1. Определение раннего и позднего связывания, механизма переопределения методов

Присоединение метода к телу метода называется связыванием.

Если связывание производится перед запуском программы (например, компилятором), то оно называется ранним связыванием. Тело методы выбирается в зависимости от типа ссылки. Оно применяется при вызове static и final методов (private методы по умолчанию является final): статические методы существуют на уровне класса, а не на уровне экземпляра класса, а final методы нельзя переопределять.

Если связывание производится во время работы программы, то оно называется поздним или динамическим связыванием. Тело метода выбирается в зависимости от фактического типа объекта.

```
public class Shape {
    protected String color = "white";

    public void printMe() {
        System.out.println("I'm a Shape");
    }

    public static void testStatic() {
        System.out.println("Shape.testStatic");
    }
}


public class Circle extends Shape {
    private double radius;

    @Override
    public void printMe() {
        System.out.println("I'm a Circle");
    }

    public static void testStatic() {
        System.out.println("Circle.testStatic");
    }
}


public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.printMe();
        shape.testStatic();
    }
}
```

/* Output:
I'm a Circle
Shape.testStatic

При вызове метода testStatic был задействован код класса Shape - того класса, какого типа ссылка использовалась для вызова. Это ранее связывание.

При вызове метода printMe был задействован код класса Circle, не смотря на то, что он был вызван для ссылки типа Shape. Это позднее связывание.

Переопределение метода (Overriding) - это создание в производном класса метода, который полностью совпадает по имени и сигнатуре с методом базового класса. Переопределение используется для изменения поведения метода базового класса в

производных классах. Для переопределения методов используется аннотация @Override, которая сообщает компилятору о попытке переопределись метод, а компилятор, в свою очередь, проверяет, правильно ли метод переопределен. Когда предопределенный метод вызывается из своего производного класса, он всегда ссылается на свой вариант. Если же требуется получить доступ к методу базового класса, то используется ключевое слово super.

---

2. Переопределение метода toString()

Метод toString в Java используется для предоставления ясной и достаточной информации об объекте (Object) в удобном для человека виде. Правильное переопределение метода toString может помочь в ведении журнала работы и в отладке Java программы, предоставляя ценную и важную информацию. Поскольку toString() определен в классе java.lang.Object и его реализация по умолчанию не предоставляет много информации, всегда лучшей практикой является переопределение данного метода в производном классе. По умолчанию реализация toString создает вывод в виде:

package.class@hashCode

```
public class User {
    private String name;
    private String surname;
    private int birthYear;

    public User(String name, String surname, int birthYear) {
        this.name = name;
        this.surname = surname;
        this.birthYear = birthYear;
    }

    public int getBirthYear() {
        return birthYear;
    }

    @Override
    public String toString() {
        return this.name + " " + this.surname + ", " + getBirthYear() + " г.р.";
    }
}

public class Main {
    public static void main(String[] args) {
        User user1 = new User("Petya", "Lee", 1988);
        System.out.println(user1.toString());
    }
}
```

/*Output:

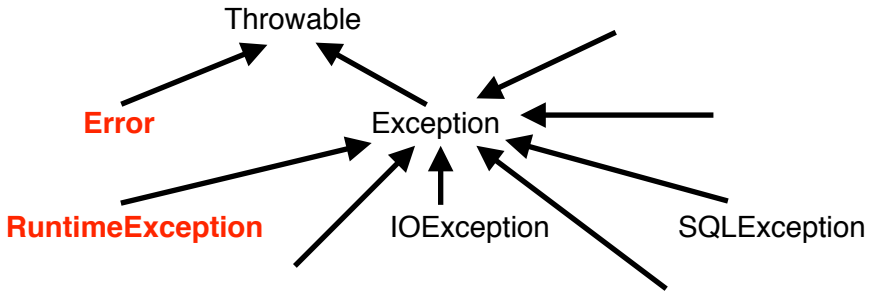
Petya Lee, 1988 г.р.

---

3. Checked и runtime исключения.

Исключительные ситуации (исключения) возникают во время выполнения программы, когда появившаяся проблема не может быть решена в текущем контексте. При возникновении исключения в приложении создается объект, описывающий это исключение, текущий ход выполнения приложения останавливается, и включается механизм обработки исключений. При этом ссылка на объект-исключение передается обработчику исключений, который пытается решить возникшую проблему и продолжить выполнение программы.

Каждой исключительной ситуации поставлен в соответствие некоторый класс, экземпляр которого инициируется при исключительной ситуации. Если подходящего класса не существует, то он может быть создан разработчиком. Все исключения являются наследниками суперкласса Throwable и его подклассов **Error** и **Exception** из пакета java.lang.



Исключительные ситуации типа Error возникают только во время выполнения программы. Такие исключения связаны с серьезными ошибками уровня JVM, к примеру, с переполнением стека, не подлежат исправлению и не могут обрабатываться приложением.

Классы, наследуемые от Exception являются проверяемыми исключениями (checked), за исключением RuntimeException. Возможность возникновения проверяемого исключения отслеживается еще на этапе компиляции. Проверяемые исключения должны быть либо обработаны в методе, который их генерирует, либо выброшены на уровень выше и обработаны в вызывающем методе, за этим следит компилятор.

Класс RuntimeException и порожденные от него классы относятся к непроверяемым исключениям, компилятор не проверяет обработку этих исключений, они могут не обрабатываться. Такие исключения генерируются во время выполнения программы (в runtime) и связаны с ошибками программиста, например, недопустимое приведение типов, выход за пределы массива, деление целого числа на ноль и тд.

---

4. Организация связи между потоками: назначение и работа с методами wait, notify.

Представим ситуацию, что у нас многопоточное приложение. Есть некий класс, выполняющий отправку данных по почте в одном потоке, и есть класс, который эти данные подготавливаются в другом потоке. Перед отправкой данных необходимо как-то связаться с потоком, подготавливающим данные, чтобы узнать, когда данные будут готовы к отправке. Можно, конечно использовать флаг и цикл while, однако это слишком дорого - теряется время CPU.

Для реализации этой задачи Java содержит механизм связи через методы wait(), notify() и notifyAll(). Они реализованы в классе Object, поэтому доступны всем классам.

wait ()	останавливает выполнение текущего потока, переводит его в состояние WAITING и освобождает от блокировки захваченный объект. Вернуть поток в состояние RUNNABLE можно вызовом notify() или notifyAll() из другого потока, либо это произойдет автоматически по истечении срока ожидания, если в режим ожидания он был переведен методом wait(long timeout)
notify ()	возвращает поток, для которого ранее был вызван wait(), в состояние RUNNABLE
notifyAll()	возвращает все потоки для которых ранее был вызван wait(), в состояние RUNNABLE. Первым начнет выполняться самый высокоприоритетный поток.

Методы wait(), notify() и notifyAll() должны обязательно находиться внутри блока synchronized, либо внутри synchronized-метода, иначе будет сгенерирован Exception.

```
public class DataManager {
```

```
    private static final Object monitor = new Object();

    public void sendData() {
        synchronized (monitor) {
            System.out.println("Waiting for data...");
            try {
                monitor.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // continue execution and sending data
            System.out.println("Sending data...");
        }
    }

    public void prepareData() {
        synchronized (monitor) {
            System.out.println("Data prepared");
            monitor.notifyAll();
        }
    }
}
```

---

5. Перегрузка методов: правила перегрузки, разрешение перегрузки.

Java разрешает определение внутри одного класса двух или более методов с одним именем, если списки их параметров различны по количеству и/или типу. Такие методы называются перегруженными. Возвращаемые типы перегруженных методов могут быть различными, но если методы различаются только типом возвращаемого значения, этого недостаточно для перегрузки метода.

Перегрузка реализует «раннее связывание», то есть версия вызываемого метода определяется на этапе компиляции. Методы с одинаковыми именами, но с различными списком параметров и возвращаемыми значениями могут находиться в разных классах одной цепочки наследования и также будут перегруженными. Статические методы могут перегружаться нестатическими, и наоборот.

Механизм перегрузки снижает гибкость кода, однако позволяет избежать ошибок при обращении к перегруженным методам, которые отслеживаются на этапе компиляции. Следует избегать ситуаций, когда компилятор будет не в состоянии выбрать правильный метод, например, из-за неявного приведения типов. Следует также по возможности избегать перегрузку с одинаковым числом параметров, так как это может привести с ошибкам в связи с тем же неявным приведением типов компилятором.

---

6. Quick Sort, принцип и краткое описание алгоритма

Принцип

Опирается на принцип «разделяй и властвуй». Выбирается опорный элемент, это может быть любой элемент. От выбора элемента не зависит корректность работы алгоритма, но в отдельных случаях выбор этого элемента может повысить эффективность работы алгоритма. Оставшиеся элементы сравниваются с опорным и переставляются так, чтобы массив представлял собой последовательность: элементы меньше опорного-равные опорному элементы-элементы больше опорного. Для частей «больше» и «меньше» опорного элемента рекурсивно выполняется та же последовательность операций, если размер этой части составляет больше 1 элемента.

На практике входные данные обычно делят не на 3, а на 2 части. Например, «меньше опорного элемента» и «больше или равны опорному элементу». В общем случае разделение на 2 части эффективнее.

Алгоритм

1. проверяется, что во входном массиве больше 1 элемента, в противном случае алгоритм завершает свое действие;
2. с помощью какой-то схемы разбиения (например, схема разбиения Хоара или Ломуто) элементы в массиве меняются местами и определяется опорный элемент;
3. массив разбивается на 2: «меньше опорного элемента» и «больше или равны опорному элементу»;
4. рекурсивно вызывается этот же алгоритм для частей массива, полученных в пункте 3;

Разбиение Ломуто:

1. последний элемент выбирается опорным элементом, индекс хранится в переменной
2. каждый раз, когда находится элемент меньший или равный опорному, индекс увеличивается, а элемент вставляется перед опорным

Свойства

- время работы: лучшее – O(n*log n), среднее – O(n*log n), худшее – O(n^2);
- затраты памяти – O(log n);
- неустойчивая;
- количество обменов – обычно O(n*log n) обменов;

Плюсы

- в среднем очень быстрая;
- требует небольшое количество дополнительной памяти;

Минусы

- зависит от данных, на плохих данных деградирует до O(n^2);
- может привести к ошибке переполнения стека;
- сложность реализации;
- неустойчива;

---

3. Java Heap: принципы работы, структура

В связи с тем, что Java постоянно развивается, на данный момент существует ряд противоречий в терминологии относительно типов памяти в Java. Например, часть ресурсов термины non-heap и stack приравнивают друг к другу. В своем ответе я подразумеваю разделение памяти в Java на три области: stack, heap, non-heap. Принцип работы памяти heap я описываю на примере применения одного из сборщиков мусора HotSpot VM – Serial GC. Есть и другие сборщики мусора, принцип работы которых отличается в той или иной мере.

Heap (Куча):

- используется для выделения памяти под объекты;
- создание нового объекта происходит в куче;
- объекты кучи доступны разным потокам;
- в куче работает сборщик мусора;
- куча имеет больший размер памяти, чем стек (stack);

Heap делится на 2 части:

- Young Generation Space тоже делится на 2 части:
    - Eden Space – область памяти, в которую попадают только созданные объекты, после сборки мусора эта область памяти должна освободиться полностью, а выжившие объекты перемещаются в Survivor Space
    - Survivor Space – область памяти, которая обычно подразделяется на две подчасти («from-space» и «to-space»), между которыми объекты перемещаются по следующему принципу:
        - «from-space» постепенно заполняется объектами из Eden после сборки мусора;

        - возникает необходимость собрать мусор в «from-space»;
        - работа приложения приостанавливается и запускается сборщик мусора;
        - все живые объекты «from-space» копируются в «to-space»;
        - после этого «from-space» полностью очищается;
        - «from-space» и «to-space» меняются местами («to-space» становится «from-space» и наоборот);
        - все объекты, пережившие определенное количество перемещений между двумя частями Survivor Space перемещаются в Old Generation;
- Old Generation Space – область памяти, в которую попадают долгоживущие объекты, пережившие определенное количество сборок мусора. Работает по принципу перемещения живых объектов к началу «old generation space», таким образом мусор остается в конце (мусор не очищается, поверх него записываются новые объекты), имеется указатель на последний живой объект, для дальнейшей аллокации памяти указатель просто сдвигается к концу «old generation space»;

Non-heap делится на 2 части:

- Permanent Generation – это пул памяти, который содержит интернированные строки, информацию о метаданных, классах, это пространство зарезервировано для классов, статических данных и т.п. Metaspace – замена Permanent Generation в Java 8. Основное различие в том, что Metaspace может динамически расширять свой размер во время выполнения. Размер Metaspace по умолчанию не ограничен.
- Code Cache – используется JVM при JIT-компиляции, здесь кэшируется скомпилированный код.