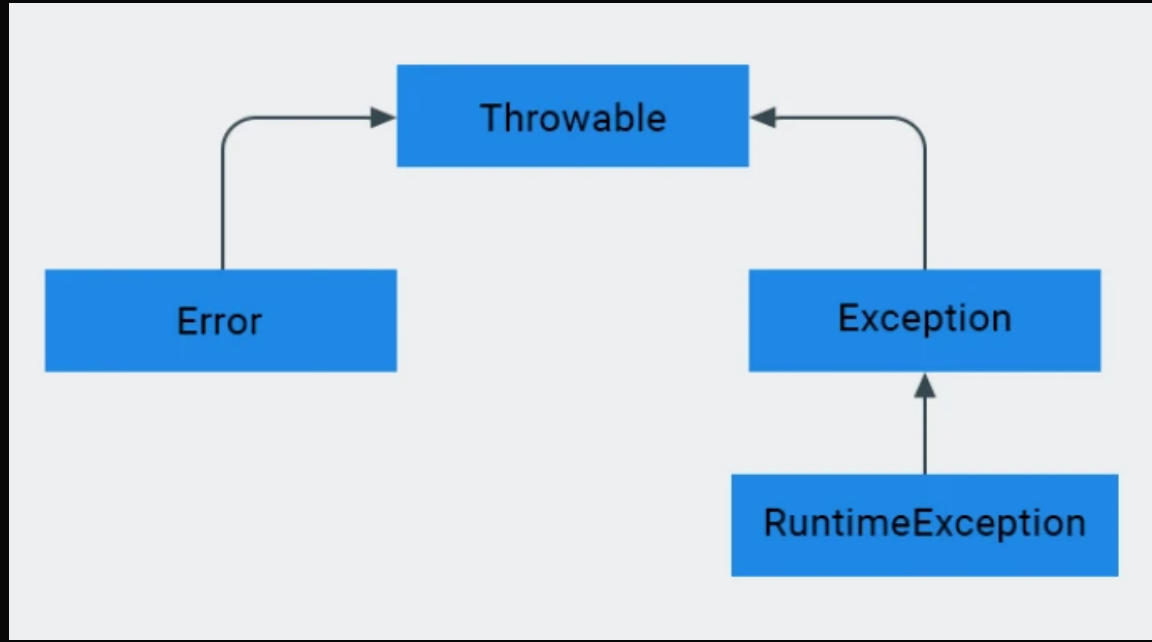
Базовые классы исключений

Все исключения делятся на **4 вида**, которые на самом деле являются классами, унаследованными друг от друга.



1. Класс **Throwable**

   Самым базовым классом для всех исключений является класс **Throwable**. В классе **Throwable** содержится код, который записывает текущий стек-трейс вызовов функций в массив.

2. Класс **Error**

   Следующим классом исключений является класс **Error** — прямой наследник класса **Throwable**. Объекты типа **Error** (и его классов-наследников) создает Java-машина в случае каких-то серьезных проблем.

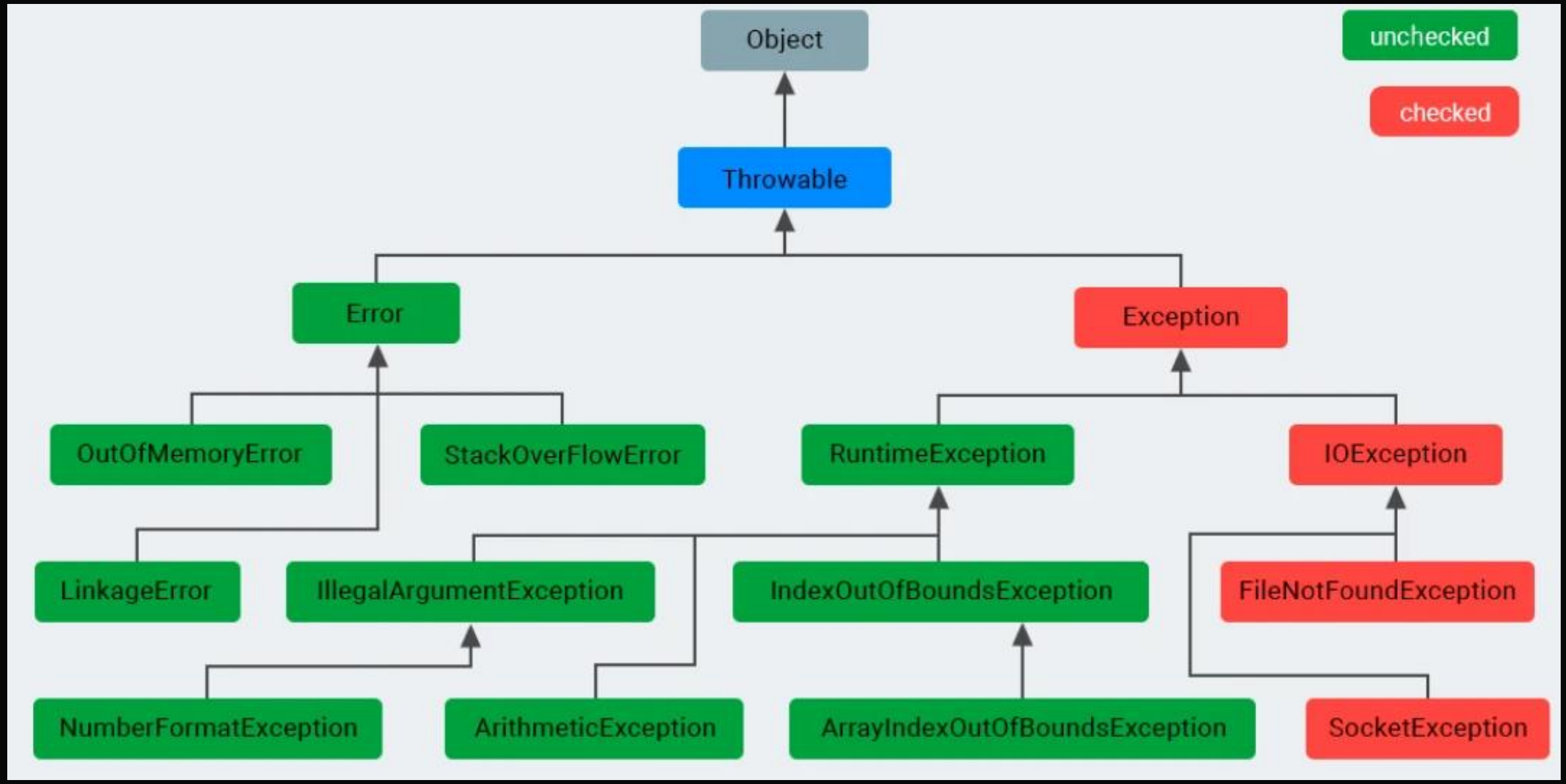
3. Класс **Exception**

   Исключения типа **Exception** (и **RuntimeException**) — это обычные ошибки, которые возникают во время работы многих методов. Цель каждого выброшенного исключения — быть захваченным тем блоком **catch**, который знает, что нужно сделать в этой ситуации.

4. Класс **RuntimeException**

   RuntimeException — это разновидность (подмножество) исключений **Exception**. Можно даже сказать, что **RuntimeException** — это облегченная версия обычных исключений (**Exception**): на такие исключения налагается меньше требований и ограничений

Виды исключений



Базовый шаблон

Базовый шаблон генерации исключения

```
// код, который может вызвать исключение
throw new ExceptionType("Сообщение об ошибке");
```

Базовый шаблон обработки исключения

```
try {
    // код, который может вызвать проверяемое исключение
} catch (CheckedExceptionType e) {
    // код для обработки исключения
} finally {
    // код, который будет выполнен в любом случае
}
```

Ключевое слово **throws** используется для указания, что метод может выбросить проверяемое исключение. Тип исключения указывается после ключевого слова **throws**.

Если метод может выбросить несколько **проверяемых исключений**, вы можете указать их все, разделив запятыми

```
public void methodName() throws CheckedExceptionType1, CheckedExceptionType2 {
    // код метода
```

Полезные методы

1. getMessage(). Возвращает сообщение об ошибке, которое было передано в конструктор исключения.

Это полезно для получения описания ошибки.

```
public void getMessage() {  no usages
    try {
        int result = 10 / 0;
    } catch (ArithmeticException e) {
        System.out.println("Ошибка: " + e.getMessage()); // Выведет: / by zero
    }
}
```

2. getCause(). Возвращает причину исключения (другое исключение, которое вызвало текущее). Это полезно, если исключение является оберткой для другого исключения.

```
public void getCause() {  no usages
    try {
        throw new RuntimeException("Ошибка", new IOException("Причина"));
    } catch (RuntimeException e) {
        System.out.println("Причина: " + e.getCause());
    }
}
```

Полезные методы

3. printStackTrace(). Выводит трассировку стека (stack trace) в стандартный поток ошибок (System.err). Это помогает понять, где именно произошла ошибка.

```
public void printStackTrace() {  no usages
    try {
        int result = 10 / 0;
    } catch (ArithmeticException e) {
        e.printStackTrace(System.out);
    }
}
```

4. getStackTrace(). Возвращает массив элементов StackTraceElement, которые представляют собой трассировку стека. Это полезно для программного анализа стека вызовов.

```
public void getStackTrace() {  no usages
    try {
        int result = 10 / 0;
    } catch (ArithmeticException e) {
        StackTraceElement[] stackTrace = e.getStackTrace();
        for (StackTraceElement element : stackTrace) {
            System.out.println(element);
        }
    }
}
```

Полезные методы

5. toString(). Возвращает строковое представление исключения, включая имя класса и сообщение об ошибке.

```
public void toStringMeth() {  no usages
    try {
        int result = 10 / 0;
    } catch (ArithmeticException e) {
        System.out.println(e.toString());
    }
}
```

6. initCause(Throwable cause). Устанавливает причину исключения. Это полезно, если вы создаете собственное исключение и хотите указать, что его вызвало.

```
public void initCause() {  no usages
    try {
        IOException ioException = new IOException("Ошибка ввода-вывода");
        RuntimeException runtimeException = new RuntimeException();
        runtimeException.initCause(ioException);
        throw runtimeException;
    } catch (RuntimeException e) {
        System.out.println(e.getCause());
    }
}
```

Блок finally

В программировании на Java часто возникает ситуация, когда необходимо обеспечить выполнение определенного кода, независимо от того, произошло исключение в блоке try или нет. Для этого существует специальный блок кода — **finally**.

Чаще всего он используется для закрытия потоков, чтобы это происходило вне зависимости от исключений.

```
try {
    callChecked();
} catch (RuntimeException e) {
    Throwable cause = e.getCause();
    if (cause instanceof Tester.CheckedExceptionV1) {
        System.out.println("Caught CheckedExceptionV1");
    } else if (cause instanceof Tester.CheckedExceptionV2) {
        System.out.println("Caught CheckedExceptionV2");
    }
} finally {
    System.out.println("Finally");
}
```

Отличия checked от unchecked

Если метод выбрасывает checked исключение, то он должен содержать в своем заголовке тип этого исключения (в сигнатуре) после throws. Вызывающие методы должны быть в курсе возможного исключения.

Если метод планирует выкинуть несколько исключений, то их можно перечислить через запятую.

```
public void checked(int n) throws Exception, IOException  1 usage
{
    if (n == 0)
        throw new Exception("n равно нулю!");
    if (n == 1)
        throw new IOException("n равно единице");
}

public void unchecked(int n)  1 usage  new *
{
    if (n == 0)
        throw new RuntimeException("n равно нулю!");
    if (n == 1)
        throw new IllegalArgumentException("n равно единице");
}
```

Отличия checked от unchecked

Checked исключения проверяются на этапе компиляции: компилятор требует, чтобы **checked** исключения либо были обработаны с помощью try-catch, либо проброшены и объявлены в сигнатуре метода с помощью throws.

```
try {
    tester.checked( n: 1);
    //tester.checked(1);
} catch (Tester.CheckedExceptionV1 e) {
    e.printStackTrace();                    //Трейс и сообщение
} catch (Tester.CheckedExceptionV2 e) {
    System.err.println(e.getMessage()); //Только сообщение
}
```

Можно выполнить обработку тремя способами:

1) Не перехватываем возникающие исключения, просто добавить в сигнатуру throws и прокинуть их дальше.
2) Перехватить в блоке catch нужные исключение, а остальные прокинуть через thorws.
3) Перехватить в блоке catch все исключения.

Множественный перехват и собственные исключения

```
try {
    tester.checked( n: 1);
    //tester.checked(1);
} catch (Tester.CheckedExceptionV1 | Tester.CheckedExceptionV2 e) {
    e.printStackTrace();                    //Трейс и сообщение
}
```

```
public static class CheckedException extends Exception {  2 usages  2 inheritors  new *
    public CheckedException(String message) {  2 usages  new *
        super(message);
    }
}
public static class CheckedExceptionV1 extends CheckedException {  2 usages  new *
    public CheckedExceptionV1(String message) {  1 usage  new *
        super("Exception-1 " + message);
    }
}
public static class CheckedExceptionV2 extends CheckedException {  2 usages  new *
    public CheckedExceptionV2(String message) {  1 usage  new *
        super("Exception-2 " + message);
    }
}
```

Обертка checked в unchecked

Если у метода с checked исключением большой стек вызова, программисты часто его оборачивают в Unchecked исключение, а потом достают его.

```
public static void main(String[] args) {
    System.out.println("Hello world!");
    try {
        callChecked();
    } catch (RuntimeException e) {
        Throwable cause = e.getCause();
        if (cause instanceof Tester.CheckedExceptionV1) {
            System.out.println("Caught CheckedExceptionV1");
        } else if (cause instanceof Tester.CheckedExceptionV2) {
            System.out.println("Caught CheckedExceptionV2");
        }
    }
}
public static void callChecked() {  1 usage
    try {
        tester.checked( n: 1);
    } catch (Tester.CheckedExceptionV1 | Tester.CheckedExceptionV2 e) {
        throw new RuntimeException(e);
    }
}
```