Принципы SOLID

1. **Single Responsibility Principle (Принцип единственной ответственности)**
   - На каждый объект возлагается одна обязанность, полностью инкапсулированная в класс. Все сервисы класса направлены на обеспечение этой обязанности. Такие классы всегда будет просто изменять, если это понадобится, потому что понятно, за что класс отвечает, а за что – нет. То есть можно будет вносить изменения и не бояться последствий – влияния на другие объекты. А еще подобный код гораздо проще тестировать, ведь вы покрываете тестами одну функциональность в изоляции от всех остальных.

2. **Open-Closed Principle (Принцип открытости/закрытости)**
   - Программные сущности (классы, модули, функции и т.п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что должна быть возможность изменять внешнее поведение класса, не внося физические изменения в сам класс. Следуя этому принципу, классы разрабатываются так, чтобы для подстройки класса к конкретным условиям применения было достаточно расширить его и переопределить некоторые функции.

```java
// Пример нарушения
class Animal {
    private String name;
    private int weight;

    public Animal(String name, int weight) {...}
}

// Вариант исправления
class Animal {
    private String name;
    private int weight;

    public Animal(String name, int weight) {...}

    public String getName() { return name; }
    public int getWeight() { return weight; }

    public void setName(String name) { this.name = name; }
    public void setWeight(int weight) { this.weight = weight; }
}
```

3. **Liskov Substitution Principle (Принцип подстановки Лисков)**
   - Объекты в программе можно заменить их наследниками без изменения свойств программы. Это означает, что подклассы должны переопределять методы базового класса так, чтобы не нарушалась функциональность с точки зрения клиента. То есть, если разработчик расширяет ваш класс и использует его в приложении, он не должен изменять ожидаемое поведение переопределенных методов. Помогает избежать дублирования кода.

```java
// Пример нарушения
public class Main {
    public static void main(String[] args) {
        Object object = new String();
        System.out.println(object.toString());
    }
}
// Вариант исправления
public class Main {
    public static void main(String[] args) {
        Object object = new String();
        System.out.println(object);
    }
}
```

4. **Interface Segregation Principle (Принцип разделения интерфейса)**
   - Слишком «толстые» интерфейсы необходимо разделять на более мелкие и специфические, чтобы клиенты мелких интерфейсов знали только о методах, необходимых в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

```java
// Пример нарушения
interface Animal {
    void run();
    void jump();
    void eat();
}
```

```java
// Вариант исправления
interface Runnable {
    void run();
}
interface Jumpable {
    void jump();
}
interface Eatable {
    void eat();
}
```

5. **Dependency Inversion Principle (Принцип инверсии зависимостей)**
   - Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

```
// Пример нарушения
class Car {
    private Wheel[] wheels;
    public int countWheels = 4;

    public Car() {
        wheels = new Wheel[countWheels];
    }
}
// Вариант исправления
class Car {
    private Wheel[] wheels;

    public Car(int countWheels) {
        wheels = new Wheel[countWheels];
    }
}
```

Переопределение методов

- Переопределить можно только методы, описанные в родительском классе.
- Отсутствие аннотаций не говорит о том, что метод перегружен, а не переопределён.

```
class MyClass /*extends Object*/ {
    @Override
    public String toString() { return "MyClass {}"; }

    public int hashCode() { return super.hashCode(); }

    @Override
    protected void save() {}
}
```

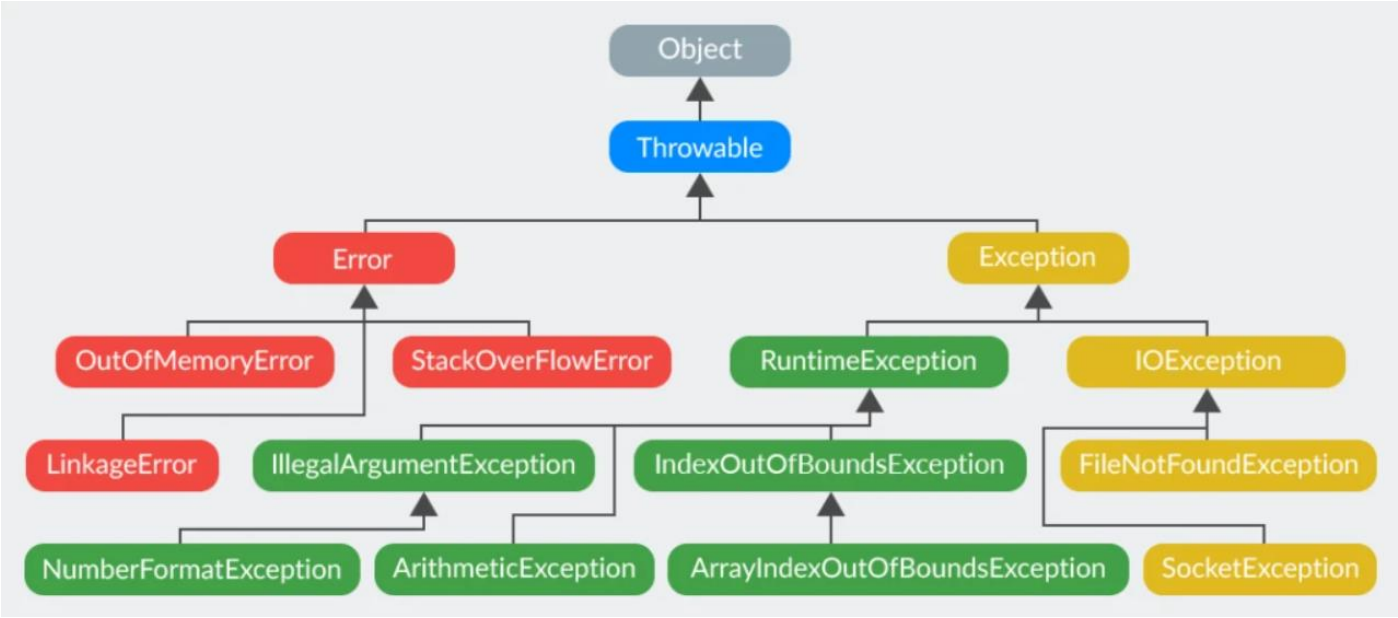
- В данном примере переопределены методы `toString()` и `hashCode()`, т.к. метод `save()` не определён в классе `Object`, от которого по умолчанию наследуется `MyClass`.

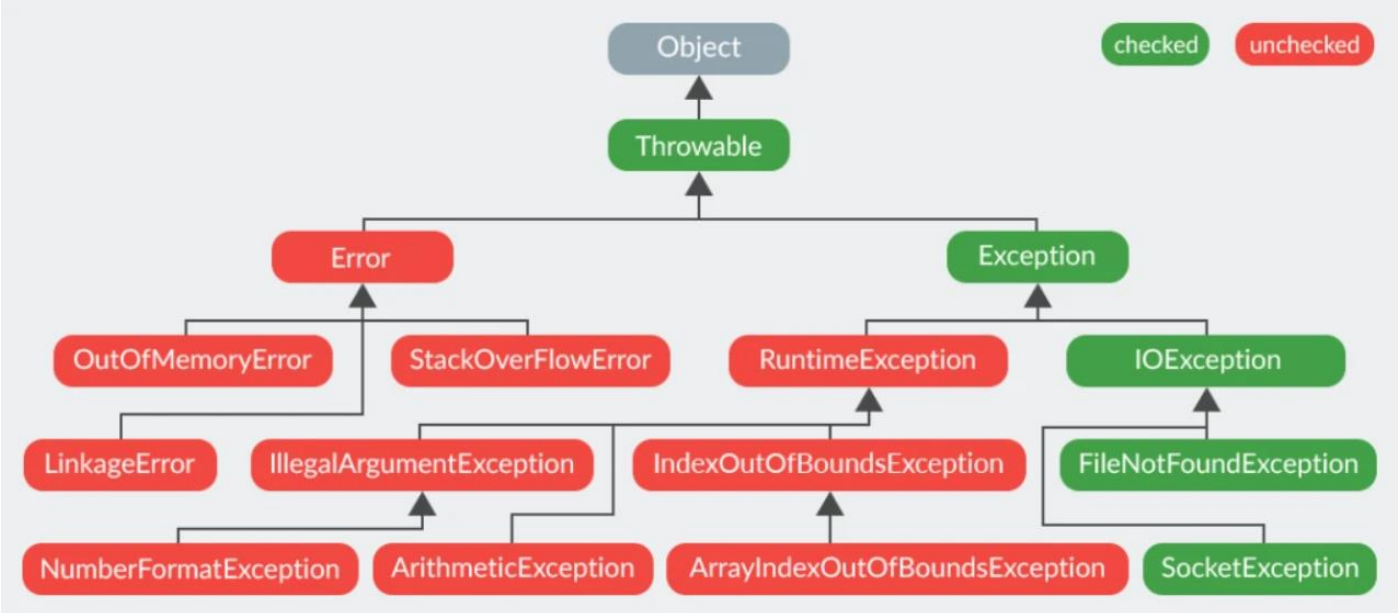
Пример создания абстрактного класса без `extends`

```
abstract class Animal {
    abstract void jump();

    Animal animal = new Animal() {
        @Override
        void jump() {
            // Doing something
        }
    };
}
```

Исключения





- Пробрасывать (`throw`) или обрабатывать (`try-catch`) можно любые исключения и ошибки, но необходимо это только для проверяемых исключений.
- И пробрасывать (`throw`) и обрабатывать (`try-catch`) ошибки/исключения нельзя.
- `throws` выводит в `System.out` сообщение с информацией об ошибке/исключении.
- `try-catch` позволяет прописать внутри `try` обрабатываемое исключение, а внутри `catch` – действия при его возникновении. Есть также необязательная часть `finally`, которая выполняется в любом случае (независимо от результатов `try`).
- Внутри `catch` следует писать максимально частный случай ошибки/исключения (тот вид, на выявление которого направлен `try-catch`).

Функциональное программирование

```
class Car {
    @Resource
    private Wheel[] wheels;
    public int countWheels = 4;
}
```

- Написать код, который выводит имена всех полей класса, отмеченных аннотацией `@Resource`:

```
public class Main {
    public static void main(String[] args) {
        Field[] fields = Car.class.getDeclaredFields();
        for(Field field : fields)
            if(field.getAnnotation(Resource.class) != null)
                System.out.println(field.getName());
    }
}
```