Тема 3

Наследование и интерфейсы. Обработка исключительных ситуаций

Макейчик Е.Г.

Наследование

Наследование – механизм передачи данных и кода от одних классов другим, тип отношений между классами.

Для выражения наследования в языке имеется специальное ключевое слово extends.

```
class A {...}
class B extends A {...}
```

Наследование

```java
public class Person {
        protected String name;
        protected int born;
        public Person(String name, int born) {
            this.name = name;
            this.born = born;
        }
        public void info() {
            System.out.println(name + "-" + born);
        }
}
```

```java
public class Employee extends Person {
        protected int payment;
        public Employee(String name, int born, int pay) {
            super(name, born);
            this.payment = pay;
        }
}
```

Ключевое слово super

Вызов конструктора непосредственного суперкласса:

super (parameters) – вызов должен быть первым в конструкторе подкласса;

Доступ к элементу суперкласса, скрытому элементом подкласса:

super.var_name или super.metod_name()

super не влияет на тип доступа.

Наследование

```
public class Program {
    public static void main(String[] args) {
        Employee emp = new Employee( name: "Левчук Р.П.", born: 2002, pay: 3500);
        emp.info();
    }
}
```

Запрет наследования

В некоторых случаях наследование можно запретить

```
final class Person { // Наследовать нельзя!
        protected String name;
        protected int born;
        public Person(String name, int born) {
            this.name = name;
            this.born = born;
        }
        public void info() {
            System.out.println(name + "-" + born);
        }
}
```

Запрет наследования

С помощью final можно запретить переопределение избранных методов.

```
// Неизменяемое поле
final double pi = 3.14;

// Метод, для которого запрещено переопределение (overriding)
final int getX() {...}

// Класс, для которого запрещено наследование
final class A {...}
```

Переопределение методов

В производных классах можно переопределять методы родительских классов

```
class Person {
    public void info() {...}
}
class Employee extends Person {
    public void info() {...}
}
```

Переопределение методов

Для переопределения нужно точно воспроизвести заголовок родительского метода.

Для надежности в производном классе используют аннотацию

```
@Override
public void info() {...}
```

Переопределение метода (overriding) – создание в подклассе метода, совпадающего по сигнатуре с методом суперкласса.

Переопределение не стоит путать с **перегрузкой**, когда в класс добавляется метод с тем же именем, но с другим набором параметров.

Полиморфизм

Ссылочная переменная суперкласса (родителя) может ссылаться на объект подкласса (на том основании, что экземпляры производного класса являются разновидностями базового).

```
class Point {
public int x, int y;
}
class Point3D extends Point {
public int z;
}
Point Pobj = new Point();
Point3D Cobj = new Point3D();
Pobj = Cobj;
Pobj.x = 1;  // Верно, x определен в Point
Pobj.z = 10; // Ошибка, z не определен в Point
```

Полиморфизм

Динамическая диспетчеризация методов – это механизм, позволяющий определить какой из переопределенных методов нужно вызвать, во время выполнения, а не во время компиляции.

```
class A { public int get() {...} }
class B extends A {public int get() {...} }
class C extends B {public int get() {...} }
...
    A one = new A();
    A two = new B();
    A three = new C();
    ...
    one.get()
    two.get()
    three.get()
```

Абстрактные методы и классы

В Java существуют абстрактные классы, которые можно использовать только для наследования, но не для создания экземпляров (объектов).

Производный класс должен переопределить и реализовать все абстрактные методы из базового класса, иначе он сам будет считаться абстрактным.

Модификатор abstract

Можно объявлять абстрактными методы и классы.

Если в классе есть хотя бы один абстрактный метод, то класс должен быть объявлен абстрактным.

Любой подкласс абстрактного класса должен или реализовать все его абстрактные методы или сам должен быть объявлен абстрактным.

Модификатор abstract

Объявление абстрактного метода

```
abstract <type> <name> (<params>);
```

Объявление абстрактного класса

```
abstract class <name> {
...
}
```

Интерфейсы

Интерфейсы – это особый вид классов.

1. Интерфейсы допускают множественное наследование.
2. Все методы – абстрактные (без модификатора abstract).
3. Все переменные – static и final (без соотв. модификаторов), необходима инициализация.
4. Все переменные и методы – public (без модификатора).

Интерфейсы

Особенности реализации интерфейсов:

1. Методы, которые реализуют интерфейс, должны быть объявлены как public. Сигнатура типа реализующего метода должна точно соответствовать сигнатуре типа, указанной в определении интерфейса.

2. Если класс включает интерфейс, но реализует не все его методы, то такой класс должен быть объявлен как абстрактный.

3. Если класс реализует интерфейс, унаследованный от другого интерфейса, класс должен реализовать все методы, определенные в цепочке наследования интерфейсов.

Интерфейсы

```
public interface Callback {
    void callback(int param);
}
```

```
public class Client implements Callback {
        public void callback(int p) {
                System.out.println("Callback called with " + p);
        }
}
```

```
public class Client implements Callback {
        public void callback(int p) // Реализация метода интерфейса
        {
                System.out.println("Callback called with " + p);
        }
        int getSquare(int p) // Собственный метод класса
        {
                System.out.println("Square = " + (p * p));
                return p;
        }
}
```

Интерфейсы

Интерфейсы можно использовать для импорта в различные классы совместно используемых констант.

В том случае, когда вы реализуете в классе какой-либо интерфейс, все имена переменных этого интерфейса будут видимы в классе как константы.

Если интерфейс не включает в себя методы, то любой класс, объявляемый реализацией этого интерфейса, может вообще ничего не реализовывать.

Интерфейсы

```
public interface Constants {
        int NO = 0;
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int SOON = 4;
        int NEVER = 5;
}
```

```
public class Question implements Constants
{
    Random rand = new Random();
    int ask() {
    int prob=(int)(100 * rand.nextDouble());
    if (prob < 30)  return NO;
       else if (prob < 60) return YES;
       else if (prob < 75) return LATER;
       else if (prob < 98) return SOON;
       else return NEVER;
    }
}
```

Класс Object

Все классы, создаваемые разработчиком, имеют наследование от системного класса Object.

Для своих классов можно переопределить:

- toString() – преобразование в строку;
- hashCode() – вычисление хэш-кода объекта;
- equals() – проверка двух объектов на равенство;
- clone() – создание копии объекта;
- finalize() – аналог деструктора.

Класс Object

Нельзя переопределить:

getClass() – получение типа объекта;

Object clone() – создает новый объект, являющийся копией вызывающего;

boolean equals(Object object) – определяет, является ли один объект равным другому;

void finalize() – завершающие действия перед вызовом gc.

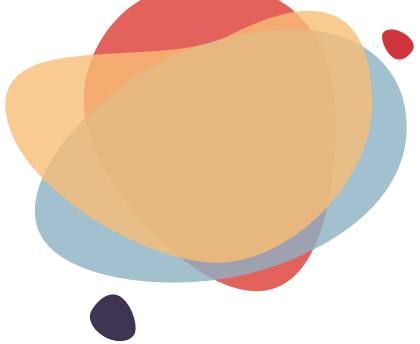
String toString() – возвращает строку, содержащую описание вызывающего объекта. Этот метод вызывается автоматически, когда объект выводится методом print() или println(). Многие классы переопределяют данный метод, приспосабливая описание специально для типов объектов, которые они создают.

Класс Object

При определении equals() обычно требуется переопределить hashCode().

Для hashCode() должны выполняться правила:

- Во время работы значение хэш-кода не меняется, если объект не был изменен;
- Все одинаковые по содержанию объекты одного типа должны иметь одинаковые коды;
- Различные по содержанию объекты одного типа должны иметь разные хэш-коды.



Исключения

Классификация типов ошибок:

1. Build-time errors
    - Синтаксические
    - Семантические
    - Логические
2. Run-time errors (исключения)

Исключения

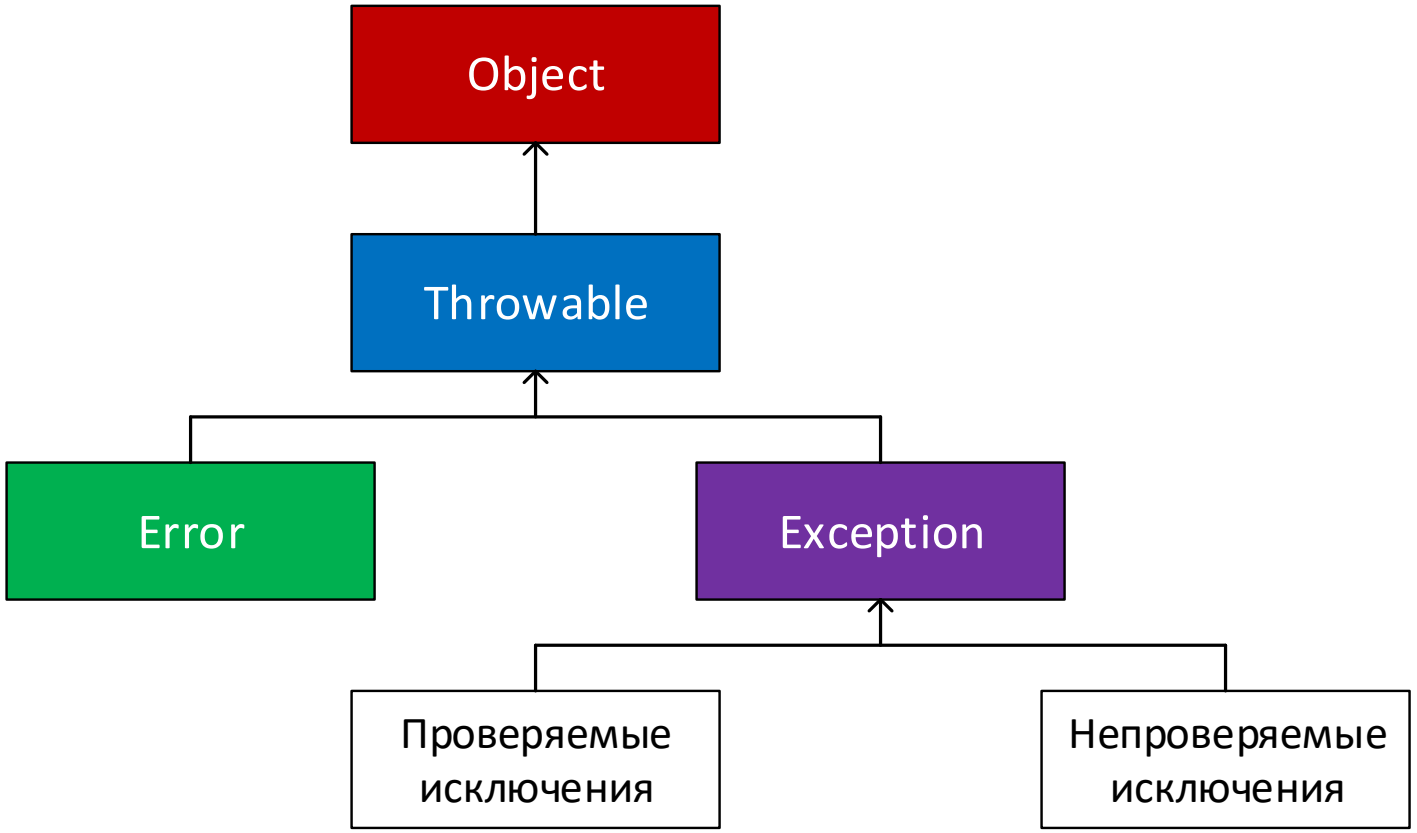
Исключение в Java – это объект некоторого класса, который описывает исключительное состояние, возникшее в каком-либо участке программного кода.

При возникновении исключения исполняющая система Java создает объект класса, связанного с данным исключением. Этот объект хранит информацию о возникшей исключительной ситуации (точка возникновения, описание и т.п.)

Возможна как автоматическая так и программная генерация исключений.

Исключения

В языке Java исключения образуют иерархию классов



Исключения

Среди исключений выделяют:

- Errors – ошибки, связанные с работой виртуальной машины. Программой не обрабатываются (исчерпание памяти и т.д.)
- Runtime Exceptions – необрабатываемые исключения, унаследованные от класса RuntimeException. Могут обрабатываться в программе, а могут и нет (например, NullPointException).
- Exceptions, не унаследованные от RuntimeException и считающиеся обрабатываемыми. Требуют обязательной обработки.

Исключения

Формат обработчика

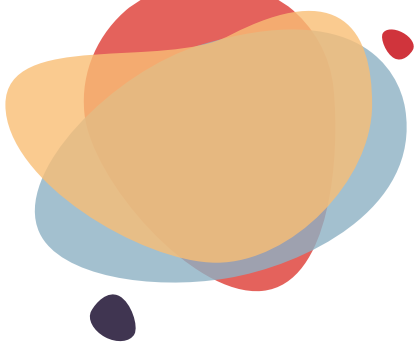
```
try {
    // Блок кода
}
catch (ExceptionType1 exOb1) {
    // Обработчик исключений типа ExceptionType1
}
catch (ExceptionType2 exOb2) {
    // Обработчик исключений типа ExceptionType2
}
finally
{
    // Код, который выполняется перед выходом из блока try
}
```

```
try {
    read_from_file("data.txt");
    calculate();
}
catch (FileNotFoundException fe) {
    System.out.println("Файл data.txt не найден");
}
catch (ArithmeticException aex) {
    System.out.println("Деление на ноль");
}
```

Исключения

try-catch блок:

- управление никогда не возвращается из блока catch обратно в блок try, после выполнения catch-блока управление передается строке, следующей сразу после try-catch-блока;
- область видимости catch-блока ограничена ближайшим предшествующим утверждением try, т.е. catch-блок не может захватывать исключение, выброшенное «не своим» try-блоком;
- операторы, контролируемые утверждением try, должны быть окружены фигурными скобками даже если это одиночная инструкция;
- блоки try могут быть вложенными/



Исключения

Catch-блоки просматриваются в порядке их появления в программе, при этом обработчик catch для суперкласса перехватывает исключения как для своего класса так и для всех его подклассов.

Следовательно, в последовательности catch-блоков подклассы исключений должны следовать перед любым из суперклассов.

Исключения

Программная генерация исключения:

```
throw new <ExceptionClassName>();
throw new <ExceptionClassName>("...");
```

```
public static void demoproc() {
    try {
        throw new NullPointerException("Demo");
    }
    catch (NullPointerException e) {
        System.out.println("Caught inside demoproc"); throw e;
    }
}
public static void main(String args[]) {
    try {
        demoproc();
    }
    catch (NulPointerException e) {
        System.out.println("Recaught: " + e);
    }
}
```

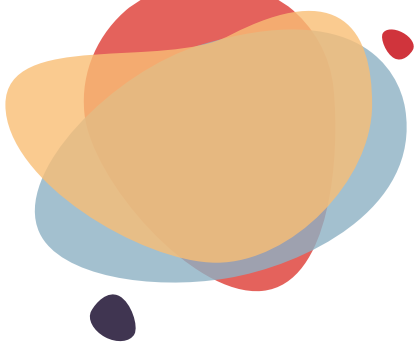
Исключения

Исключения, которые порождены от Exception, но не от RuntimeException, могут быть сгенерированы только явно операцией throw.

При этом если метод может выбрасывать одно из таких исключений, то должно выполняться одно из двух условий:

- либо для такого исключения должен быть catch-обработчик;
- либо в заголовке такого метода должна стоять конструкция:

```
throws <ExceptionClassName>
...
public String readLine() throws IOException
```



Исключения

Вызов метода, в описании которого стоит “throws ...“, тоже должен находиться либо внутри try-catch-блока, либо внутри метода с конструкцией “throws ...“ в его заголовке и т.д. вплоть до метода main().

Таким образом, где-то в программе любое возможное исключение обязано быть перехвачено и обработано.

Подклассы Exception

```
public class MyException extends Exception
{
    private int detail;
    MyException(int a)
    {
        detail = a;
    }
    public String toString()
    {
        return "MyException[" + detail + "]";
    }
}
```

Если надо создать исключение необязательное к перехвату – его надо унаследовать от RuntimeException.