**1. Алгоритмы в библиотеке stl.**

Алгоритмы STL реализованы в виде глобальных функций, которые работают с использованием итераторов. Это означает, что каждый алгоритм нужно реализовать всего лишь один раз, и он будет работать со всеми контейнерами, которые предоставляют набор итераторов (включая и ваши собственные (пользовательские) контейнерные классы). Хотя это имеет огромный потенциал и предоставляет возможность быстро писать сложный код, у алгоритмов также есть и «тёмная сторона» — некоторая комбинация алгоритмов и типов контейнеров может не работать/работать с плохой производительностью/вызывать бесконечные циклы, поэтому следует быть осторожным.

Библиотека STL предоставляет довольно много алгоритмов. На этом уроке мы затронем лишь некоторые из наиболее распространенных и простых в использовании алгоритмов. Для их работы нужно подключить заголовочный файл **algorithm**.

for\_each() - выполнение действия для группы элементов контейнера

Поиск:

find() – поиск элемента в коллекции

find\_if() – поиск элемента в коллекции по условию

find\_if\_not()

find\_first\_of() – нахождение первого совпадения в коллекции

find\_end() – нахождение последнего совпадения в коллекции

adjacent\_find() - поиск первого вхождения одинаковых элементов, которые являются соседними.

count() – возвращает число совпадений value с элементами коллекции

count\_if() – возвращает число элементов коллекции, подходящих под условие

search() – поиск элемента коллекции

binary\_search() – бинарный поиск

min() – поиск минимального из значений

max() – поиск максимального из значений

minmax\_element() – возвращает пару иттераторов, первый из которых указывает на минимальный элемент в диапазоне коллекции, второй-на максимальный

min\_element() – возвращает иттератор, который указывает на минимальный элемент в диапазоне коллекции

max\_element() - возвращает иттератор, который указывает на максимальный элемент в диапазоне коллекции

equal() – возвращает true, если обе последовательности одинаковы в диапазоне, ограниченном парой итераторов [first,last).

Тасовка коллекции:

fill() – заполнение контейнера значением

reverse() – реверс элементов контейнера в диапазоне [first, last)

rotate() -  вращает порядок элементов в диапазоне [first, last) таким образом, что элемент, на который указывает середина, становится новым первым элементом, то есть слева.

rotate\_copy() - копирует элементы в диапазоне [first, last) в диапазон, начинающийся с результата, но поворачивает порядок элементов таким образом, что элемент, на который указывает середина, становится первым элементом в результирующем диапазоне, т. е. оставленным поворот.

shuffle(), random\_shuffle() - переставляет элементы из диапазона [first,last) в случайном порядке. Во втором варианте можно передать объект-функцию или указатель на функцию, генерирующую случайные числа. Ожидается, что генератор rand возвращает значение типа double в интервале [0,1].

Единственное отличие состоит в том, что random\_shuffle использует функцию rand () для рандомизации элементов, тогда как shuffle использует urng, который является лучшим генератором случайных чисел, хотя с определенной перегрузкой random\_shuffle мы можем получить то же поведение (как и с shuffle).

transform() – генерирует новую последовательность по старой по какому-либо правилу

replace() - заменяет в диапазоне [first,last) все элементы со значением old\_value на new\_value.

replace\_if() - заменяет в диапазоне [first,last) все элементы удовлетворяющие условию на значение

Взаимодействие с 2 коллекциями:

copy() – копирует последовательность из диапазона в новый диапазон

copy\_if() – то же самое только те, которые удовлетворяют условию

merge() – слияние двух отсортированных коллекций

Сортировка:

sort() – сортирует коллекцию

stable\_sort() - «устойчивая» сортировка слиянием, устойчивость которой означает, что она сохраняет относительный порядок равных элементов после сортировки.

is\_sorted() – проверка на отсортированность в порядке возрастания

is\_sorted\_until() возвращает иттератор на первый элемент в коллекции, который не стоит в порядке возрастания

**2. Конструктор. Конструктор по умолчанию, конструктор копирования. Оператор присваивания.**

Конструктор – метод класса с именем класса. Вызывается при создании переменной. Не возвращается значения. У класса могут быть несколько конструкторов, которые различаются параметрами. Служит для инициализации. Конструктор по умолчанию автоматически неявно создаётся для каждого класса и используется в случае отсутствия конструктора, описанного программистом.

Конструктор копирования — это специальная функция-член, принимающая в качестве входных данных ссылку на объект того же типа и создающая его копию. Начиная с версии C++ 11 в языке поддерживаются два вида присваивания: присваивание копированием и присваивание перемещением.

Компилятор C++ по умолчанию автоматически генерирует конструктор копирования или оператор присваивания, если они не объявлены в классе явно. Автоматически сгенерированный конструктор выполняет почленное копирование, а автоматически сгенерированный оператор присваивания – почленное присваивание. Операция присваивания копированием отличается от конструктора копирования тем, что должна очищать члены-данные цели присваивания (и правильно обрабатывать самоприсваивание), тогда как конструктор копирования присваивает значения неинициализированным членам-данным.

**3 Контейнеры в библиотеке stl.**

Стандартная библиотека предоставляет различные типобезопасные контейнеры для хранения коллекций связанных объектов. Контейнеры — это шаблоны классов. При объявлении переменной контейнера указывается тип элементов, которые будет содержать контейнер. Контейнеры могут создаваться с использованием списков инициализаторов. Они содержат функции-члены для добавления и удаления элементов и выполнения других операций.

Итерация элементов в контейнере и доступ к отдельным элементам осуществляются с помощью итераторов. Вы можете использовать итераторы явно, с помощью их функций-членов и операторов, а также глобальных функций. Вы можете также использовать их неявно, например с помощью цикла range-for. Итераторы для всех контейнеров STL имеют общий интерфейс, но каждый контейнер определяет собственные специализированные итераторы.

Контейнеры можно разделить на три категории: последовательные контейнеры, ассоциативные контейнеры и контейнеры-адаптеры.

## [Последовательные контейнеры](javascript:void(0))

Последовательные контейнеры поддерживают указанный пользователем порядок вставляемых элементов.

Контейнер **vector** ведет себя как массив, но может автоматически увеличиваться по мере необходимости. Он поддерживает прямой доступ и связанное хранение и имеет очень гибкую длину. По этим и многим другим причинам контейнер **vector** является наиболее предпочтительным последовательным контейнером для большинства областей применения.

Контейнер **array** обладает некоторыми преимуществами контейнера **vector**, однако его длина не обладает такой гибкостью.

Контейнер **deque** (двусторонняя очередь) обеспечивает быструю вставку и удаление в начале и в конце контейнера. Он, как и контейнер **vector**, обладает преимуществами прямого доступа и гибкой длины, но не обеспечивает связанное хранение.

Контейнер **list** — это двунаправленный список, который обеспечивает двунаправленный доступ, быструю вставку и удаления в любом месте контейнера, но не поддерживает прямой доступ к элементам контейнера.

Контейнер **forward\_list** — однонаправленный список. Это версия контейнера **list** только с доступом в прямом направлении.

## [Ассоциативные контейнеры](javascript:void(0))

В ассоциативных контейнерах элементы вставляются в предварительно определенном порядке — например, с сортировкой по возрастанию. Также доступны неупорядоченные ассоциативные контейнеры. Ассоциативные контейнеры можно объединить в два подмножества: сопоставления (set) и наборы (map).

Контейнер **map**, который иногда называют словарем, состоит из пар "ключ-значение". Ключ используется для упорядочивания последовательности, а значение связано с ключом. Например, **map** может содержать ключи, представляющие каждое уникальное ключевое слово в тексте, и соответствующие значения, которые обозначают количество повторений каждого слова в тексте. **unordered\_map** — это неупорядоченная версия **map**.

**set** — это контейнер уникальных элементов, упорядоченных по возрастанию. Каждое его значение также является и ключом. **set** — это неупорядоченная версия **unordered\_set**.

Контейнеры **map** и **set** разрешают вставку только одного экземпляра ключа или элемента. Если необходимо включить несколько экземпляров элемента, следует использовать контейнер **multimap** или **multiset**. Неупорядоченные версии этих контейнеров —**unordered\_multimap** и **unordered\_multiset**.

Упорядоченные контейнеры map и set поддерживают двунаправленные итераторы, а их неупорядоченный аналоги — итераторы с перебором в прямом направлении.

## [Контейнеры-адаптеры](javascript:void(0))

Контейнер-адаптер — это разновидность последовательного или ассоциативного контейнера, который ограничивает интерфейс для простоты и ясности. Контейнеры-адаптеры не поддерживают итераторы.

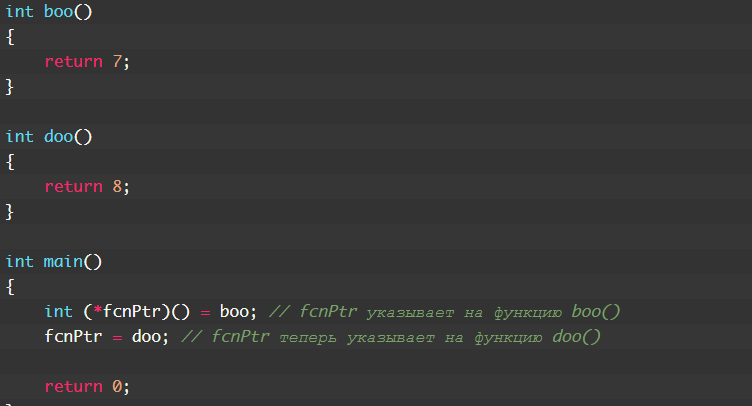
Контейнер **queue** соответствует семантике FIFO (первым поступил — первым обслужен).Первый элемент, который отправляется, то есть вставляется, в очередь, должен быть первым элементом, извлекаемым из очереди.

Контейнер **priority\_queue** упорядочен таким образом, что первым в очереди всегда оказывается элемент с наибольшим значением.

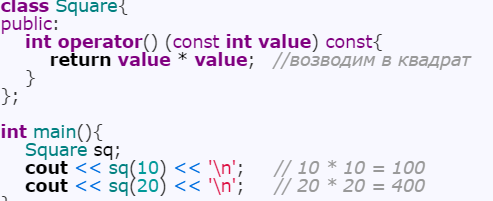
Контейнер **stack** соответствует семантике LIFO (последним поступил — первым обслужен).Последний элемент, отправленный в стек, становится первым извлекаемым элементом.

Поскольку контейнеры-адаптеры не поддерживают итераторы, их нельзя использовать в алгоритмах STL.

**4 Указатели на функции. Функциональные объекты и Лямбда выражения.**

Указатель на функцию может быть инициализирован функцией.

Функторы (или функциональные объекты) — это любой объект, использование которого возможно подобно вызову функции. Пишется за счет “operator()”, тем самым мы можем в main вызывать объект класса, т.е sq.

****

**Лямбда-выражение** (или просто **«лямбда»**) **в программировании позволяет определить анонимную функцию внутри другой функции**. Возможность сделать функцию вложенной является очень важным преимуществом, так как позволяет избегать как захламления [**пространства имен**](https://ravesli.com/urok-53-prostranstva-imen/) лишними объектами, так и определить функцию как можно ближе к месту её первого использования.

[ captureClause ] ( параметры ) -> возвращаемыйТип  
{  
стейтменты;  
}

**5. Механизмы синхронизации и параллельное выполнение в стандартной библиотеке С++11/14**

Методы синхронизации выполнения потоков:

•Ожидание завершения потока при помощи pthread\_join

•Использование mutex

•Использование условных переменных

**Использование pthread\_join**

int pthread\_join(pthread\_t tid,void \*\*status)

Блокирует выполнение текущего потока до того момента, как поток указанный при помощи tid завершится

Пример использования pthread\_join

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void \* print\_message\_function( void \*ptr ) {

char \*message; message = (char \*) ptr; printf("%s \n", message);

}

int main() {

pthread\_t thread1, thread2;

char \*message1 = "Thread 1";

char \*message2 = "Thread 2";

int iret1, iret2;

iret1 = pthread\_create( &thread1, NULL, print\_message\_function, (void\*) message1);

iret2 = pthread\_create( &thread2, NULL, print\_message\_function, (void\*) message2);

pthread\_join( thread1, NULL);

pthread\_join( thread2, NULL);

printf("Thread 1 returns: %d\n",iret1); printf("Thread 2 returns: %d\n",iret2);

exit(0);

}

**Mutex**

Мутекс —переменная, которая может быть захвачена только одним процессом в данный момент времени. Если несколько потоков пытаются захватить mutex, то один захватывает, остальные ждут.

Мутексы используются для предотвращения некорректного изменения данных из за гонок. Гонки возникают, когда несколько потоков выполняют действия над общими переменными, а результат выполенения зависит от порядка выполенения действий.

Типичная последовательность действий с Mutex:

•Создать и инициализировать переменную mutex

•Несколько потоков пытаются захватить mutex

•Только один захыватывает mutex

•Захвативший поток выполняет необходимые операции

•Захвативший поток освобождает mutex

•Другие потоки пытаются захватить mutex ...

•По окончанию работы mutex освобождается

**Вызовы создания mutex**

pthread\_mutex\_init (mutex,attr)

pthread\_mutex\_destroy (mutex)

pthread\_mutexattr\_init (attr)

pthread\_mutexattr\_destroy (attr)

Переменная mutex имеет тип pthread\_mutex\_t.

Может инициализирована так:

pthread\_mutex\_t mymutex = PTHREAD\_MUTEX\_INITIALIZER;

или через

pthread\_mutex\_init

**Вызовы для захвата/освобождений mutex**

•pthread\_mutex\_lock (mutex)

•pthread\_mutex\_trylock (mutex)

•pthread\_mutex\_unlock (mutex)

**Мутексы пример использования**

#include <pthread.h>

pthread\_mutex\_t m = PTHREAD\_MUTEX\_INITIALIZER;

static int k;

void \* producer( void \* arg ){

while(1) {

pthread\_mutex\_lock( &m );

++k;

pthread\_mutex\_unlock( &m );

}

}

void \* consumer( void \* arg ){

while(1) {

pthread\_mutex\_lock( &m );

printf( “k = %d\n”, k );

pthread\_mutex\_unlock( &m );

}

}

int main() {

int rc1, rc2; pthread\_t thread1, thread2;

pthread\_create( &thread1, NULL, &functionC, NULL));

pthread\_create( &thread2, NULL, &functionC, NULL));

pthread\_join( thread1, NULL);

pthread\_join( thread2, NULL);

}

**Условные переменные (Condition Variables)**

Механизм условных переменных позволяет потокам преостанавливать выполнение до того момента, пока условие не будет истинным. Условная перменная всегда должна быть связана с mutex для исключения гонок.

**Создание/уничтожение:**

pthread\_cond\_init

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_destroy

Ожидание условия:

pthread\_cond\_wait

pthread\_cond\_timedwait -place limit on how long it will block.

Пробуждение потока при возникновении условия:

pthread\_cond\_signal

pthread\_cond\_broadcast –пробуждение всех ожидающих потоков ожидающих на данной переменной.

**Пример использовани условных переменных**

pthread\_mutex\_t count\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t condition\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t condition\_cond = PTHREAD\_COND\_INITIALIZER; int count = 0;

#define COUNT\_DONE 10

#define COUNT\_HALT1 3

#define COUNT\_HALT2 6

void \*functionCount1() {

for(;;) {

pthread\_mutex\_lock( &condition\_mutex );

while( count >= COUNT\_HALT1 && count <= COUNT\_HALT2 )

pthread\_cond\_wait( &condition\_cond, &condition\_mutex );

pthread\_mutex\_unlock( &condition\_mutex );

pthread\_mutex\_lock( &count\_mutex );

count++; printf("Counter value functionCount1: %d\n",count);

pthread\_mutex\_unlock( &count\_mutex );

if(count >= COUNT\_DONE) return(NULL);

}

}

void \*functionCount2() {

for(;;) {

pthread\_mutex\_lock( &condition\_mutex );

if( count < COUNT\_HALT1 || count > COUNT\_HALT2 ) pthread\_cond\_signal( &condition\_cond );

pthread\_mutex\_unlock( &condition\_mutex );

pthread\_mutex\_lock( &count\_mutex );

count++; printf("Counter value functionCount2: %d\n",count) ;

pthread\_mutex\_unlock( &count\_mutex );

if(count >= COUNT\_DONE) return(NULL);

}

}

main() {

pthread\_t thread1, thread2;

pthread\_create( &thread1, NULL, &functionCount1, NULL); pthread\_create( &thread2, NULL, &functionCount2, NULL);

pthread\_join( thread1, NULL); pthread\_join( thread2, NULL);

exit(0);

}

**6 Обработка исключительных ситуаций - исключения.**

**Обработка исключения**

try {

// protected code

} catch( ExceptionName e1 ) {

// catch block

} catch( ExceptionName e2 ) {

// catch block

} catch( ExceptionName eN ) {

// catch block

}

**Выброс исключения throw <тип>**

**Пример**

#include <iostream>

using namespace std;

double division(int a, int b) {

if( b == 0 )

throw "Division by zero condition!";

return (a/b);

}

int main () {

int x = 50;

int y = 0;

double z = 0;

try {

z = division(x, y);

cout << z << endl;

} catch(const char\* msg) {

cerr << msg << endl;

}

return 0;

}

**Ещё пример**

#include <iostream>

using namespace std;

double division(inta, intb) {

if( b == 0 )

throw "Division by zero condition!";

return (a/b);

}

intmain () {

intx = 50;

inty = 0;

double z = 0;

try {

z = division(x, y);

cout<< z << endl;

} catch(constchar\* msg) {

cerr<< msg<< endl;

}

catch (…) {

// Поймает все

cout<<“Error!!!”<<end;

}

}

**7 Обработка ошибок в стиле С и в стиле С++. Коды возврата и исключения.**

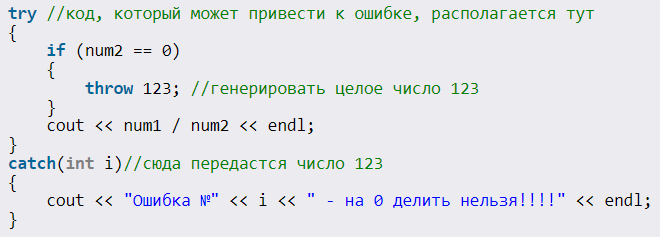
**В языке С errno** – переменная, хранящая целочисленный код последней ошибки. В каждом потоке существует своя локальная версия **errno**, чем и обусловливается её безопасность в многопоточной среде. Обычно **errno** реализуется в виде макроса, разворачивающегося в вызов функции, возвращающей указатель на целочисленный буфер. При запуске программы значение **errno** равно нулю.  
  
Все коды ошибок имеют положительные значения, и могут использоваться в директивах препроцессора **#if**. В целях удобства и переносимости заголовочный файл <errno.h> определяет макросы, соответствующие кодам ошибок.  
  
Стандарт ISO C определяет следующие коды:

* **EDOM** – (**E**rror **dom**ain) ошибка области определения.
* **EILSEQ** – (**E**rror **i**nva**l**id **seq**uence) ошибочная последовательность байтов.
* **ERANGE** – (**E**rror **range**) результат слишком велик.

Прочие коды ошибок (несколько десятков) и их описания определены в стандарте POSIX. Кроме того, в спецификациях стандартных функций обычно указываются используемые ими коды ошибок и их описания.

**Обработка исключений в С++** использует три ключевых слова: try, catch и throw. Те инструкции программы, где ожидается возможность появления исключительных ситуаций, содержатся в блоке try. Если в блоке try возникает исключение, т. е. ошибка, то генерируется исключение. Исключение перехватывается, используя catch, и обрабатывается. Ниже это общее описание будет рассмотрено более подробно.

Инструкция, генерирующая исключение, должна исполняться внутри блока try. Вызванные из блока try функции также могут генерировать исключения. Всякое исключение должно быть перехвачено инструкцией catch, которая непосредственно следует за инструкцией try, сгенерировавшей исключение.



**8 Объектно-ориентированное программирование. Наследование, множественное наследование и полиморфизм в C++.**

Объе́ктно-ориенти́рованное программи́рование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. Необходимо обратить внимание на следующие важные части этого определения: 1) объектно-ориентированное программирование использует в качестве основных логических конструктивных элементов объекты, а не алгоритмы; 2) каждый объект является экземпляром определенного класса; 3) классы образуют иерархии. Программа считается объектно-ориентированной, только если выполнены все три указанных требования. В частности, программирование, не использующее наследование, называется не объектно-ориентированным, а программированием с помощью абстрактных типов данных.

Полиморфизм (polymorphism) (от греческого polymorphos) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Например для языка Си, в котором полиморфизм поддерживается недостаточно, нахождение абсолютной величины числа требует трёх различных функций: abs(), labs() и fabs(). Эти функции подсчитывают и возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. В С++ каждая из этих функций может быть названа abs(). Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. В С++ можно использовать одно имя функции для множества различных действий. Это называется перегрузкой функций (function overloading).

Наследование (inheritance) - это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов (hierarchical classification). Применение иерархии классов делает управляемыми большие потоки информации.

Множественное наследование представляет собой возможность порождать класс из нескольких базовых классов. При использовании множественного наследования производный класс получает характеристики (элементы) существующих базовых классов. Поддержка множественного наследования в C++ предоставляет вашим программам огромные возможности объектно-ориентированного программирования.

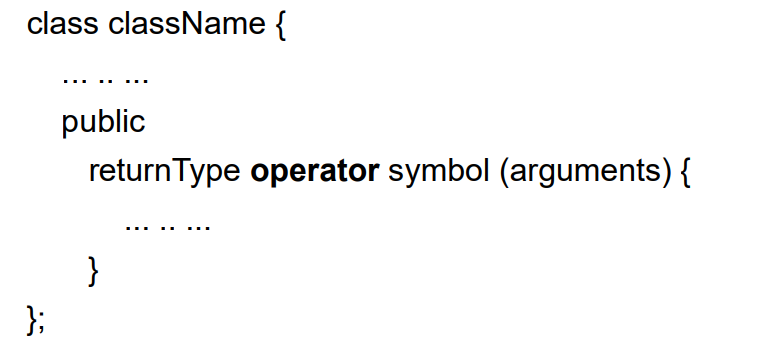
Множественное наследование является способностью порожденного класса наследовать характеристики нескольких базовых классов.

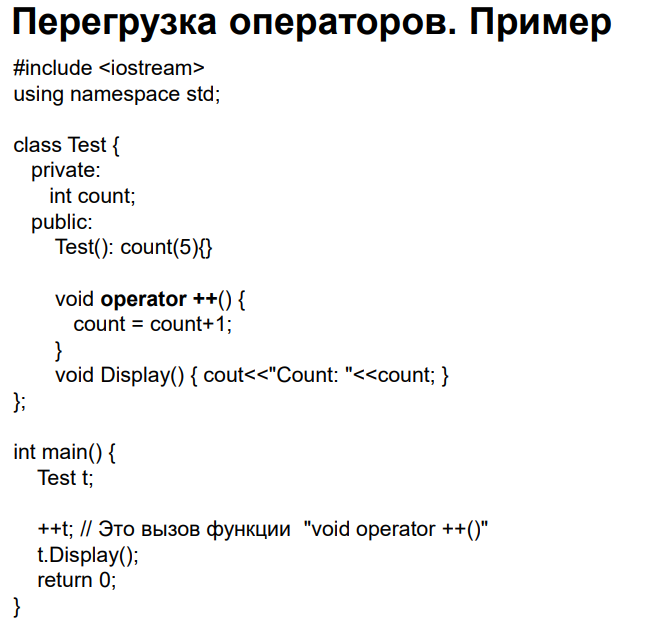
Для порождения класса из нескольких базовых после имени нового класса и двоеточия вы указываете имена базовых классов, разделяя их запятыми, например class cabbit: public cat, public rabbit.

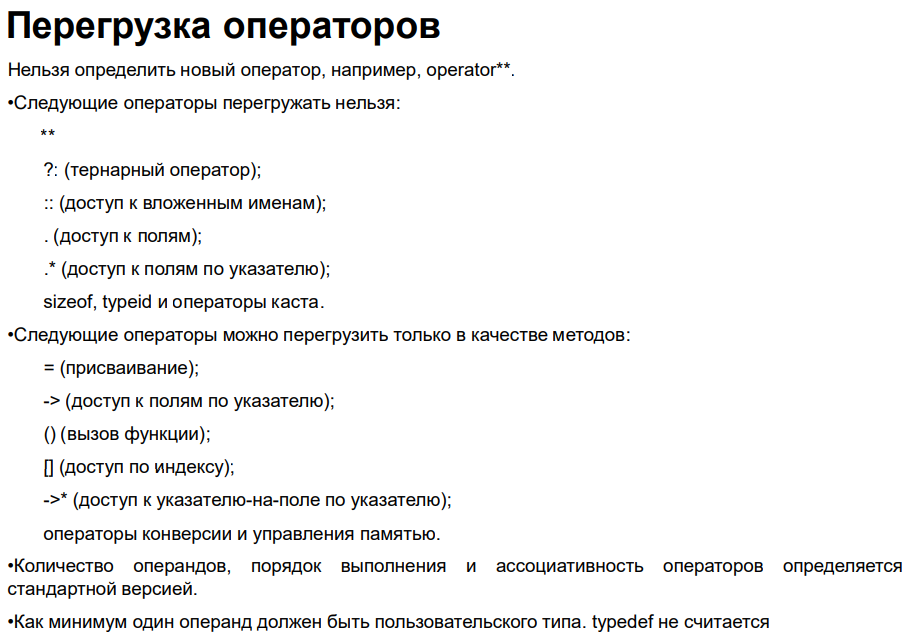
При определении конструктора производного класса вы должны вызвать конструкторы всех базовых классов, передавая им необходимые параметры.

**9 Перегрузка операторов в C++. Операторы вывода в поток.**

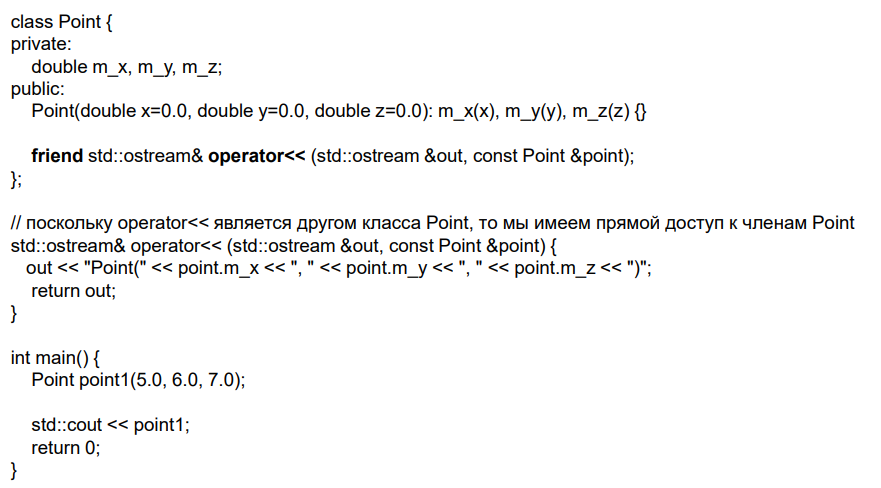
Перегрузка операторов при помощи ключевого слова operator, это всего лишь более удобный способ вызова функций. Использовать её следует только тогда, когда это упростит написание кода. Возможность перегрузки есть только для пользовательских типов/классов.





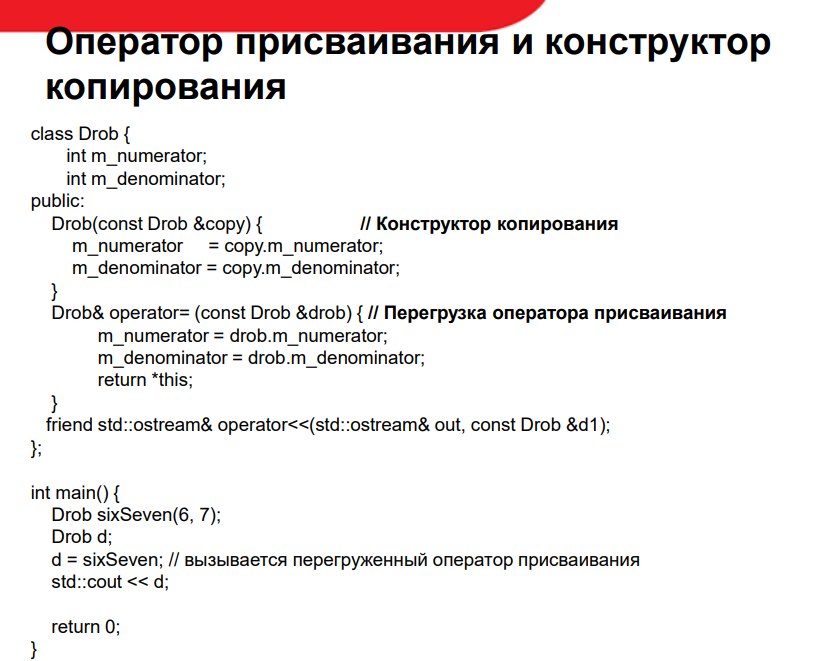


**Перегрузка оператора <<**



**10 Перегрузка операторов в C++. Оператор присваивания. Арифметические операторы.**

**ПРО ПЕРЕГРУЗКУ ОПЕРАТОРОВ СМ ПРЕДЫДУЩИЙ ВОПРОС.**



**Перегрузка арифметических операторов аналогично.**

**11. Потоки ввода вывода в stl.**

### Методы потока ввода

* ***gcount***() const - возвращает число символов, полученных предыдущей функцией ввода;
* ***get***() - возвращает символ из потока, если такой есть;
* ***get***(char\_type& c) - считать символ в переменную c;
* ***get***(char\_type\* s, streamsize n,char\_type delim) - считать n-1 символов в массив s, ввод пректатиться если следующий символ eof (конец файла) или delim;
* ***get***(char\_type\* s, streamsize n) - аналогично вызову get(s,n,widen("\n"));
* ***get***(streambuf\_type& sb,char\_type delim) - считать символы (до eof или delim) в указанный буфер;
* ***get***(streambuf\_type& sb) - аналогично вызову get(sb,widen("\n"));
* ***getline***(char\_type\* s, streamsize n, char\_type delim) - считать строку, ограниченную символом eof или delim в указанный буфер s;
* ***getline***(char\_type\* s, streamsize n) - аналогично вызову getline(s,n,widen("\n"));
* ***ignore***(streamsize n=1, int\_type delim=traits::eof()) - пропускает данные в потоке ввода до символа delim;
* ***peek***() - возвращает traits::eof() если конец, иначе следующий символ в потоке, при этом указатель позиции не меняется;
* ***putback***(char\_type c) - вставляет символ в поток ввода;
* ***read***(char\_type\* s, streamsize n) - cчитывает n (а не n-1) символов в буфер s;
* ***seekg***(off\_type& off, ios\_base::seek\_dir dir) - сместить указатель ввода;
* ***tellg***() - возвращает позицию указателя ввода.

### методы потока вывода

* ***flush***() - сбрасывает буфер вывода;
* ***put***(char\_type c) - вставка символа;
* ***seekp***(off\_type off, ios\_base::seekdir dir) - смещает позицию вывода;
* ***tellp***() - возвращает позицию указателя вывода;
* ***write***(const char\_type\* s, streamsize n) - вставляет n символов из массива s;

Там где не указано возвращаемое значение, предполагается ссылка на поток (т.е. на себя). Как интерпретировать смещение позиции ввода/вывода определяется вложенным типом seek\_dir.

typedef int seekdir;

enum seek\_dir {

beg = 0x0, // смещение от начала потока

cur = 0x1, // смещение от текущей позиции

end = 0x2 // смещение от конца

};

### состояние потока

* ***eof***() - возвращает true, если достигнут конец ввода;
* iostate ***rdstate***() const - возвращает состояние потока.

Значение состояния определяется вложенным типом io\_state.

typedef int iostate; // состояние потока

enum io\_state {

goodbit = 0x00, // все нормально

badbit = 0x01, // потеряна целостность последовательности ввода иди вывлжа

eofbit = 0x02, // достигнут конец в последовательности входных данных

failbit = 0x04 // невозможно взять или вывести символ

};

### манипуляторы

Кроме методов для работы с потоками определены глобальные функции *манипуляторы*:

* ***ws***(basic\_istream& is) - пропускает пустые символы (пробелы, табуляторы) в потоке ввода;
* ***endl***(basic\_ostream& os) - вставляет символ новой строки и сбрасывает буфер в потоке вывода;
* ***ends***(basic\_ostream& os) - вставляет нулевой символ в поток вывода;
* ***flush***(basic\_ostream& os) - сбрасывает буфер потока вывода.

**12. Потоки и параллельное выполнение в стандартной библиотеке С++11/14**

**ЛЕКЦИЯ №6, ПЕРВАЯ ПОЛОВИНА (ОТ THREAD ДО СИНХРОНИЗАЦИИ)**

**13 Преобразование типов в C++. Явное и неявное преобразование типов.**

Преобразование значения переменной одного типа в значение другого типа называется **приведение типа** и бывает **явным** и **неявным**:

В языке C++ есть 5 видов **операций явного преобразования типов**:

**конвертация C-style**;

**применение оператора static\_cast**;

**применение оператора const\_cast**;

**применение оператора dynamic\_cast**;

**применение оператора reinterpret\_cast**.

#### Пример явного приведения типа C-style

**int x = 5;  
double y = 15.3;  
x = (int) y;**  
**y = (double) x;**

#### Пример неявного приведения типа

**int x = 5;  
double y = 15.3;**  
**y = x;** //здесь происходит неявное приведение типа к double  
**x = y;** //здесь происходит неявное приведение типа к int

#### Неявное приведение типа при арифметических операциях

|  |  |
| --- | --- |
| **типы операндов** | **тип результата** |
| float / float | float |
| float / int | float |
| int / float | float |
| int / int | int |

Заметим, что в последнем случае (и только в нем) осуществляется **целочисленное деление с отбрасыванием остатка**.

### Зачем нужно явное приведение типов если есть неявное?

#### Чему в этом случае будет равно z?

**int x = 12;**  
**int y = 7;**  
**double z = x/y;**

В данном выражении разбор начнется с операции наиболее высокого приоритета — с деления, и только потом дело дойдет до присваивания.

Т.е. операция с точки зрения типов будет такой: **z = (double)( (int)x / (int)y )**

Итого, **z = 1.0**

#### А чему будет равно z в этом случае?

**int x = 12;**  
**int y = 7;**  
**double z = (double)x/y;**

В данном выражении разбор начнется с операции наиболее высокого приоритета — с унарной операции приведения типов, и только потом дело дойдет до деления.

Т.е. операция с точки зрения типов будет такой: **z = (double)x / (int)y**

Итого, **z = 1.714285714285714**

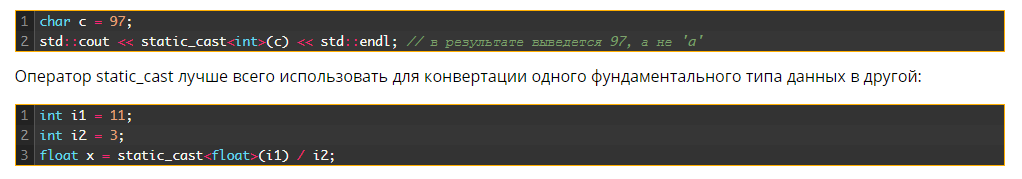
Таким образом, провести вещественное деление над целыми числами можно только при помощи явного преобразования типов.

Конвертация C-style не проверяется компилятором во время компиляции, поэтому она может быть неправильно использована, например, при конвертации типов const или изменении типов данных, без учета их диапазонов (что приведет к переполнению).

Следовательно, конвертацию C-style лучше не использовать.

**Правило: Не используйте конвертацию C-style.**

В языке C++ есть еще один оператор явного преобразования типов данных — **оператор static\_cast**.



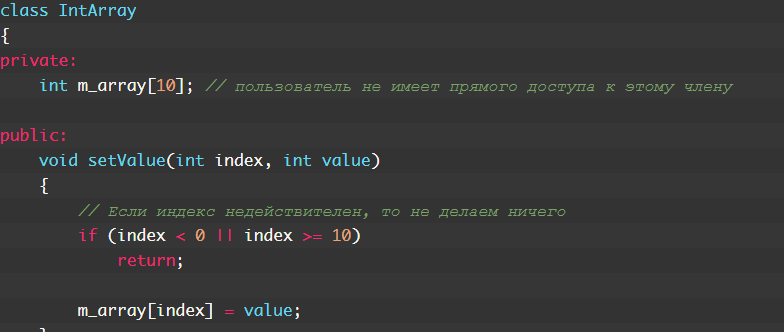
Основным преимуществом оператора static\_cast является проверка его выполнения компилятором во время компиляции, что усложняет возможность возникновения непреднамеренных проблем.

**14 Проектирование программ – инкапсуляция**

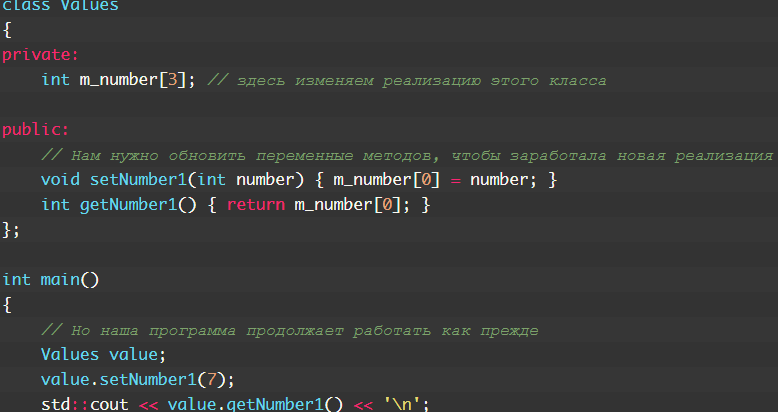
В ООП **инкапсуляция** (или ***«сокрытие информации»***) — это процесс скрытого хранения деталей реализации объекта. Пользователи обращаются к объекту через открытый интерфейс. В языке C++ инкапсуляция реализована через [**спецификаторы доступа**](https://ravesli.com/urok-114-spetsifikatory-dostupa-public-i-private/)(private, public, protected(для дочерних классов)) Как правило, все переменные-члены класса являются закрытыми (скрывая детали реализации), а большинство методов являются открытыми (с открытым интерфейсом для пользователя).

**Преимущества инкапсулированных классов**:

* **Инкапсулированные классы проще в использовании и уменьшают сложность ваших программ. Для использования** класса, вам не нужно знать детали его реализации. Это значительно снижает сложность ваших программ, а также уменьшает количество возможных ошибок. Это является ключевым преимуществом инкапсуляции.
* **Инкапсулированные классы помогают защитить ваши данные и предотвращают их неправильное использование. Это позволяет защитить программу от чужого воздействия, чтобы сохранялся принцип “Работает – не надо трогать”**

****

* **Инкапсулированные классы легче изменить. Если нам понадобиться, к примеру, изменить тип переменной в public, то существует вероятность сломать программу. Инкапсуляция позволяет** изменить способ реализации классов, не нарушая при этом работу всех программ, которые их используют.

****

**15 Пространства имен. Области видимости в C++.**  
**Пространство имен** определяет область кода, в которой гарантируется уникальность всех идентификаторов. По умолчанию, [**глобальные переменные**](https://ravesli.com/urok-49-globalnye-peremennye/) и обычные функции определены в **глобальном пространстве имен. Если 2 идентификатора имеют одинаковые имена и оба определены в глобальном пространстве, то это вызовет ошибку. Чтобы этого избежать, в языке есть возможность объявлять собственные пространства имен через ключевое слово namespace.**

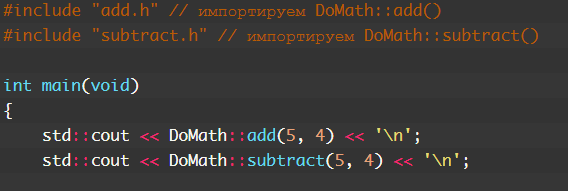
**namespace** foo {

int bar;

}

Но вне блока требуется указание имени пространства имен, в случае с примером выше должны написать **foo::bar**.

Допускается объявление пространств имен в нескольких местах (либо в нескольких файлах, либо в нескольких местах внутри одного файла). Всё, что находится внутри одного блока имен, считается частью только этого блока.



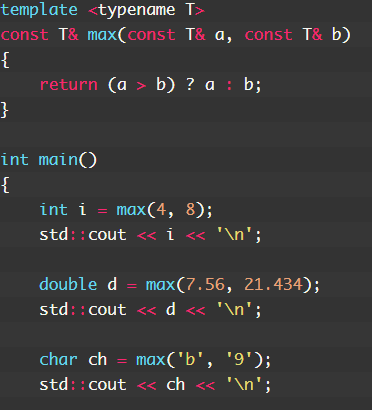
Также одни пространства имен могут быть вложены в другие пространства имен, тогда обращение будет **Boo::Doo**, где **Boo** родитель.

**Область видимости** определяет, где можно использовать переменную. Переменные, определенные внутри вложенных блоков, уничтожаются, как только заканчивается вложенный блок. Вложенные блоки считаются частью внешнего блока, в котором они определены. Следовательно, переменные, определенные во внешнем блоке, могут быть видны и внутри вложенного блока. Переменная внутри вложенного блока может иметь то же имя, что и переменная внутри внешнего блока. Когда подобное случается, то переменная во вложенном (внутреннем) блоке «скрывает» внешнюю переменную. Это называется **сокрытием имен.**

**16 Расширения языка C++11. Auto. Nultr. Range based циклы**

**17 Темплейты. Пример шаблонный функции.**

**шаблоны функций** — **это функции, которые служат образцом для создания других подобных функций**. Главная идея — создание функций без указания точного типа(ов) некоторых или всех переменных. Для этого мы определяем функцию, указывая **тип параметра шаблона**, который используется вместо любого типа данных. После того, как мы создали функцию с типом параметра шаблона, мы фактически создали «трафарет функции».



Плюсы шаблона заключаются в том, что их возможно использовать с разными типами данных и следовательно не нужно писать для каждой функции с разными типами переменных одно и тоже. Если же хотим использовать в шаблоне функции переменные с разными типами то пишем в <> несколько typename: **template <typename T1, typename T2>.** Это нужно, чтобы функция могла принимать разные типы шаблонных параметров.

**18 Темплейты. Пример шаблонного класса**

**Шаблоны классов** -– обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа. Представляют собой конструкции, по которым могут быть сгенерированы действительные классы путём подстановки вместо параметров конкретных аргументов. Является тем же, что и шаблон функции, т.е имеет такой же смысл, но применим уже к классам. Тем самым 2 одинаковых классов с разным типом переменных, мы можешь написать один через шаблон.

// шаблон класса, реализующего число разных типов

template <class T>

class MyNumber

{

public:

// конструктор

MyNumber(void) { }

// метод, возвращающий квадрат числа для некоторого типа T

T MySquare(T);

// метод, который делит два числа типа T и возвращает результат типа T

T DivNumbers(T, T);

};

// реализация метода, возвращающего квадрат числа

template <class T> T MyNumber<T>::MySquare(T number)

{

return (T)(number\*number);

}

// метод, который делит 2 числа и возвращает результат от деления

template <class T> T MyNumber<T>::DivNumbers(T t1, T t2)

{

return (T)(t1/t2);

}

**19 Указатели на функции и их использование совместно с массивами и структурами.**

**20 Умные указатели в библиотеке stl.**

Умный указатель — это объект, который хранит указатель на выделенный в куче объект:

* умный указатель ведет себя, как указатель
* реализуется путем перегрузки \*, -> , [ ] , и др.
* может быть полезен для управлению памятью
* логика освобождения объектов при использовании умных указателей зависит от типа умного указателя

Умные указатели могут быть разбиты на несколько категорий:

* scoped/unique — при выходе из области видимости объект удаляется
* shared (Refcounting usually) — хранят счетчик ссылок и могут ссылаться на один объект
* intrusive / interfacebased — умеют работать с классами, которые сами производят подсчет ссылок, как например объекты в COM (пример boost::intrusive)
* framework specific — обладающие специфичными для фреймвока фунцкиями (пример - QPointer служит для хранения указателя на объекты порожденные от QObject)

21 Функции. Передача параметров. По значению. По ссылке. По указателю. Использование const с параметрами.  
22 lvalue и rvalue. Move семантика.

**23 Структура программы в java. Работа с пакетами.**

### Выполнение программы. Метод main

Java является объектно-ориентированным языком, поэтому всю программу можно представить как набор взаимодействующих между собой классов и объектов. В первой главе при создании первого приложения программа была определена следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Program{        public static void main (String args[]){             System.out.println("Hello Java!");      }  } |

То есть основу нашей программы составляет класс Program. При определении класса вначале идет модификатор доступа **public**, который указывает, что данный класс будет доступен всем, то есть мы сможем его запустить из командной строки. Далее идет ключевое слово **class**, а затем название класса. После названия класса идет блок кода, в котором расположено содержимое класса.

Входной точкой в программу на языке Java является метод **main**, который определен в классе Program. Именно с него начинается выполнение программы. Он обязательно должен присутствовать в программе. При этом его заголовок может быть только таким:

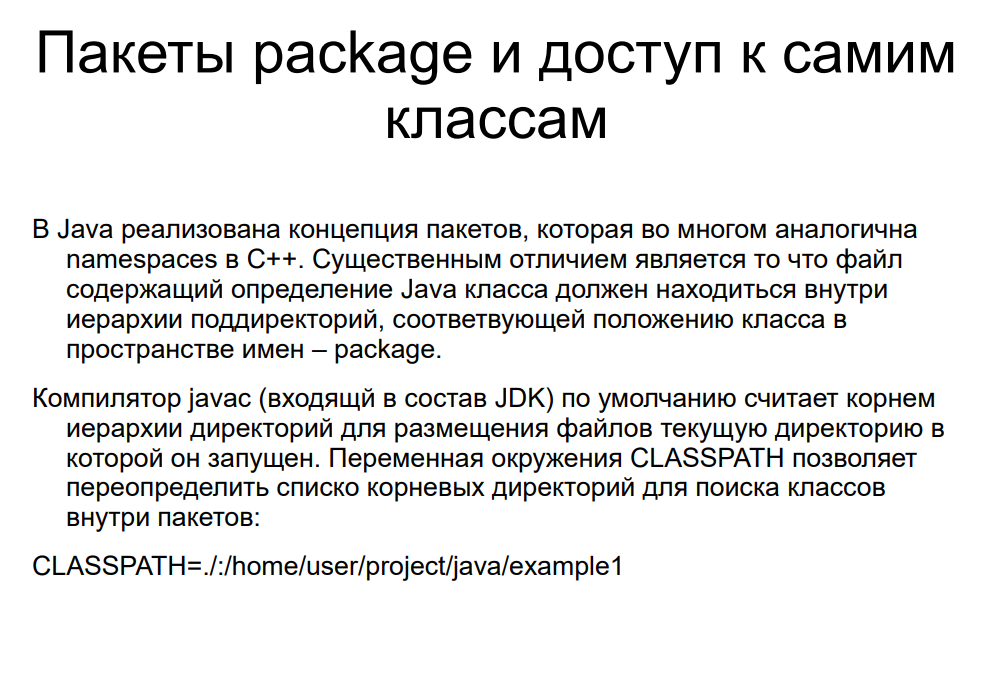
|  |  |
| --- | --- |
| 1 | public static void main (String args[]) |

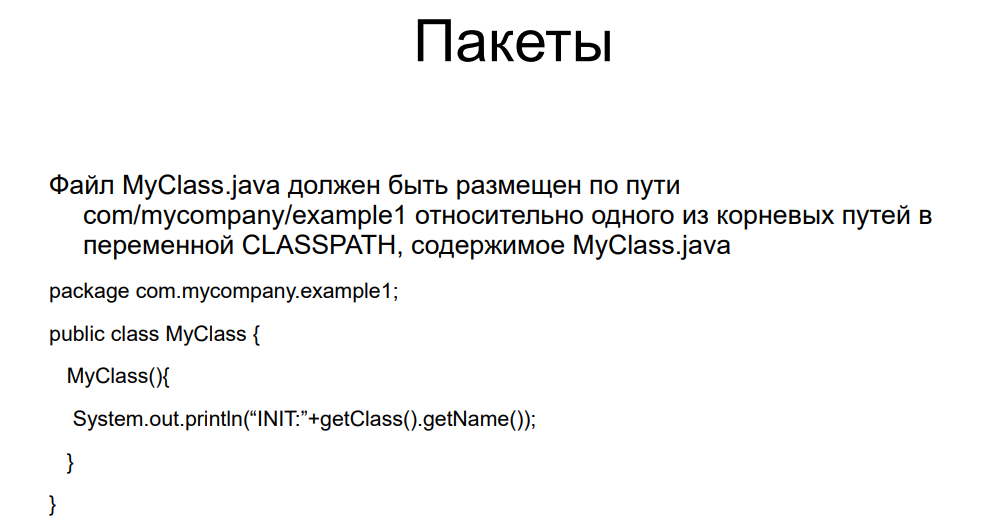
При запуске приложения виртуальная машина Java ищет в главном классе программы метод main с подобным заголовком, и после его обнаружения запускает его.

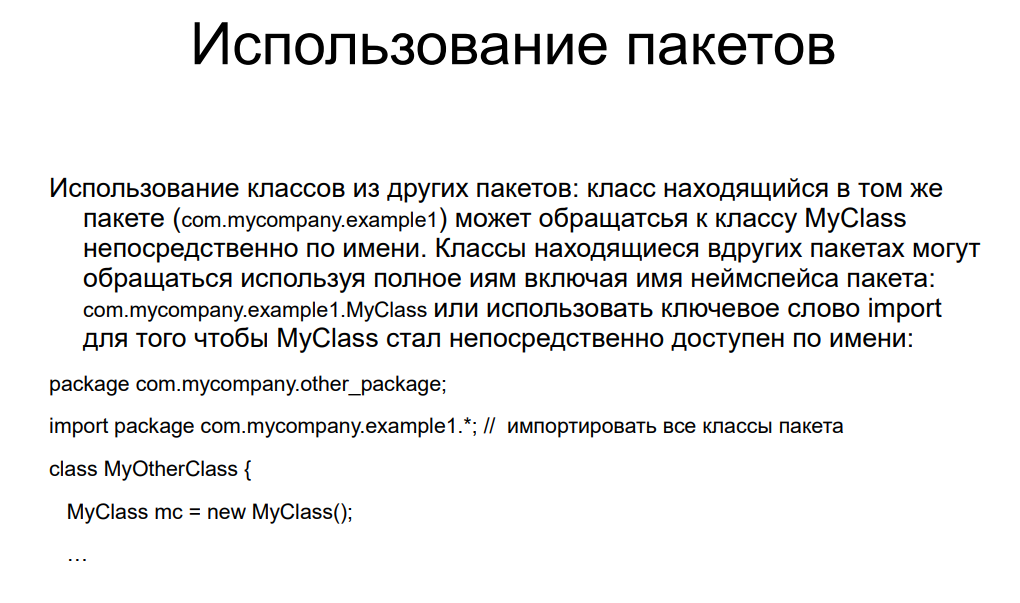
Вначале заголовка метода идет модификатор public, который указывает, что метод будет доступен извне. Слово static указывает, что метод main - статический, а слово void - что он не возвращает никакого значения. Далее в скобках у нас идут параметры метода - String args[] - это массив args, который хранит значения типа String, то есть строки. При запуске программы через этот массив мы можем передать в программу различные данные.

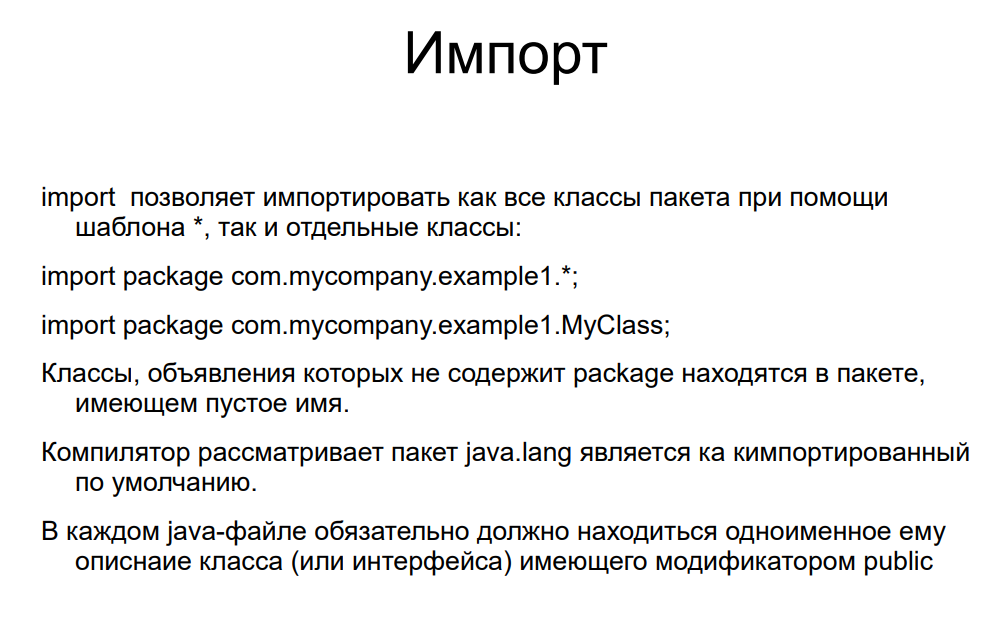
После заголовка метода идет его блок, который содержит набор выполняемых инструкций.

**Пакеты**

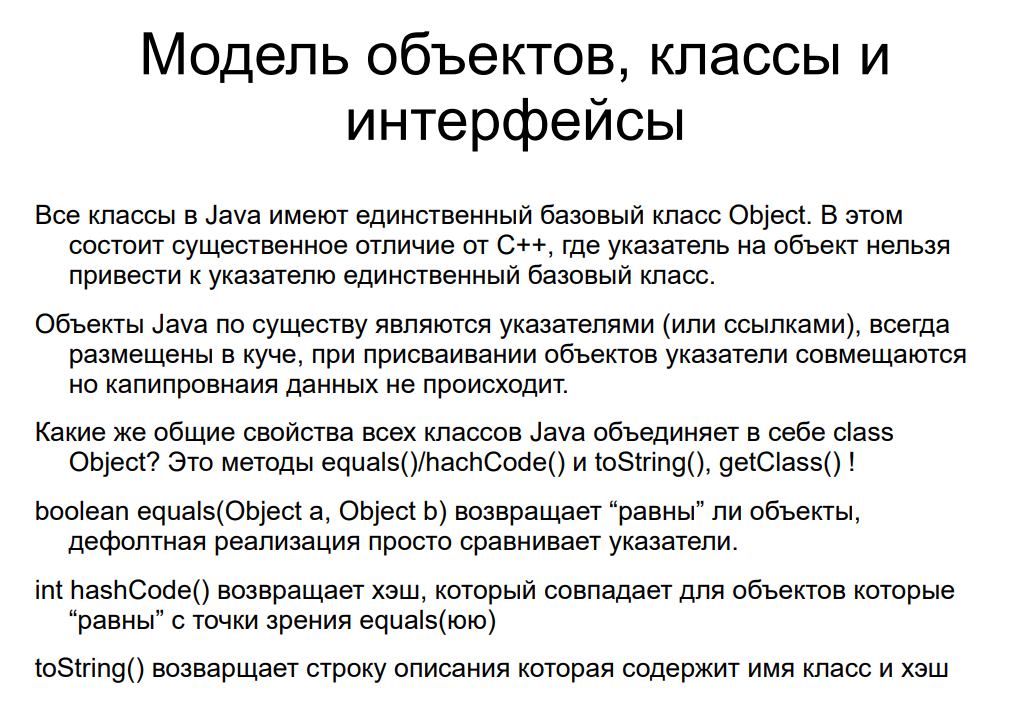
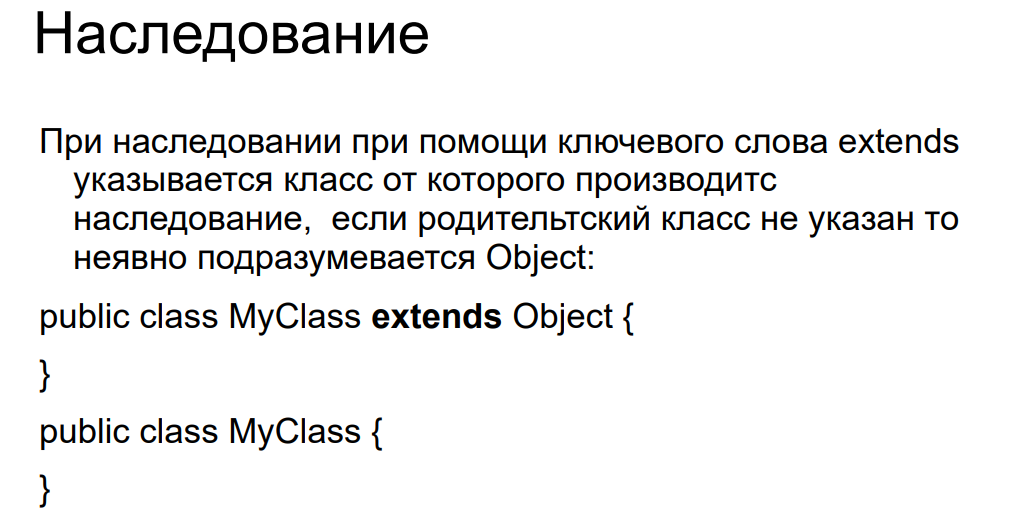


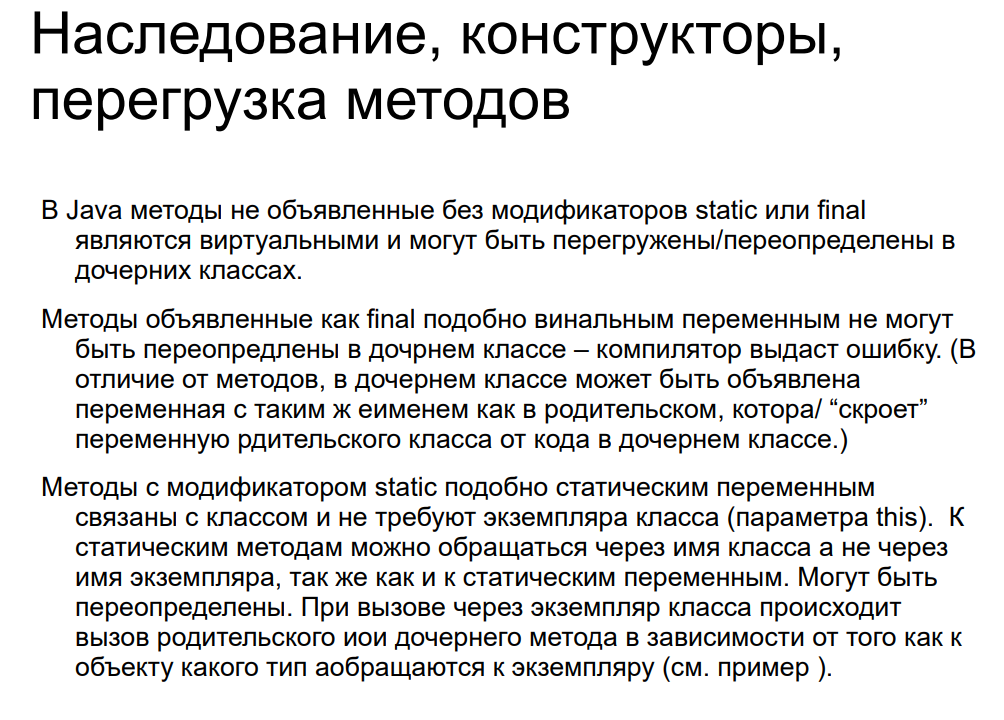
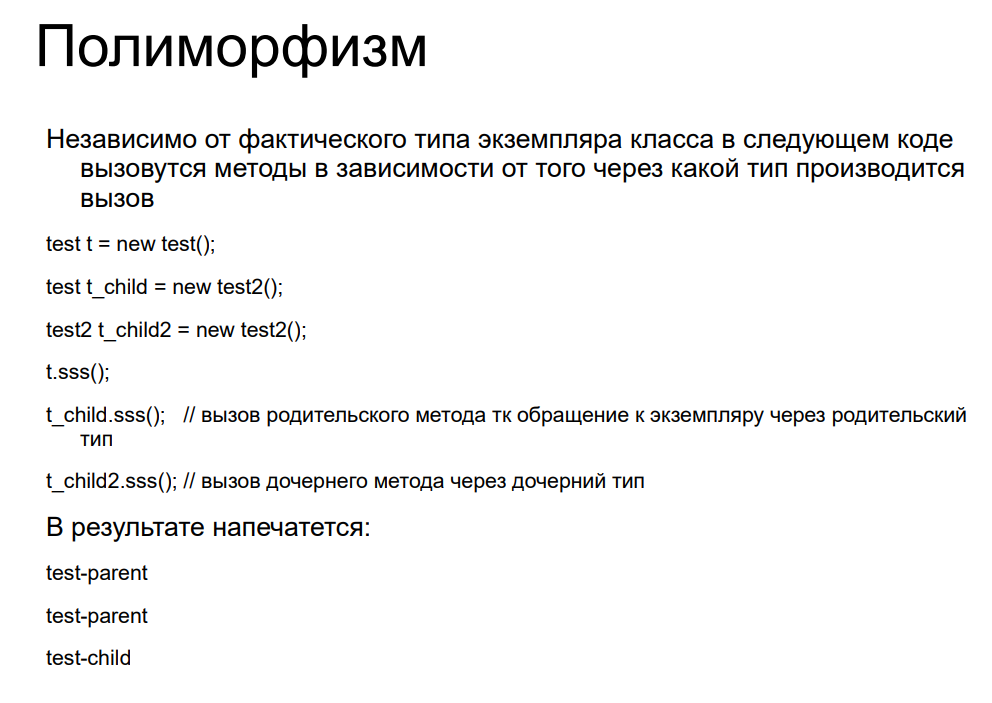




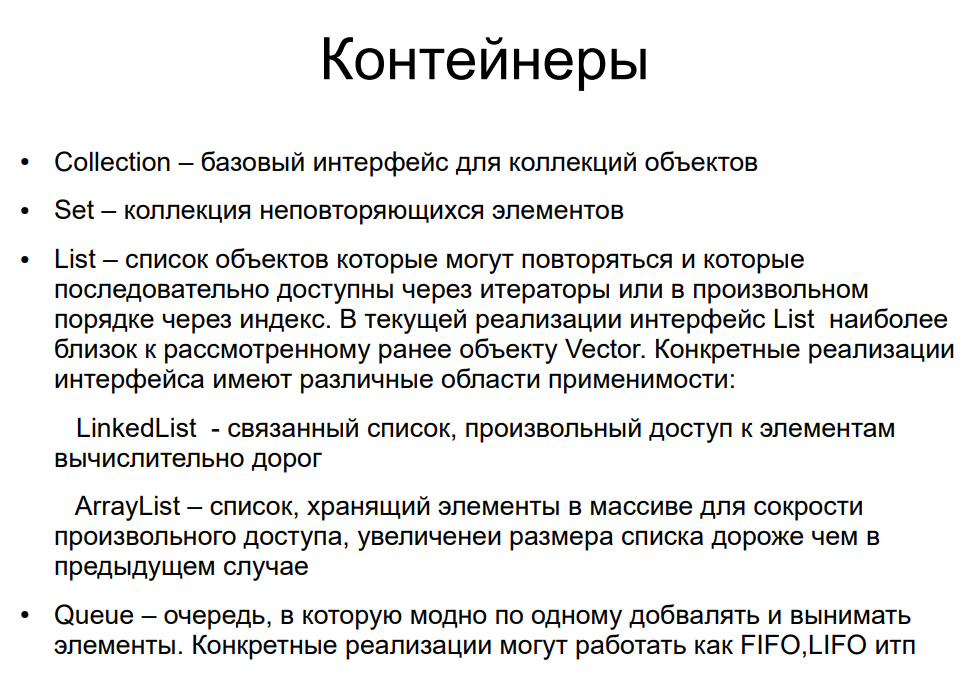


**24 ООП в java. Класс, наследование, полиморфизм, интерфейсы**

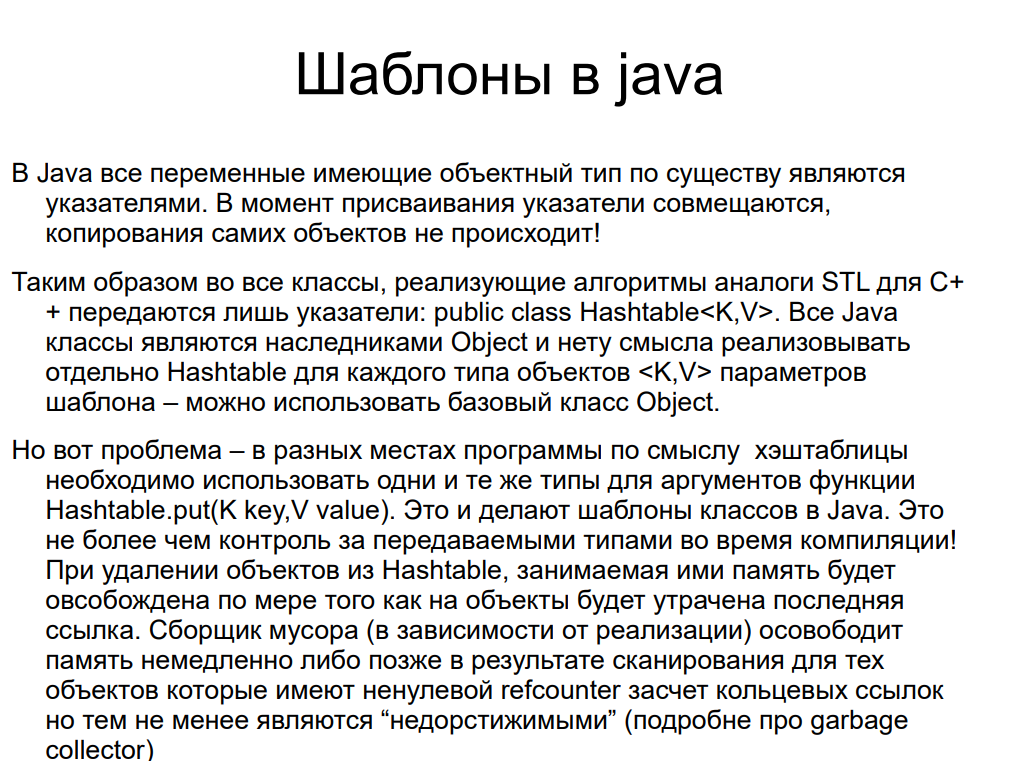
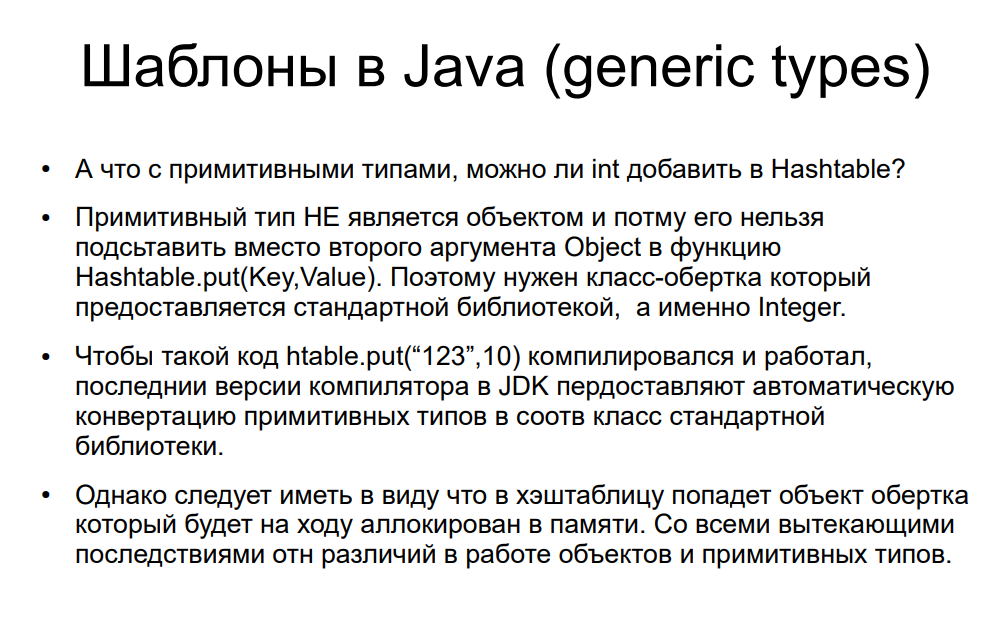
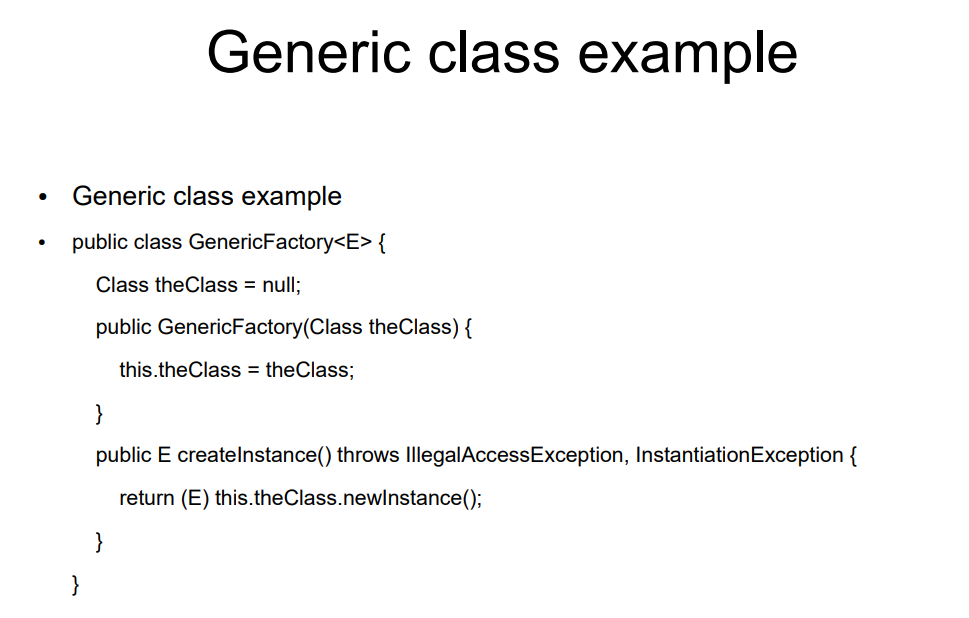
 

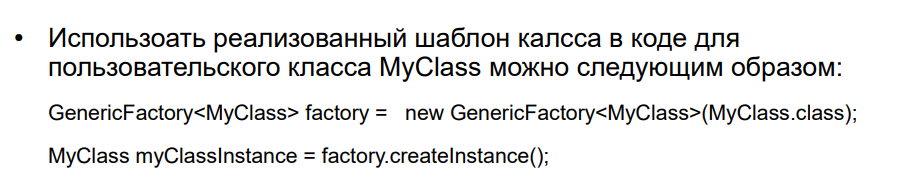
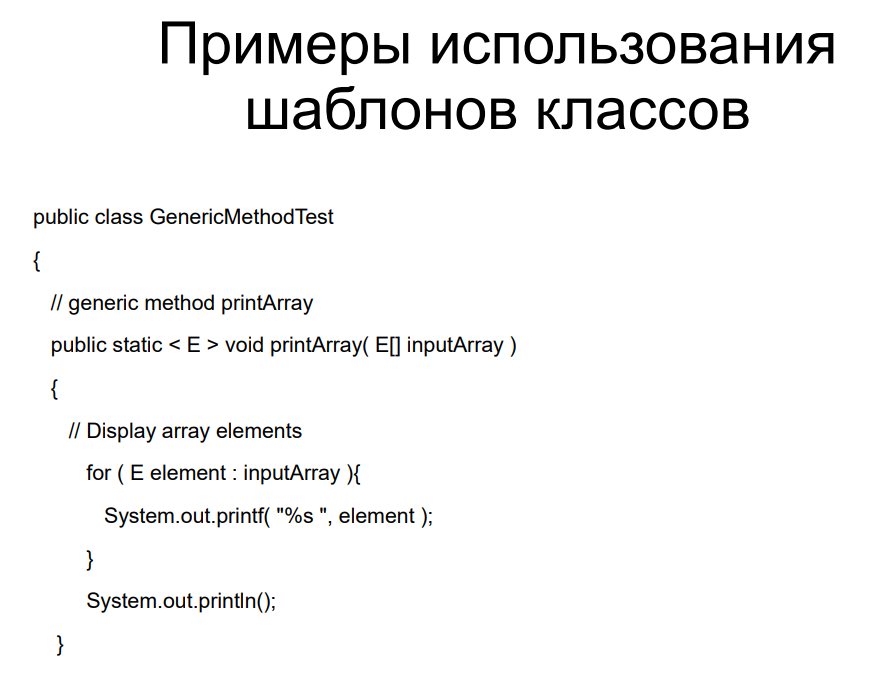
 

**25 Контейнерные классы в java**

**26 Generic-и в java**

**27 Инкапсуляция.**