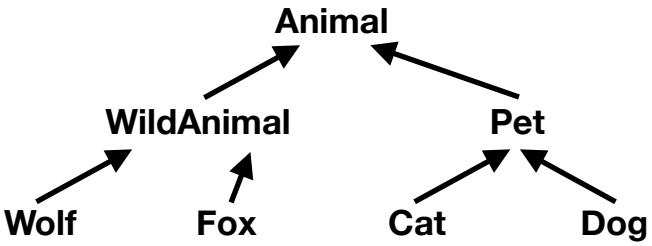
1. Приведение типов при наследовании.

Приведение типов - это преобразование значения переменной одного типа в значение другого типа.

Рассмотрим следующую иерархию:



Класс Cat является производным от класса Pet, который, в свою очередь является производным от класса Animal. Таким образом, такая запись будет верна:

```
Animal animalCat = new Cat();
Animal animalDog = new Dog();
```

Пример выше - это расширяющее (или неявное) приведение: ссылки animalCat и animalDog указывают на объекты Cat и Dog. Через эти ссылки можно получить доступ к любым методам, которые есть в классе Animal, но нельзя получить доступ к специфическим методам классов Cat и Dog.

Сужающее (явное) приведение происходит в другую сторону:

```
Cat cat = (Cat) animalCat;
Dog dog = (Dog) animalDog;
```

Сужающее приведение не всегда возможно, при этом компилятор не укажет на ошибку, но в RunTime будет сгенерирован ClassCastException. Такая ситуация возникнет, например, при преобразовании Cat в Dog:

```
Dog dog = (Dog) animalCat;
```

Для того, чтобы избежать exception, необходимо использовать instanceof:

```
if (animalCat instance Dog) {
        Dog dog = (Dog) animalCat;
}
```

Если объект, на который указывает ссылка animalCat, не является объектом класса Dog, то преобразование не произойдет и exception сгенерирован не будет.

Расширяющее преобразование используется вместе с динамическим связыванием - мы можем создать массив ссылок на Animal, которые будут указывать на разных животных в иерархии, а затем для каждой ссылки вызовем метод sleep, который определен в базовом классе. При этом, мы не сможем для такой ссылки вызвать метод eatRabbit, определенный в классе Fox, чтобы не заставить кота есть кроликов.

2. Класс Object, методы класса Object.

Все классы в Java являются производными класса java.lang.Object, реализуя парадигму Java "все есть объект" (кроме примитивных типов). То есть объект Object может ссылаться на объект любого другого типа, а любому объекту доступны методы класса Object:

Object clone()	создает поверхностную копию вызывающего объекта
boolean equals(Object)	сравнивает содержимое двух объектов, должен быть переопределен для корректной работы с пользовательскими классами
Class getClass()	возвращает класс объекта - объект типа Class, который является набором метаданных класса
void finalize()	вызывается сборщиком мусора автоматически перед удалением объекта; может служить рекомендацией сборщику мусора о начале работы, однако может быть проигнорирована
int hashCode()	возвращает хэш-код объекта - integer, которые является числовым представлением объекта

String toString()	возвращает представление объекта в виде строки, должен быть переопределен для корректной работы с пользовательскими классами

Кроме того, именно в Object определены методы для работы с потоками:

void wait()	поток переходит в режим ожидания
void notify()	просыпается один поток, который ждет на "мониторе" данного объекта
void notifyAll()	просыпаются все потоки, которые ждут на "мониторе" данного объекта

---

3. Назначение оператора throws.

Метод может генерировать исключения, которые сам не обрабатывает, а передает для обработки другим методам, вызывающим данный метод. В этом случае метод должен объявить о таком поведении с помощью ключевого слова throws с перечислением всех возможных типов исключений, чтобы вызывающий метод мог защитить себя от этих исключений. Это необходимо для всех исключений, кроме исключений типа Error, RuntimeException или любых их подклассов. В вызывающем методе должна быть предусмотрена или обработка этих исключений, или последующая передача выше, вызывающему методу. При этом этот метод сам может содержать блоки try-catch, а может и не содержать их. Этот метод может выбрасывать исключение как "вручную" при помощи оператора throw, так и автоматически. Например:

```
private void printResult() {
    String[] userStrings = {"demo", ""};
    for (String userString : userStrings) {
        try {
            System.out.println(parseString(userString));
        } catch (IllegalArgumentException e) {
            System.err.println("iae");
        } catch (NumberFormatException e) {
            System.err.println("pe");
        }
    }
}

private int parseString(String userString) throws NumberFormatException,
        IllegalArgumentException {
    if (userString.isEmpty()) {
        throw new IllegalArgumentException();
    }
    int integerFromString = Integer.parseInt(userString);
    return integerFromString;
}

        /* Output:
        pe
        iae
```

Пример, конечно, искусственный и надуманный, но вполне демонстративный.
В методе parseString могут быть сгенерированы два типа исключений: NumberFormatException и IllegalArgumentException. Обработка этих исключений в этом методе не производится (хотя, конечно, может), а объекты исключений выбрасываются на уровень выше, то есть в метод printResult. При этом объект типа IllegalArgumentException выбрасывается вручную, а объект типа NumberFormatException автоматически (на самом деле точно так же вручную, но методом parseInt(String s, int radix), но для нашего метода parseString этот выбрал можно считать автоматическим).

В методе printResult происходит вызов parseString и обработка исключений, которые могут быть сгенерированы и выброшены из parseString.

---

4. Методы isAlive() и getState() класса Thread.

public final boolean isAlive() возвращает логическое значение true, если поток, для которого он вызван, еще не имеет статус TERMINATED

public Thread.State getState() возвращает текущее состояние потока

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления Thread.State:
NEW — поток создан, но еще не запущен;
RUNNABLE — поток выполняется;
BLOCKED — поток блокирован;
WAITING — поток ждет окончания работы другого потока;
TIMED_WAITING — поток некоторое время ждет окончания другого потока;
TERMINATED — поток завершен.

---

5. Статические поля, статические константные поля.

Поле данных, объявленное в классе как static, является общим для всех объектов класса. Может быть использовано без создания экземпляра класса. Если один объект изменит значение такого поля, то это изменение увидят все объекты. Для объявления статических переменных перед их объявлением используется ключевое слово static.

Константное поле - это поле, значение которого не изменяется в процессе работы программы. Константами принято называть public static final поля, которые сразу проинциализированы. Согласно Code Convention, имена констант должны содержать только большие буквы, разделенные нижним подчеркиванием:

```
public static final FIELD_SIZE = 5;
```

Статические переменные инициализируются во время загрузки класса. К статическим переменным нужно обращаться через имя класса. Локальные переменные, как и абстрактные методы, не могут быть объявлены как static.

---

6. Бинарный поиск, принцип и краткое описание алгоритма.

Бинарный поиск – алгоритм поиска объекта по заданному признаку в множестве объектов, упорядоченных по тому же признаку.

Принцип.
На каждом шаге множество объектов делится на 2 части и в работе остается та часть, где находится искомый объект, деление множества на 2 части продолжается до тех пор, пока элемент не будет найден, либо пока не будет установлено, что его нет в заданном множестве.

Алгоритм бинарного поиска (на отсортированной по возрастанию структуре данных):

1. задаем границы: левая граница = 0вой индекс; правая граница = максимальный индекс структуры данных;
2. берем элемент посередине между границами, сравниваем его с искомым;
3. оцениваем результат сравнения:
   - если искомое равно элементу сравнения, возвращаем индекс элемента, на этом работа алгоритма заканчивается;
   - если искомое больше элемента сравнения, то сужаем область поиска таким образом: левая граница = индекс элемента сравнения + 1;
   - если искомое меньше элемента сравнения, то сужаем область поиска таким образом: правая граница = индекс элемента сравнения - 1;

4. Повторяем шаги 1-3 до тех пор, пока правая граница не станет меньше левой;
5. Возвращаем -1 (до этого шага доходим только в случае, если элемент не был найден);

Время выполнения алгоритма – O(log n);

---

6. Представление о JIT

JIT (Just-In-Time) компиляция – динамическая компиляция, технология увеличения производительности посредством компиляции байт-кода в машинный код во время выполнения программы с применением различных оптимизаций. Благодаря этому достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом. **Основная цель JIT** – приблизиться в производительности к языкам со статической компиляцией (C/C++). С появлением **JIT** Java стала несколько более производительной.

Пример оптимизации **JIT**:

Некий метод исполняется очень часто, поэтому этот метод передается в **JIT**, он компилируется, все последующие вызовы этого метода будет кэшироваться и вызываться без перекомпиляции.

JIT позволяет достигать более высокой скорости выполнения, сохраняя одно из главных преимуществ Java-языка – переносимость. Несмотря на то, что языки со статической компиляцией все равно выигрывают в скорости, статическая компиляция лишает эти языки той переносимости, которая есть в Java.