при переопределении метода шапка метода не меняется.

1. Какие свойства имеют финальные методы и финальные классы? Как вы думаете, зачем их использовать?

Answer

Методы, помеченные как final не переопределяются.

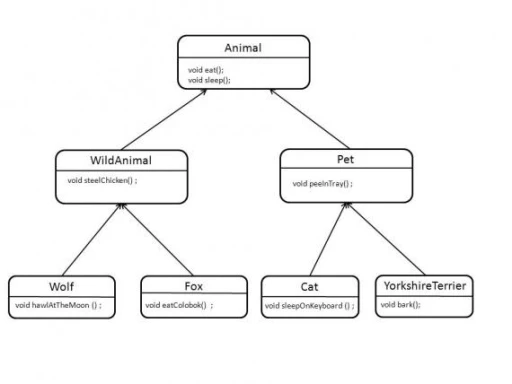
Класс, помеченный как **final**, не поддается наследованию и все его методы косвенным образом приобретают свойство **final**.

Модификатор **final** используется для безопасности кода. То есть программист видит, что стоит final и этот метод или класс не переопределяется у наследников он конечный.

1. Укажите правила приведения типов при наследовании. Напишите примеры явного и неявного преобразования ссылочных типов. Объясните, какие ошибки могут возникать при явном преобразовании ссылочных типов.

Answer

Приводиться могут классы только классы, которые наследуются друг от друга



Пример не явного приведения (расширяющее):

Animal animalCat = **new** Cat();

Animal animalDog = **new** YorkshireTerrier();

Объяснение:

Мы создали ссылку класса Animal и в ссылку положили объект класса Cat и мы создали ссылку класса Animal и в ссылку положили объект класса YoukshireTerrier.

Таким образом мы расширили ссылки animalCat и animalDog. Они ссылаются на объекты Cat и YorkshireTerrier. При таком приведении мы не можем через ссылку animalCat/animalDog вызвать методы, которые есть в Cat/Dog, но которых нету в Animal.

Пример явного приведения (сужающее):

Cat cat =(Cat)animalCat;

YorkshireTerrier dog = (YorkshireTerrier) animalDog;

Объяснение:

Мы явно указали к какому типу хотим привести данный объект. Теперь мы можем вызывать методы, которые есть у объектов класса Cat или объекта класса YorkshireTerrier.

Если привести классы, которые не наследуются друг от друга вылезет ошибка **ClassCastException**.

Animal animalCat = **new** Cat();

YorkshireTerrier dog = (YorkshireTerrier) animalCat;

Она вылезет так как Cat и YorkshireTerrier два разных класса.

1. Что такое объект класса Class? Чем использование метода getClass() и последующего сравнения возвращенного значения с Type.class отличается от использования оператора instanceof?

Answer

Смотреть раздел 4 вопрос 11

1. Укажите правила переопределения методов equals(), hashCode() и toString().

Answer

Смотреть раздел 4 вопрос 12

1. Что такое абстрактные классы и методы? Зачем они нужны? Бывают ли случаи, когда абстрактные методы содержат тело? Можно ли в абстрактных классах определять конструкторы? Могут ли абстрактные классы содержать неабстрактные методы? Можно ли от абстрактных классов создавать объекты и почему?

Answer

***Абстрактный класс*** — это максимально абстрактная, о-о-о-чень приблизительная «заготовка» для группы будущих классов и класс который объявлен ключевым словом **abstract**.

***Абстрактный метод*** — это метод, который объявлен ключевым словом **abstract** и не имеющий тела метода.

Абстрактный класс нужен для максимально абстрактного описания нужной нам сущности. Абстрактный класс нельзя создать как объект, от него надо наследоваться.

У абстрактного класса могут быть абстрактные методы (без тела) и обычные методы (с телом).

У абстрактного класса нет конструкторов, так как нельзя создать экземпляр абстрактного класса (это шаблон).

1. Что такое интерфейсы? Как определить и реализовать интерфейс в java-программе? Укажите спецификаторы, которые приобретают методы и поля, определенные в интерфейсе. Можно ли описывать в интерфейсе конструкторы и создавать объекты? Можно ли создавать интерфейсные ссылки и если да, то на какие объекты они могут ссылаться?

Answer

***Интерфейс*** – это набор методов, которые описывают поведение класса.

Если методы должны быть в интерфейсе реализованы в шапке метода необходимо указать ключевое слово **default** иначе метод не должен быть реализован.

**Пример:**

**public** **interface** Swimmable {

**public** **default** **void** swim() {

System.***out***.println("Плыви!");

}

**public** **void** eat();

**public** **void** run();

}

Интерфейс может содержать поля, но они автоматически являются статическими (**static**) и неизменными (**final**). Все методы и переменные неявно объявляются как **public**.

Класс, который собирается использовать определённый интерфейс, использует ключевое слово **implements**.

Если класс содержит интерфейс, но не полностью реализует определённые им методы, он должен быть объявлен как **abstract**.

Интерфейсы — это не классы. С помощью ключевого слова **new** нельзя создать экземпляр интерфейса:

х = **new** List(...); // Нельзя!

Но можно объявлять интерфейсные переменные:

List<String> catNames; // Можно!

При этом интерфейсная переменная должна ссылаться на объект класса, реализующего данный интерфейс.

List<String> catNames = **new** ArrayList<>();

**Пример:**

**public** **interface** SimpleInterface {

String getClassName();

**int** getAge();

}

**public** **class** Cat **extends** Animal **implements** SimpleInterface {

@Override

**public** String getClassName() {

**return** "Cat";

}

@Override

**public** **int** getAge() {

**return** 5;

}

}

1. Для чего служит интерфейс Clonable? Как правильно переопределить метод clone() класса Object, для того, что объект мог создавать свои адекватные копии?

Answer

Интерфейс **Cloneable** – это так называемый интерфейс-маркер, который не содержит никаких методов. Он используется, чтобы маркировать (помечать) некоторые классы.

Если разработчик класса считает, что объекты класса можно клонировать, он помечает класс этим интерфейсом (наследует (**implements**) класс от Cloneable).

Если разработчика не устраивает стандартная реализация метода clone, он должен написать свою, которая будет создавать дубликат объекта правильным образом.

Object.clone() выбрасывает исключение CloneNotSupportedException при попытке клонировать объект, не реализующий интерфейс Cloneable.

Метод clone() в родительском классе Object является protected, поэтому желательно переопределить его как public. Реализация по умолчанию метода Object.clone() выполняет неполное/поверхностное (shallow) копирование. Рассмотрим пример:

**Пример 1. Поверхностное клонирование**

|  |  |
| --- | --- |
| **public** **class** Car **implements** Cloneable {  **private** String name;  **private** Driver driver;  **public** Car(String name, Driver driver) {  **this**.name = name;  **this**.driver = driver;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Driver getDriver() {  **return** driver;  }  **public** **void** setDriver(Driver driver) {  **this**.driver = driver;  }  @Override  **public** Car clone() **throws** CloneNotSupportedException {  **return** (Car) **super**.clone();  }  } | **public** **class** Driver **implements** Cloneable {  **private** String name;  **private** **int** age;  **public** Driver(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  @Override  **public** Driver clone() **throws** CloneNotSupportedException {  **return** (Driver) **super**.clone();  }  } |
| **public** **class** CloneCarDemo {  **public** **static** **void** main(String[] args) **throws** CloneNotSupportedException {  Car car = **new** Car("Грузовик", **new** Driver("Василий", 45));  Car clonedCar = car.clone();  System.***out***.println("Оригинал:\t" + car);  System.***out***.println("Клон: \t" + clonedCar);  Driver clonedCarDriver = clonedCar.getDriver();  clonedCarDriver.setName("Вася");  System.***out***.println("Оригинал после изменения имени водителя:\t" + car);  System.***out***.println("Клон после изменения имени водителя: \t\t" + clonedCar);  }  } | |

В этом примере клонируются объект класса Car. Клонирование выполняется поверхностное - новый объект clonedCar содержит ссылку на тот же объект класса Driver, что и объект car. Если вас это не устраивает, то необходимо самим написать "глубокое" клонирование - создать новый объект класса Driver. Перепишем метод clone() класса Car:

**Пример 2. Глубокое клонирование**

@Override

**public** Car clone() **throws** CloneNotSupportedException {

Car newCar = (Car) **super**.clone();

Driver driver = **this**.getDriver().clone();

newCar.setDriver(driver);

**return** newCar;

}

**Пример 3. Конструктор копирования**

Еще один вариант клонирования объекта - это конструктор копирования. Создается конструктор, принимающий на вход объект того же класса, который необходимо клонировать:

**public** **class** Car **implements** Cloneable {

**private** String name;

**private** Driver driver;

**public** Car(String name, Driver driver) {

**this**.name = name;

**this**.driver = driver;

}

/\*\*

\* Конструктор копирования.

\*

\* **@param** otherCar

\*/

**public** Car(Car otherCar) {

**this**(otherCar.getName(), otherCar.getDriver());

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Driver getDriver() {

**return** driver;

}

**public** **void** setDriver(Driver driver) {

**this**.driver = driver;

}

}

Опять же - пример показывает неглубокое клонирование. Перепишем конструктор для реализации "глубокого" копирования:

**Пример 4.  Конструктор копирования с "глубоким" клонированием**

**public** Car(Car otherCar) **throws** CloneNotSupportedException {

**this**(otherCar.getName(), otherCar.getDriver().clone());

}

1. Для чего служат интерфейсы Comparable и Comparator? В каких случаях предпочтительнее использовать первый, а когда – второй? Как их реализовать и использовать?

Answer

***Интерфейс Comparable*** – это интерфейс в котором находится метод для сравнения объектов (compareTo()). Метод переопределяется в классе, объекты которого предполагается сравнивать и в него передаётся один объект такого же класса. Данный интерфейс служит для ***сравнения двух объектов*** и возвращает значение int (0, 1, -1).

***Интерфейс Comparator*** – это интерфейс в котором находится метод для сравнения объектов (compare()). Метод переопределяется в отдельном классе и в него передаются два объекта одинакового типа. Данный интерфейс служит для ***сравнения и сортировки объектов в коллекции*** и возвращает значение int (0, 1, -1).

# Generic classes and Interfaces, Enums

1. Что такое перечисления в Java. Как объявить перечисление? Чем являются элементы перечислений? Кто и когда создает экземпляры перечислений?

Answer

Для перечислений в Java существует класс Enum.

***Enum*** – это такой же класс, только от него нельзя наследоваться и создавать объекты. Он предназначен для хранения перечислений.

Элементами Enum являются константы (с полями static final type [name]). При создании элемента перечисления пишется только название элемента (большими буквами (типом написания CamelCase)), а далее, если нужно, в скобках указывается значение. Чтобы указать значение в скобках нужно создать поле класса такого же типа и создать конструктор.

Обращение к элементу перечисления происходит путём обращения к статическим переменным, например: [Название Enum]**.** [название константы (элемента перечисления)] - DayOfWeek.***FRIDAY***

**Пример:**

**public** **enum** DayOfWeek {

SUNDAY("Воскресенье"), MONDAY("Понедельник"), TUESDAY("Вторник"), WEDNESDAY("Среда"), THURSDAY("Четверг"), FRIDAY("Пятница"), SATURDAY("Суббота");

**private** String title;

DayOfWeek(String title) {

**this**.title = title;

}

**public** String getTitle() {

**return** title;

}

@Override

**public** String toString() {

**return** "DayOfWeek{" + "title='" + title + '\'' + '}';

}

}

У перечислений есть характерные только для них методы:

* **values()**: возвращает массив из всех хранящихся в Enum значений:

**public** **static** **void** main(String[] args) {

System.***out***.println(Arrays.*toString*(DayOfWeek.*values*()));

}

Вывод:

[DayOfWeek{title='Воскресенье'}, DayOfWeek{title='Понедельник'}, DayOfWeek{title='Вторник'}, DayOfWeek{title='Среда'}, DayOfWeek{title='Четверг'}, DayOfWeek{title='Пятница'}, DayOfWeek{title='Суббота'}]

* **ordinal()**: возвращает порядковый номер константы. Отсчет начинается с нуля:

**public** **static** **void** main(String[] args) {

**int** sundayIndex = DayOfWeek.***SUNDAY***.ordinal();

System.***out***.println(sundayIndex);

}

Вывод:

*0*

* **valueOf()**: возвращает объект Enum, соответствующий переданному имени:

**public** **static** **void** main(String[] args) {

DayOfWeek sunday = DayOfWeek.*valueOf*("SUNDAY");

System.***out***.println(sunday);

}

Вывод:

DayOfWeek{title='Воскресенье'}

1. Можно ли самостоятельно создать экземпляр перечисления? А ссылку типа перечисления? Как сравнить, что в двух переменных содержится один и тот же элемент перечисления и почему именно так?

Answer

Объекты Enum класса создать нельзя.

Экземпляр перечисления (константа) создаётся следующим образом: смотреть вопрос 1.

В ссылку типа перечисления (Enum) создать можно и инициализировать её экземпляром перечисления (константой).

В Enum нельзя создать две одинаковые константы, но значения в константах могут быть одинаковыми. Сравниваются константы, в зависимости, что нам необходимо сравнить, ссылки или значения, если нам нужно сравнить ссылки (то есть сравнить участки памяти куда указывают константы) нам нужно воспользоваться оператором сравнения (==), если нам нужно сравнить значения констант, нам нужно воспользоваться методом equals().

1. Что такое анонимные классы?

Answer

***Анонимные классы*** - это классы, что не имеют имени и их создание происходит в момент инициализации объекта.

Рассмотрим небольшой пример. Предположим, что у нас есть интерфейс, который необходимо реализовать. Для реализации можно создать отдельный класс, в нём реализовать интерфейс, далее создать объект на основе класса и через объект ссылаться на методы классы.

Звучит логично, но не оправдано с точки зрения количества действий. Куда проще создать объект и для него сразу выполнить создание анонимного класса.

Пример создания анонимного класса:

Auto car = **new** Auto() {

**void** flyingMoto () {

System.***out***.print ("Летающий мотоцикл");

}

};

1. Что такое параметризованные классы? Для чего они необходимы? Приведите пример параметризованного класса и пример создания объекта параметризованного класса? Объясните, ссылки какого типа могут ссылаться на объекты параметризованных классов? Можно ли создать объект, параметризовав его примитивным типом данных?

Answer

***Параметризованные классы*** – это классы-коллекции или интерфейсы, которые ссылаются на объект класса-коллекции, такие как ArrayList (класс-коллекция) или List (интерфейс), в которые передаются в треугольных скобках (<>) название классов, объекты которых должна содержать коллекция. Если класс не передаётся, то по умолчанию - это Object.

Параметры, которые передаются в треугольных скобках называются – ***Generic***.

Пример:

List<String> list = **new** ArrayList<>();

В качестве параметра можно передать только название класса. Примитивный тип данных не принимается.

Если передать в качестве параметра <?> - это означает, что коллекция может принимать любые объекты любых классов.

List<?> list = **new** ArrayList<>();

Если передать в качестве параметра <? **extends** Animal - /\*название класса\*/> - это означает что коллекция принимает объекты класса Animal и наследников этого класса.

List<? **extends** Animal> list = **new** ArrayList<>();

Если передать в качестве параметра <? **super** Animal> - это означает что коллекция будет принимать объекты классов родителей класса Animal.

List<? **super** Animal> list = **new** ArrayList<>();

# Exceptions and Errors

1. Что для программы является исключительной ситуацией? Какие способы обработки ошибок в программах вы знаете?

Answer

***Исключительная ситуация*** – это ситуация, в которой может возникнуть ошибка или непредвиденная ситуация.

Обработка исключений в Java основана на использовании в программе следующих ключевых слов:

* **try** – определяет блок кода, в котором может произойти исключение;
* **catch** – определяет блок кода, в котором происходит обработка исключения;
* **finally** – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока try.

Эти ключевые слова используются для создания в программном коде специальных обрабатывающих конструкций: try{}catch, try{}catch{}finally, try{}finally{}.

* **throw** – используется для возбуждения исключения;
* **throws** – используется в сигнатуре методов для предупреждения, о том, что метод может выбросить исключение.

Пример:

**public** **static** **void** main(String[] args) **throws** MyException { // предупреждаем с помощью throws, что метод может выбросить исключение MyException

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.***in***));

String s = **null**;

**try** { // в блок try заключаем код, в котором может произойти исключение,

s = reader.readLine();

} **catch** (IOException e) { // в блок catch заключаем код по обработке исключения IOException

System.***out***.println(e.getMessage());

} **finally** { // в блоке finally закрываем поток чтения

**try** { // при закрытии потока тоже возможно исключение, например, если он не был открыт, поэтому “оборачиваем” код в блок try

reader.close();

} **catch** (IOException e) { // пишем обработку исключения при закрытии потока чтения

System.***out***.println(e.getMessage());

}

}

**if** (s.equals("")) {

// мы решили, что пустая строка может нарушить в дальнейшем работу нашей программы, например, на результате этого метода нам надо вызывать метод substring(1,2), поэтому мы вынуждены прервать выполнение программы с генерацией своего типа исключения MyException с помощью throw

**throw** **new** MyException("String can not be empty!");

}

}

1. Что такое исключение для Java-программы? Что значит “программа выбросила исключение”? Опишите ситуации, когда исключения выбрасываются виртуальной машиной (автоматически), и когда необходимо их выбрасывать вручную?

Answer

Возникновение ошибок и непредвиденных ситуаций при выполнении программы называют ***исключением***.

Исключительные ситуации, возникающие в программе, можно разделить на две группы:

1. Ситуации, при которых восстановление дальнейшей нормальной работы программы невозможно;
2. Восстановление возможно.

К **первой группе** относят ситуации, когда возникают исключения, унаследованные из класса Error. Это ошибки, возникающие при выполнении программы в результате сбоя работы JVM, переполнения памяти или сбоя системы. Обычно они свидетельствуют о серьезных проблемах, устранить которые программными средствами невозможно. Такой вид исключений в Java относится к неконтролируемым (unchecked) на стадии компиляции.

К этой группе также относят RuntimeException – исключения, наследники класса Exception, генерируемые JVM во время выполнения программы. Часто причиной возникновения их являются ошибки программирования. Эти исключения также являются неконтролируемыми (unchecked) на стадии компиляции, поэтому написание кода по их обработке не является обязательным.

Ко **второй группе** относят исключительные ситуации, предвидимые еще на стадии написания программы, и для которых должен быть написан код обработки. Такие исключения являются контролируемыми (checked). Основная часть работы разработчика на Java при работе с исключениями – обработка таких ситуаций.

1. Приведите иерархию классов-исключений, делящую исключения на проверяемые и непроверяемые. В чем особенности проверяемых и непроверяемых исключений?

Answer

У всех исключений есть общий класс-предок ***Throwable***.

От него происходят две большие группы — ***исключения (Exception)*** и ***ошибки (Error)***.

***Error*** — это критическая ошибка во время исполнения программы, связанная с работой виртуальной машины Java. В большинстве случаев Error не нужно обрабатывать, поскольку она свидетельствует о каких-то серьезных недоработках в коде.

Наиболее известные ошибки: ***StackOverflowError*** — возникает, например, когда метод бесконечно вызывает сам себя, и ***OutOfMemoryError*** — возникает, когда недостаточно памяти для создания новых объектов.

***Exceptions*** — это, собственно, исключения: исключительная, незапланированная ситуация, которая произошла при работе программы.

Это не такие серьезные ошибки, как Error, но они требуют нашего внимания.

Все исключения делятся на 2 вида — проверяемые (**checked**) и непроверяемые (**unchecked**).

* ***проверяемые (checked)***

Все проверяемые исключения происходят от класса Exception.

Компилятор Java знает о самых распространенных исключениях, и знает, в каких ситуациях они могут возникнуть.

Например, он знает, что, если программист в коде считывает данные из файла, может легко возникнуть ситуация, что файл не существует. И таких ситуаций, которые он может заранее предположить, очень много.

Поэтому компилятор заранее проверяет наш код на наличие потенциальных исключений. Если он их найдет, то не скомпилирует код, пока мы не обработаем их или не пробросим наверх.

* ***непроверяемые (unchecked)***

Они происходят от класса RuntimeException.

Разница в том, что этих ошибок компилятор не ожидает.

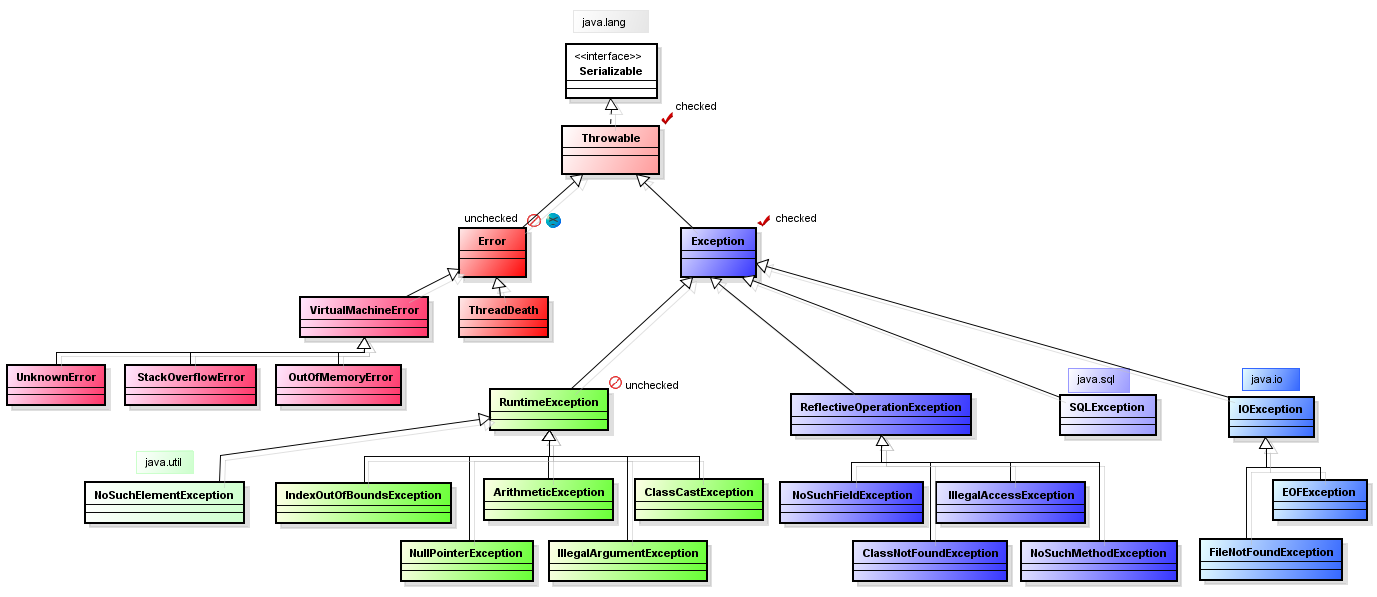
Он как бы говорит: “На момент написания кода я ничего подозрительного не обнаружил, но при его работе что-то пошло не так. Видимо, в коде есть ошибки!”

И это действительно так. Непроверяемые исключения чаще всего являются следствием ошибок программиста.

А компилятор явно не в силах предусмотреть все возможные неправильные ситуации, которые люди могут создать своими руками :) Поэтому он не будет проверять обработку таких исключений в нашем коде.

Ты уже сталкивался с несколькими непроверяемыми исключениями:

* ***ArithmeticException*** - возникает при делении на ноль
* ***ArrayIndexOutOfBoundsException*** -  возникает при попытке обратиться к ячейке за пределами массива.



1. Объясните работу оператора try-catch-finally. Когда данный оператор следует использовать? Сколько блоков catch может соответствовать одному блоку try? Можно ли вкладывать блоки try друг в друга, можно ли вложить блок try в catch или finally? Как происходит обработка исключений, выброшенных внутренним блоком try, если среди его блоков catch нет подходящего? Что называют стеком операторов try? Как работает блок try с ресурсами.

Answer

Смотреть вопросы 1, 2, 3 этого раздела.

Блоку try может принадлежать несколько блоков catch.

1. Укажите правило расположения блоков catch в зависимости от типов перехватываемых исключений. Может ли перехваченной исключение быть сгенерировано снова, и, если да, то как и кто в этом случае будет обрабатывать повторно сгенерированное исключение? Может ли блок catch выбрасывать иные исключения, и если да, то опишите ситуацию, когда это может быть необходимо.

Answer

Если у блока try несколько блоков catch, они должны идти в порядке возрастания.

В примере 1 если в блоке try случится ArithmeticException или NullPointerException, сработает первый или второй catch. Если это будет другое исключение, Java проверит, не Exception ли это, и выполнит третий catch. Всё потому, что любое исключение наследуется от Exception.

Пример 1:

**public** **static** **void** main(String[] args) {

**try** {

//Здесь какой-то код

} **catch** (ArithmeticException e) {

System.***out***.println("Что ты делаешь, говорили же не делить на ноль!");

} **catch** (NullPointerException e) {

System.***out***.println("Кто-то из входящих аргументов равен NULL. Возможно, оба.");

} **catch** (Exception e) {

System.***out***.println("Что-то пошло не так. Но это точно не ArithmeticException и не NPE.");

}

}

Если поменять порядок catch местами, код не соберётся. Компилятор поймёт, что второй и третий catch никогда не сработают, потому что они наследники исключения Exception, которое обрабатывается выше. (Пример 2)

Пример 2

**public** **static** **void** main(String[] args) {

**try** {

//Здесь какой-то код

} **catch** (Exception e) {

System.***out***.println("Что-то пошло не так. Но это точно не ArithmeticException и не NPE.");

} **catch** (NullPointerException e) {

System.***out***.println("Кто-то из входящих аргументов равен NULL. Возможно, оба.");

} **catch** (ArithmeticException e) {

System.***out***.println("Что ты делаешь, говорили же не делить на ноль!");

}

}

1. Когда происходит вызов блока finally? Существуют ли ситуации, когда блок finally не будет вызван? Может ли блок finally выбрасывать исключений? Может ли блок finally выполнится дважды?

Answer

Блок finally вызывается после блока try и catch.

Блок finally сработает, даже если в try или в catch код наткнётся на return. Сначала выполнится finally, а потом программа выйдет из метода.

Единственный случай, когда finally не срабатывает, — критическая ошибка, когда программа вылетела или у неё закончилась выделенная память.