1. Класс Object

Существует один особый класс Object, определенный в Java. Все остальные классы являются подклассами Object, т.е. Object представляет собой суперкласс для всех остальных классов. Это означает, что ссылочная переменная типа Object может ссылаться на объект любого другого класса. Кроме того, поскольку массивы реализованы в виде классов, переменная типа Object также может ссылаться на любой массив.

Метод	Назначение
`Object clone()`	Создает новый объект, который совпадает с клонируемым объектом
`boolean equals(Object объект)`	Определяет, равен ли один объект другому
`void finalize()`	Вызывается перед удалением неиспользуемого объекта. (Объявлен устаревшим в JDK 9.)
`Class<?> getClass()`	Получает класс объекта во время выполнения
`int hashCode()`	Возвращает хеш-код, ассоциированный с вызывающим объектом
`void notify()`	Возобновляет выполнение потока, ожидающего вызывающий объект
`void notifyAll()`	Возобновляет выполнение всех потоков, ожидающих вызывающий объект
`String toString()`	Возвращает строку, которая описывает объект

Метод	Назначение
`void wait()`	Ожидает другого потока выполнения
`void wait(long миллисекунд)`	
`void wait(long миллисекунд, int наносекунд)`	

Методы getClass(), notify(), notifyAll() и wait() объявлены как final. Остальные методы можно переопределять. Метод equals() сравнивает два объекта. Он возвращает true, если объекты равны, и false в противном случае. Если метод equals переопределен (например, в классе String, Integer и большинстве других встроенных классов Java), он сравнивает содержимое объектов (например, значение строки в String). Точное определение равенства может варьироваться в зависимости от типа сравниваемых объектов. Метод toString() возвращает объект String, содержащий описание исключения. Вызывается оператором println() при выводе объекта Throwable

2. Абстрактный класс, абстрактные методы

Бывают ситуации, когда желательно определить суперкласс, который объявляет структуру заданной абстракции, не предоставляя полные реализации методов. То есть иногда нужно создать суперкласс, определяющий только обобщенную форму, которая будет применяться всеми его подклассами, оставляя каждому подклассу возможность заполнить детали. Такой класс определяет природу методов, подлежащих реализации в подклассах. В Java проблема решается посредством абстрактных методов. Вы можете потребовать, чтобы некоторые методы были переопределены в подклассах, указав модификатор abstract. Иногда их называют методами, подпадающими под ответственность подкласса, потому что они не имеют реализации, указанной в суперклассе. Таким образом, подкласс обязан их переопределить - он не может просто использовать версию, определенную в суперклассе. Для объявления абстрактного метода применяется следующая общая форма:
abstract тип имя(список-параметров); (без тела)
Любой класс, содержащий один или несколько абстрактных методов, тоже должен быть объявлен абстрактным. Объектов абстрактного класса не бывает, т.е. экземпляр абстрактного класса нельзя создать напрямую с помощью операции new. Подобного рода объекты был и бы бесполезными, т.к. абстрактный класс не определен полностью. Кроме того, не допускается объявлять абстрактные конструкторы или абстрактные статические методы. Конкретные методы в абстрактных классах по-прежнему разрешены. Хотя абстрактные классы нельзя задействовать для создания объектов, их можно применять для создания ссылок на объекты, поскольку подход Java к полиморфизму во время выполнения обеспечивается через использование ссылок на суперклассы. Таким образом, должна быть возможность создания ссылки на абстрактный класс, чтобы ее можно было применять для указания на объект подкласса.

3. SOLID и STUPID

SOLID:
S — Single Responsibility Principle — Принцип единственной ответственности. Один класс решает одну задачу. Это значит, что каждый класс должен выполнять только одну четко определенную функцию. Если он решает более одной задачи, это может привести к сложностям в поддержке и расширении кода.
O — Open/Closed Principle — Принцип открытости/закрытости. Программные сущности (классы, модули, функции и т.д.) должны быть открыты для расширения, но закрыты для модификации. Это значит, что мы должны уметь добавлять новую функциональность, не изменяя существующий код.

```
1  public class AreaCalculator {
2      public double calculateArea(Object shape) {
3          double area = 0;
4          if (shape instanceof Circle) {
5              Circle circle = (Circle) shape;
6              area = Math.PI * Math.pow(circle.getRadius(), 2);
7          } else if (shape instanceof Rectangle) {
8              Rectangle rectangle = (Rectangle) shape;
9              area = rectangle.getWidth() * rectangle.getHeight();
10         }
11         return area;
12     }
13 }
```

При добавлении новых фигур нам придется модифицировать метод calculateArea, что нарушает OCP.

```
1  public interface Shape {
2      double calculateArea();
3  }
4
5  public class Circle implements Shape {
6      private double radius;
7
8      public Circle(double radius) {
9          this.radius = radius;
10     }
11
12     @Override
13     public double calculateArea() {
14         return Math.PI * Math.pow(radius, 2);
15     }
16 }
17
18 public class Rectangle implements Shape {
19     private double width;
20     private double height;
21
22     public Rectangle(double width, double height) {
23         this.width = width;
24         this.height = height;
25     }
26
27     @Override
28     public double calculateArea() {
29         return width * height;
30     }
31 }
32
33 public class AreaCalculator {
34     public double calculateArea(Shape shape) {
35         return shape.calculateArea();
36     }
37 }
```

Теперь при добавлении новых фигур (например, Triangle) не нужно изменять AreaCalculator, так как каждая фигура реализует метод calculateArea.

L — Liskov Substitution Principle — Принцип подстановки Барбары Лисков. Производные классы должны заменять свои базовые классы. Это значит, что объекты базовых классов должны быть заменяемы объектами производных классов без изменения ожидаемого поведения программы. То есть, если у нас есть класс A и его подкласс B, то мы должны иметь возможность использовать объект класса B везде, где ожидается объект класса A, и при этом поведение программы останется корректным. Принцип Лисков ориентирован на то, чтобы соблюсти целостность и предсказуемость поведения наследников.

```
public class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}

public class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width;
    }

    @Override
    public void setHeight(int height) {
        this.width = height;
        this.height = height;
    }
}
```

На первый взгляд, может показаться, что Square подходит для наследования от Rectangle, так как квадрат — это частный случай прямоугольника. Однако на практике это нарушает LSP. Если мы попытаемся использовать Square там, где ожидается Rectangle, поведение будет неожиданным:

```
public class Main {
    public static void main(String[] args) {
        Rectangle rect = new Square();
        rect.setWidth(5);
        rect.setHeight(10);

        System.out.println("Expected area = 5 * 10 = 50");
        System.out.println("Actual area = " + rect.getArea());
    }
}
```

Мы ожидаем, что площадь будет равна $5 \cdot 10 = 50$, но из-за логики в Square оба поля width и height устанавливаются в одно и то же значение. Это приводит к неправильному результату. Вместо того чтобы наследовать Square от Rectangle, лучше выделить общий интерфейс:

```
public interface Shape {
    int getArea();
}

public class Rectangle implements Shape {
    protected int width;
    protected int height;

    public Rectangle(int width, int height) {
        this.width = width;
```

```
        this.height = height;
    }

    @Override
    public int getArea() {
        return width * height;
    }
}

public class Square implements Shape {
    private int side;

    public Square(int side) {
        this.side = side;
    }

    @Override
    public int getArea() {
        return side * side;
    }
}
```

I — Interface Segregation Principle — Принцип разделения интерфейса. Клиенты не должны зависеть от интерфейсов, которые они не используют. Это значит, что нужно создавать только небольшие и узконаправленные интерфейсы, не перегруженные ненужными методами.
D — Dependency Inversion Principle — Принцип инверсии зависимостей. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей, но детали должны зависеть от абстракций. Это значит, что код должен быть организован таким образом, чтобы зависимости между компонентами программы были основаны на абстракциях, а не на конкретных реализациях. Таким образом, компоненты легко заменить или изменить без воздействия на другие части системы. Другими словами, высокоуровневые классы (те, которые содержат основную логику) не должны напрямую зависеть от низкоуровневых классов (которые выполняют конкретные действия, такие как работа с базой данных или отправка сообщений). Вместо этого и высокоуровневые, и низкоуровневые классы должны зависеть от интерфейсов или абстрактных классов.
Рассмотрим на примере приложения, которое сохраняет данные пользователя. Допустим, у нас есть класс UserService, который отвечает за регистрацию пользователя. Чтобы сохранить данные пользователя, он обращается к классу MySQLDatabase.

```
public class MySQLDatabase {
    public void saveUser(User user) {
        System.out.println("Saving user to MySQL database");
    }
}

public class UserService {
    private MySQLDatabase database;

    public UserService() {
        this.database = new MySQLDatabase();
    }

    public void registerUser(User user) {
        database.saveUser(user);
    }
}
```

В этом примере UserService зависит от конкретной реализации MySQLDatabase. Это создает несколько проблем: если нужно изменить базу данных (например, на PostgreSQL), придется изменять UserService; этот код сложно тестировать, потому что UserService всегда использует MySQLDatabase. Чтобы решить эти проблемы, вынесем взаимодействие с базой данных в интерфейс Database, а UserService будет зависеть от этого интерфейса, а не от конкретной реализации.

```
public interface Database {
    void saveUser(User user);
}

public class MySQLDatabase implements Database {
    @Override
```

```
7     public void saveUser(User user) {
8         System.out.println("Saving user to MySQL database");
9     }
10 }
11
12 public class PostgreSQLDatabase implements Database {
13     @Override
14     public void saveUser(User user) {
15         System.out.println("Saving user to PostgreSQL database");
16     }
17 }
18
19 public class UserService {
20     private Database database;
21
22     public UserService(Database database) {
23         this.database = database;
24     }
25
26     public void registerUser(User user) {
27         database.saveUser(user);
28     }
29 }
```

Теперь UserService зависит от интерфейса Database, а конкретная реализация передается через конструктор. Здесь UserService принимает интерфейс Database как зависимость. Теперь в UserService можно использовать любую реализацию Database, которая соответствует нужной логике.
STUPID – это набор антипаттернов проектирования, противоположный принципам SOLID, которые направлены на улучшение качества кода. Если принципы SOLID помогают разрабатывать гибкие и масштабируемые системы, то STUPID описывает проблемы, которые могут сделать код сложным в сопровождении и поддержке.

4. Record

Начиная с версии JDK 16, в Java поддерживается специальный класс, который называется записью. Запись разработана с целью обеспечения эффективного и простого в использовании способа для хранения группы значений. Для записей предусмотрено новое контекстно-чувствительное ключевое слово record. Ниже показана общая форма объявления записи:
record имя-записи (список-компонентов) {
// необязательные операторы тела
}
Тело записи является необязательным благодаря тому, что компилятор автоматически предоставит элементы, необходимые для хранения данных, создаст конструктор записи и методы получения, которые обеспечат доступ к данным, а также переопределит методы toString(), equals() и hashCode(), унаследованные от Object. В результате во многих сценариях использования тело не требуется, т.к. само объявление record полностью определяет запись. С записью связан один ключевой аспект: ее данные хранятся в закрытых финальных полях и предоставляются только методы получения. Следовательно, содержащиеся в записи данные являются неизменяемыми. Другими словами, после конструирования записи ее содержимое изменять нельзя. Тем не менее, если запись содержит ссылку на какой-то объект, тогда в этот объект можно вносить изменения, но не разрешено изменять то, на что ссылается ссылка в записи. Таким образом, в терминах Java говорят, что записи поверхностно неизменяемы. Запись не может быть унаследована от другого класса. Однако все записи неявно унаследованы от класса java.lang.Record, в котором переопределяются абстрактные методы equals(), hashCode() и toString(), объявленные в Object. Неявные реализации этих методов создаются автоматически на основе объявления записи. Тип записи не может быть расширен. Таким образом, все объявления записей считаются финальными. Хотя запись не может расширять другой класс, она может реализовывать один или несколько интерфейсов. За исключением equals применять имена методов, определенных в Object, для имен компонентов записи нельзя.

```
1 record Employee(String name, int idNum) {}
2 class RecordDemo {
3     public static void main(String[] args)
4     Employee [] empList = new Employee[2] ;
5     empList[0] new Employee("Doe, John", 1047);
6     empList[1] new Employee("Jones, Robert", 1048);
7     for (Employee e: empList) {
8     System.out.println(e.name(), e.idNum());
9     }
10 }
```

5. Static и final

Ключевое слово final используется для создания неизменяемых переменных, методов, которые нельзя переопределить, и классов, которые нельзя расширить (унаследовать). Класс, помеченный как final, нельзя наследовать.
Ключевое слово static используется для обозначения полей и методов класса, которые принадлежат самому классу, а не его экземплярам. Если поле помечено как static, оно принадлежит классу, а не его экземплярам. Статические методы можно вызывать без создания объекта класса. Они могут работать только с static полями и другими static методами, потому что нет доступа к полям или методам конкретного экземпляра. В Java только вложенные классы могут быть static. Вложенный класс — это класс, определённый внутри другого класса. Когда вложенный класс помечен как static, он становится статическим вложенным классом, и к нему можно обращаться без создания экземпляра внешнего класса.

6. Локальные, анонимные и вложенные классы

Локальные классы — это классы, которые определяются внутри методов, конструкторов или блоков кода. Они видны только внутри того метода, в котором объявлены, и их нельзя использовать вне этого метода. Они удобны для создания вспомогательных классов, когда нужна особая логика только в определённом методе.
Анонимные классы — это классы без имени, которые создаются обычно для одноразового использования. Анонимные классы обычно создаются при реализации интерфейсов или абстрактных классов и используются для определения поведения "на месте". Особенности: не имеют имени и не могут быть переиспользованы; не могут иметь конструкторов, но могут использовать поля и методы; удобны для однократного использования, часто используются в обработчиках событий и при создании простых реализаций интерфейсов или абстрактных классов.
Вложенные классы — это классы, определённые внутри других классов. Существует два типа вложенных классов:
1) Нестатические вложенные классы (или внутренние классы)
Нестатические вложенные классы, также называемые внутренними классами (inner classes), привязаны к экземпляру внешнего класса. Это означает, что внутренний класс имеет доступ ко всем полям и методам внешнего класса, включая приватные. Для создания внутреннего класса требуется экземпляр внешнего класса.
OuterClass outer = new OuterClass();
OuterClass.InnerClass inner = outer.new InnerClass();
2) Статические вложенные классы
Определяются как static и не привязаны к экземпляру внешнего класса. Они могут использоваться как обычные классы и обращаться к static полям и методам внешнего класса, но не имеют доступа к нестатическим полям и методам внешнего класса. Можно создать экземпляр статического вложенного класса без создания экземпляра внешнего класса.
OuterClass.StaticNestedClass nested = new OuterClass.StaticNestedClass();

7. Перегрузка и переопределение

Полиморфизм времени компиляции, также называемый статическим полиморфизмом, осуществляется на этапе компиляции. Он основывается на перегрузке методов и перегрузке операторов (хотя перегрузка операторов в Java не поддерживается). Перегрузка методов (method overloading) — это процесс, при котором в одном классе можно определить несколько методов с одинаковым именем, но с разными параметрами (типом или количеством аргументов). Компилятор решает, какой метод вызвать, основываясь на типах и количестве переданных аргументов на этапе компиляции.
Полиморфизм времени выполнения, также известный как динамический полиморфизм или подтиповое наследование (subtype polymorphism), происходит на этапе выполнения программы. В Java полиморфизм времени выполнения реализуется через переопределение методов (method overriding) и интерфейсы. Когда у объекта есть несколько типов (например, класс и его подкласс), JVM решает, какой метод вызвать, на основе реального типа объекта в момент выполнения, а не типа ссылки, которая ссылается на объект. Переопределение методов происходит, когда подкласс предоставляет свою реализацию метода, который был объявлен в его суперклассе. Это позволяет динамически вызывать нужную реализацию метода на основе типа объекта.
Этап выполнения (или исполнения программы) — это этап, когда программа уже компилирована и выполняется на целевой машине (или JVM в случае Java). Во время выполнения программы фактически происходит обработка данных, выполнение инструкций, взаимодействие с пользователем или другими системами. Полиморфизм времени выполнения (runtime polymorphism) происходит именно на этапе выполнения программы. Это означает, что выбор конкретного метода для вызова происходит в момент работы программы, а не в процессе компиляции. в Java, когда вы используете переопределение методов, JVM решает, какой метод вызвать, основываясь на реальном типе объекта, а не на типе переменной, которая ссылается на этот объект. Это решение принимается во время выполнения, когда объект будет создан и вызван.
Ссылочная переменная типа суперкласса может ссылаться на объект подкласса. Данный факт используется в Java для распознавания вызовов переопределенных методов во время выполнения. А каким образом? Когда переопределенный метод вызывается через ссылку на суперкласс, версия метода, подлежащая выполнению, выясняется на основе типа объекта, на который производится ссылка в момент вызова. Соответственно, такое выяснение происходит во время

выполнения. При ссылке на разные типы объектов будут вызываться разные версии переопределенного метода. Другими словами, именно тип объекта, на который делается ссылка (а не тип ссылочной переменной), определяет, какая версия переопределенного метода будет выполняться. Таким образом, если суперкласс содержит метод, который переопределяется в подклассе, то при ссылке на разные типы объектов через ссылочную переменную типа суперкласса выполняются разные версии метода.

8. Enum

В своей простейшей форме перечисление представляет собой список именованных констант, который определяет новый тип данных и его допустимые значения. Таким образом, объект перечисления может содержать только значения, которые были объявлены в списке. Другие значения не допускаются. Иными словами, перечисление дает возможность явно указывать единственные значения, которые может законно иметь тип данных. Перечисление создается с применением ключевого слова enum. Например, вот простое перечисление, в котором определен перечень различных сортов яблок:

```
1 enum Apple {
2     Jonathan, GoldenDel, RedDel, Winesap, Cortland
3 }
```

Идентификаторы Jonathan, GoldenDel и т.д. называются константами перечисления. Каждая из них неявно объявляется как открытый статический финальный член Apple. Более того, их типом является тип перечисления, в котором они объявлены, в данном случае - Apple. Однако, несмотря на то, что перечисления определяют тип класса, экземпляр перечисления не создается с помощью new. Взамен переменная перечисления объявляется и используется почти так же, как переменная одного из примитивных типов. Все перечисления автоматически содержат в себе два предопределенных метода: values() и valueOf() со следующими общими формами:
public static enum-type [] values()
public static enum-type valueOf(String str)
Метод values() возвращает массив, содержащий список констант перечисления. Метод valueOf() возвращает константу перечисления, значение которой соответствует строке, переданной в аргументе str. В обоих случаях в enum-type указывается тип данного перечисления. Как уже упоминалось, перечисление Java относится к типу класса. Хотя вы не создаете экземпляр перечисления с помощью new, в остальном он обладает теми же возможностями, что и другие классы. Тот факт, что enum определяет класс, придает перечислению в Java необычайную силу. Например, вы можете предоставить ему конструкторы, добавить переменные экземпляра и методы и даже реализовать интерфейсы. Важно понимать, что каждая константа перечисления является объектом своего типа перечисления. Таким образом, в случае определения конструктора для перечисления конструктор будет вызываться при создании каждой константы перечисления.

```
1 enum Apple {
2     Jonathan(10), GoldenDel(9), RedDel(12), Winesap(12), Cortland(8);
3     private int price;
4     Apple(int p) { price = p; }
5     int getPrice() { return price };
6 }
```

В эту версию перечисления Apple добавлены три компонента. Первый - переменная экземпляра price, которая применяется для хранения цены каждого сорта яблок. Второй - конструктор Apple, которому передается цена сорта яблок. Третий - метод getPrice(), возвращающий значение price. Обратите внимание на способ указания аргументов конструктора за счет их помещения в круглые скобки после каждой константы: Jonathan(10), GoldenDel(9), RedDel (12), Winesap(12), Cortland(8); эти значения передаются параметру p конструктора Apple(), который затем присваивает его переменной экземпляра price. Несмотря на невозможность при объявлении перечисления наследовать его от суперкласса, все перечисления автоматически унаследованы от java.lang.Enum. В этом классе определено несколько методов, доступных для использования всеми перечислениями. Есть возможность получить значение, которое указывает позицию константы перечисления в списке констант. Оно называется порядковым номером и извлекается вызовом метода ordinal(). Константу перечисления можно сравнивать на предмет равенства с любым другим объектом, используя метод equals(), который переопределяет метод equals(), определенный в Object. Несмотря на то что метод equals() способен сравнивать константу перечисления с любым другим объектом, эти два объекта будут равны, только если они оба ссылаются на одну и ту же константу внутри того же самого перечисления.

```
1     Apple ap;
2     ap = Apple.RedDel
```

9. Интерфейсы

С помощью ключевого слова interface вы можете полностью абстрагировать интерфейс класса от его реализации. То есть с применением interface можно указать, что класс должен делать, но не как конкретно. После определения интерфейса один или несколько классов могут его реализовать. Для реализации интерфейса включите в определение класса конструкцию implements и затем создайте методы, требуемые интерфейсом.
Интерфейсы синтаксически похожи на классы, но в них отсутствуют переменные экземпляра и, как правило, их методы объявляются без тела. На практике это означает, что вы можете определять интерфейсы, не делая предположений о том, каким образом они реализованы. После определения интерфейс может быть реализован любым количеством классов. Кроме того, один класс может реализовывать любое количество интерфейсов. Для реализации интерфейса класс должен предоставить полный набор методов, требуемых интерфейсом. Однако каждый класс может самостоятельно определять детали собственной реализации. За счет предоставления ключевого слова interface язык Java позволяет в полной мере задействовать аспект полиморфизма "один интерфейс, несколько методов". Если модификатор доступа отсутствует, тогда устанавливается стандартный доступ и интерфейс будет доступным только другим элементам пакета, в котором он объявлен. Когда интерфейс объявлен как pubic, его может использовать код вне пакета, где он объявлен. В таком случае интерфейс должен быть единственным открытым интерфейсом, объявленным в файле, а файл должен иметь то же имя, что и интерфейс. До выхода JDK 8 интерфейс мог определять только "что"; но не "как": В версии JDK 8 ситуация изменилась. Начиная с JDK 8, к методу интерфейса можно добавлять стандартную реализацию. Кроме того, в JDK 8 также добавлены статические методы интерфейса, а начиная с JDK 9, интерфейс может включать закрытые методы. В результате теперь интерфейс может задавать какое-то поведение. Вы можете объявлять переменные как ссылки на объекты, в которых применяется тип интерфейса, а не тип класса. С помощью переменной подобного рода можно ссылаться на любой экземпляр любого класса, реализующего объявленный интерфейс. При вызове метода через одну из таких ссылок корректная версия будет вызываться на основе фактического экземпляра реализации интерфейса, на который осуществляется ссылка. Если класс включает интерфейс, но не полностью реализует методы, требуемые этим интерфейсом, то такой класс должен быть объявлен абстрактным.
С помощью ключевого слова extends один интерфейс может быть унаследован от другого. Синтаксис используется такой же, как и при наследовании классов. Когда класс реализует интерфейс, унаследованный от другого интерфейса, он должен предоставить реализации для всех методов, требуемых интерфейсами в цепочке наследования.
Как объяснялось ранее, до выхода JDK 8 интерфейс не мог определять какую-либо реализацию. Таким образом, во всех предшествующих версиях Java методы, определяемые интерфейсом, были абстрактными и не содержали тела. Это традиционная форма интерфейса, и именно такой вид интерфейса применялся в предыдущих обсуждениях. В выпуске JDK 8 ситуация изменилась за счет добавления к интерфейсам новой возможности, которая называется стандартным методом. Стандартный метод позволяет определить реализацию по умолчанию для метода интерфейса. Другими словами, используя стандартный метод, метод интерфейса может предоставлять тело, а не быть абстрактным. Стандартный метод интерфейса определяется аналогично определению метода в классе. Основное отличие связано с тем, что объявление предваряется ключевым словом **default**. Важно отметить, что добавление стандартных методов не меняет ключевой аспект интерфейса: его неспособность поддерживать информацию о состоянии. Например, интерфейс по-прежнему не может иметь переменные экземпляра. Следовательно, определяющее различие между интерфейсом и классом заключается в том, что класс может поддерживать информацию о состоянии, а интерфейс - нет. Кроме того, по-прежнему нельзя создавать экземпляр интерфейса самого по себе. Интерфейс должен быть реализован классом. Для реализующего класса возможно и распространено определение собственной реализации стандартного метода. Реализация класса имеет приоритет над стандартной реализацией интерфейса.
В версии JDK 8 в интерфейсах появилась возможность определения одного или нескольких **статических методов**. Подобно статическим методам в классе статический метод, определенный в интерфейсе, может вызываться независимо от любого объекта. Таким образом, для вызова статического метода не требуется реализация интерфейса и экземпляр реализации интерфейса. Взамен статический метод вызывается путем указания имени интерфейса, за которым следует точка и имя метода. Вот общая форма:
ИмяИнтерфейса.имяСтатическогоМетода()
Начиная с версии JDK 9, интерфейс способен содержать закрытый метод, который может вызываться только стандартным методом или другим закрытым методом, определенным в том же интерфейсе. Поскольку закрытый метод интерфейса указан как **private**, его нельзя использовать в коде вне интерфейса, в котором он определен. Такое ограничение распространяется и на производные интерфейсы, потому что закрытый метод интерфейса ими не наследуется.

10. ArrayList

Класс ArrayList из коллекции java.util померживает динамические массивы, которые по мере необходимости могут расти. Стандартные массивы в Java имеют фиксированную длину. После создания массивы не могут увеличиваться или уменьшаться, а потому необходимо заранее знать, сколько элементов будет содержать массив. Но иногда точный размер массива не известен вплоть до времени выполнения. По существу экземпляр ArrayList представляет собой массив объектных ссылок **одного типа** переменной длины, т.е. он способен динамически увеличиваться или уменьшаться в размерах при добавлении/удалении элементов.
Элементы ArrayList сохраняют порядок их добавления. Элементы можно получить по индексу с помощью метода list.get(int index). ArrayList может хранить только объекты. Для примитивных типов (например, int, double) используется автоупаковка.

Добавляет элемент метод list.add(type el);
удаляет - list.remove(index);
проверка размера - list.size();
изменить элемент - list.set(1, "Java") (меняем 2-ой элемент на "Java");
проверка элемента на присутствие в массиве - list.contains(el);
В классе ArrayList определены следующие конструкторы: ArrayList()
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)
Первый конструктор строит пустой списковый массив. Второй конструктор создает списковый массивов, который инициализируется элементами коллекции c. Третий конструктор строит списковый массив с указанной в capacity начальной емкостью. Емкость - это размер лежащего в основе массива, который применяется для хранения элементов. По мере добавления элементов в списковый массив емкость автоматически увеличивается.

11. Исключения

Исключение Java представляет собой объект, описывающий исключительное (т.е. ошибочное) состояние, которое произошло внутри фрагмента кода. При возникновении исключительного состояния в методе, вызвавшем ошибку, генерируется объект, представляющий это исключение. Метод может обработать исключение самостоятельно или передать его дальше. Так или иначе, в какой-то момент исключение перехватывается и обрабатывается. Обработка исключений в Java управляется пятью ключевыми словами: try, catch, throw, throws и finally. Операторы программы, которые вы хотите отслеживать на наличие исключений, содержатся в блоке try. Если внутри блока try возникает исключение, тог да оно генерируется. Ваш код может перехватить это исключение (с помощью catch) и обработать его рациональным образом. Системные исключения автоматически генерируются исполняющей средой Java. Для ручной генерации исключения используйте ключевое слово throw. Любое исключение, генерируемое в методе, должно быть указано как таковое с помощью конструкции throws. Любой код, который обязательно должен быть выполнен после завершения блока try, помещается в блок finally. Все типы исключений являются подклассами встроенного класса Throwable. Таким образом, класс Throwable расположен на вершине иерархии классов исключений. Непосредственно под Throwable находятся два подкласса, которые разделяют исключения на две отдельные ветви. Одну ветвь возглавляет класс Exception, используемый для представления исключительных условий, которые должны перехватываться пользовательскими программами. Он также будет служить подклассом для создания собственных специальных типов исключений. Кроме того, у класса Exception имеется важный подкласс, который называется RuntimeException. Исключения такого типа автоматически определяются для разрабатываемых программ и охватывают такие ситуации, как деление на ноль и недопустимое индексирование массивов. Другую ветвь возглавляет класс Error, определяющий исключения, которые не должны перехватываться программой в обычных условиях. Исключения типа Error применяется исполняющей средой Java для указания ошибок, связанных с самой средой. Примером такой ошибки является переполнение стека. Когда исполняющая среда Java обнаруживает ошибку (например попытку деления на ноль), она создает новый объект исключения и затем генерирует это исключение. В результате выполнение класса останавливается, поскольку после генерации исключение должно быть перехвачено обработчиком исключений и немедленно обработано. Стандартный обработчик отображает строку с описанием исключения, выводит трассировку стека от точки, где произошло исключение, и прекращает работу программы.
Хотя стандартный обработчик исключений, предоставляемый исполняющей средой Java, удобен при отладке, обычно вы пожелаете обрабатывать исключение самостоятельно, что дает два преимущества. Самостоятельная обработка, во-первых, позволяет исправить ошибку и, во-вторых, предотвращает автоматическое прекращение работы программы.

```
1  class Exc2 {
2      public static void main(String[] args) {
3          int d, a;
4          try {
5              // Code block to monitor
6              d = 0;
7              a = 42 / d;
8              System.out.println("This will not be displayed.");
9          } catch (ArithmeticException e) {
10             // Catch division by zero error
11             System.out.println("Division by zero.");
12         }
13         System.out.println("After the catch block.");
14     }
15 }
16 /* Result:
17 Division by zero.
18 After the catch block. */
```

Оператор try и его конструкция catch образуют единицу. Область действия catch ограничена операторами, которые относятся к непосредственно предшествующему оператору try. Конструкция catch не может перехватывать исключение, сгенерированное другим оператором try. Для отображения описания исключения в операторе println() нужно

просто передать исключение в качестве аргумента. В некоторых случаях один фрагмент кода может генерировать более одного исключения. Чтобы справиться с ситуацией такого рода, можно указать две или более конструкции catch, каждая из которых будет перехватывать разные типы исключений. При использовании нескольких конструкций catch важно помнить о том, что подклассы исключений должны предшествовать любым из своих суперклассов. Дело в том, что конструкция catch, в которой применяется суперкласс, будет перехватывать исключения указанного типа плюс любых его подклассов. В итоге конструкция catch с подклассом никогда не будет достигнута, если она находится после конструкции catch с суперклассом.
Оператор try может быть вложенным. Каждый раз, когда происходит вход в try, контекст этого исключения помещается в стек. Если внутренний оператор try не имеет обработчика catch для определенного исключения, тогда стек раскручивается, и на предмет совпадения проверяются обработчики catch следующего оператора try. Процесс продолжается до тех пор, пока не будет найдена подходящая конструкция catch либо исчерпаны все вложенные операторы try. Когда задействованы вызовы методов, вложение операторов try может происходить менее очевидным образом. Например, если вызов метода заключен в блок try и внутри этого метода находится еще один оператор try, то оператор try внутри метода будет вложен во внешний блок try, где метод вызывается.
До сих пор перехватывались только те исключения, которые генерируются исполняющей средой Java. Однако программа может генерировать исключение явно с применением оператора throw со следующей общей формой:
throw ThrowableInstance;
Здесь ThrowableInstance должен быть объектом типа Throwable или подклассом Throwable. Поток выполнения останавливается сразу после оператора throw. Ближайший охватывающий блок try проверяется на предмет наличия в нем конструкции catch, соответствующей типу исключения. Если совпадение найдено, то управление передается этому оператору, а если нет, тогда проверяется следующий охватывающий оператор try и т. д. Если соответствующая конструкция catch не найдена, то стандартный обработчик исключений останавливает работу программы и выводит трассировку стека.

```
static void demoproc() {
    try {
        throw new NullPointerException("demonstration"); // Throw a
            NullPointerException with a message
    } catch (NullPointerException e) {
        System.out.println("Caught inside demoproc()."); // Handle exception inside
            demoproc
        throw e; // Rethrow the exception
    }
}
public static void main(String[] args) {
    try {
        demoproc(); // Call demoproc
    } catch (NullPointerException e) {
        System.out.println("Recaught: " + e); // Handle rethrown exception
    }
}
/* Result:
Caught inside demoproc().
Recaught: java.lang.NullPointerException: demonstration */
```

Здесь с применением операции new создается экземпляр NullPointerException. Многие встроенные исключения времени выполнения Java имеют как минимум два конструктора: один без параметров и один принимающий строковый параметр. Когда используется вторая форма, аргумент указывает строку, описывающую исключение. Эта строка отображается, когда объект передается как аргумент в print() или println(). Его также можно получить, вызвав метод getMessage(), который определен в Throwable.
Если метод способен приводить к исключению, которое он не обрабатывает, то метод должен сообщить о таком поведении, чтобы вызывающий его код мог защитить себя от этого исключения. Задача решается добавлением к объявлению метода конструкции throws, где перечисляются типы исключений, которые может генерировать метод. Поступать так необходимо для всех исключений, кроме исключений типа Error, RuntimeException или любых их подклассов.

```
// Uncorrent programm.
class ThrowsDemo {
    static void throwOne() {
    System.out.println("Inside throwOne()");
    throw new IllegalAccessException("demonstration");
    public static void main (String [] args) {
        throwOne();
    }
}
```

Чтобы пример скомпилировался, в него понадобится внести два изменения: нужно объявить, что метод throwOne() генерирует исключение IllegalAccessException; в методе main() должен быть определен оператор try/catch, который перехватывает это исключение.

Ключевое слово finally позволяет создать блок кода, который будет выполняться после завершения блока try/catch и перед кодом, следующим после try/catch. Блок finally будет выполняться независимо от того, сгенерировано исключение или нет. В случае генерации исключения блок finally будет выполняться, даже если исключение не соответствует ни одной конструкци и catch. Конструкция finally является необязательной. Тем не менее, для каждого оператора try требуется хотя бы одна конструкция catch или finally.
Внутри стандартного пакета java.lang определено несколько классов исключений Java. Наиболее общие из них являются подклассами стандартного типа RuntimeException, такие исключения не нужно включать в список throws любого метода. На языке Java они называются непроверяемыми исключениями, потому что компилятор не проверяет, обрабатывает метод подобные исключения или же генерирует их. Хотя встроенные исключения Java обрабатывают наиболее распространенные ошибки, вполне вероятно, что вы захотите создать собственные типы исключений, которые подходят для ситуаций, специфичных для ваших приложений. Делается это довольно легко: нужно определить подкласс Exception (является подклассом Throwable). Вашим подклассам фактически ничего не придется реализовывать - одно их существование в системе типов позволяет использовать их как исключения. В самом классе Exception никаких методов не определено. Разумеется, он наследует методы, предоставляемые Throwable. Иногда лучше переопределить метод toString() и вот почему: версия toString(), определенная в классе Throwable, сначала отображает имя исключения, за которым следует двоеточие и ваше описание. Переопределив toString(), вы можете запретить отображение имени исключения и двоеточия, сделав вывод более чистым, что желательно в некоторых случаях.

```
// This program creates a custom type of exception.
class MyException extends Exception {
    private int detail;
    // Constructor to initialize the detail field
    MyException(int a) {
        detail = a;
    }
    // Override the toString() method to provide a custom string representation
    public String toString() {
        return "MyException[" + detail + "]";
    }
}
class ExceptionDemo {
    // Method that throws MyException if the input is greater than 10
    static void compute(int a) throws MyException {
        System.out.println("Calling compute(" + a + ")");
        if (a > 10) {
            throw new MyException(a); // Throw custom exception
        }
        System.out.println("Normal termination"); // Normal execution path
    }
    public static void main(String[] args) {
        try {
            compute(1);  // Call compute with a valid value
            compute(20); // Call compute with an invalid value (throws exception)
        } catch (MyException e) {
            System.out.println("Caught exception: " + e); // Handle custom exception
        }
    }
}
/* Result:
Calling compute(1)
Normal termination
Calling compute(20)
Caught exception: MyException [20] */
```

Средство множественного перехвата позволяет перехватывать два или более исключений одной и той же конструкцией catch. Для применения множественного перехвата необходимо объединить все типы исключений в конструкции catch с помощью операции "ИЛИ" (catch Exception1 | Exception2).
Checked Exceptions проверяются (компилятор Java проверяет наличие обработчиков ещё на этапе компиляции кода, до его выполнения. Если проверяемое исключение не обработано компилятор выдаст ошибку) компилятором во время компиляции. Примеры: IOException, ClassNotFoundException.
Unchecked Exceptions не проверяются компилятором во время компиляции. Они возникают в результате ошибок программирования, таких как попытка деления на ноль, доступ к неинициализированному объекту, выход за пределы массива и т. д. Разработчик может, но не обязан, обрабатывать эти исключения.
Errors представляют собой серьёзные проблемы, которые нельзя обработать в приложении. Обычно это проблемы, связанные с ресурсами JVM (например, нехватка памяти). Errors являются подклассами java.lang.Error и, как правило, не должны обрабатываться разработчиком, так как они указывают на серьёзные сбои, которые требуют исправления на уровне JVM или системы.