# Лекция 1 Что такое Java?

# План

1. Немного истории
2. Особенности Java
3. Компилятор и интерпретатор
4. JVM
5. Hello world
6. Примитивные типы данных
7. Методы в Java
8. Java переменные
9. Ввод с клавиатуры Java языка

# Немного истории

Первая версия языка появилась еще в 1996 году в недрах компании Sun Microsystems, впоследствии поглощенной компанией Oracle. Java задумывался как универсальный язык программирования, который можно применять для различного рода задач.

# Немного истории

Java задумывался как универсальный язык программирования, который можно применять для различного рода задач. И к настоящему времени язык Java проделал большой путь, было издано множество различных версий.

Java превратилась из просто универсального языка в целую платформу и экосистему, которая объединяет различные технологии, используемые для целого ряда задач: от создания десктопных приложений до написания крупных веб-порталов и сервисов. Кроме того, язык Java активно применяется для создания программного обеспечения для множества устройств: обычных ПК, планшетов, смартфонов и мобильных телефонов и даже бытовой техники. Достаточно вспомнить популярность мобильной ОС Android, большинство программ для которой пишутся именно на Java.

# Особенности Java

Ключевой особенностью языка Java является то, что его код сначала транслируется в специальный байт-код, независимый от платформы. А затем этот байт-код выполняется виртуальной машиной **JVM** (Java Virtual Machine). В этом плане Java отличается от стандартных интерпретируемых языков как PHP или Perl, код которых сразу же выполняется интерпретатором. В то же время Java не является и чисто компилируемым языком, как C или C++.

Подобная архитектура обеспечивает кроссплатформенность и аппаратную переносимость программ на Java, благодаря чему подобные программы без перекомпиляции могут выполняться на различных платформах - Windows, Linux, Mac OS и т.д. Для каждой из платформ может быть своя реализация виртуальной машины JVM, но каждая из них может выполнять один и тот же код.

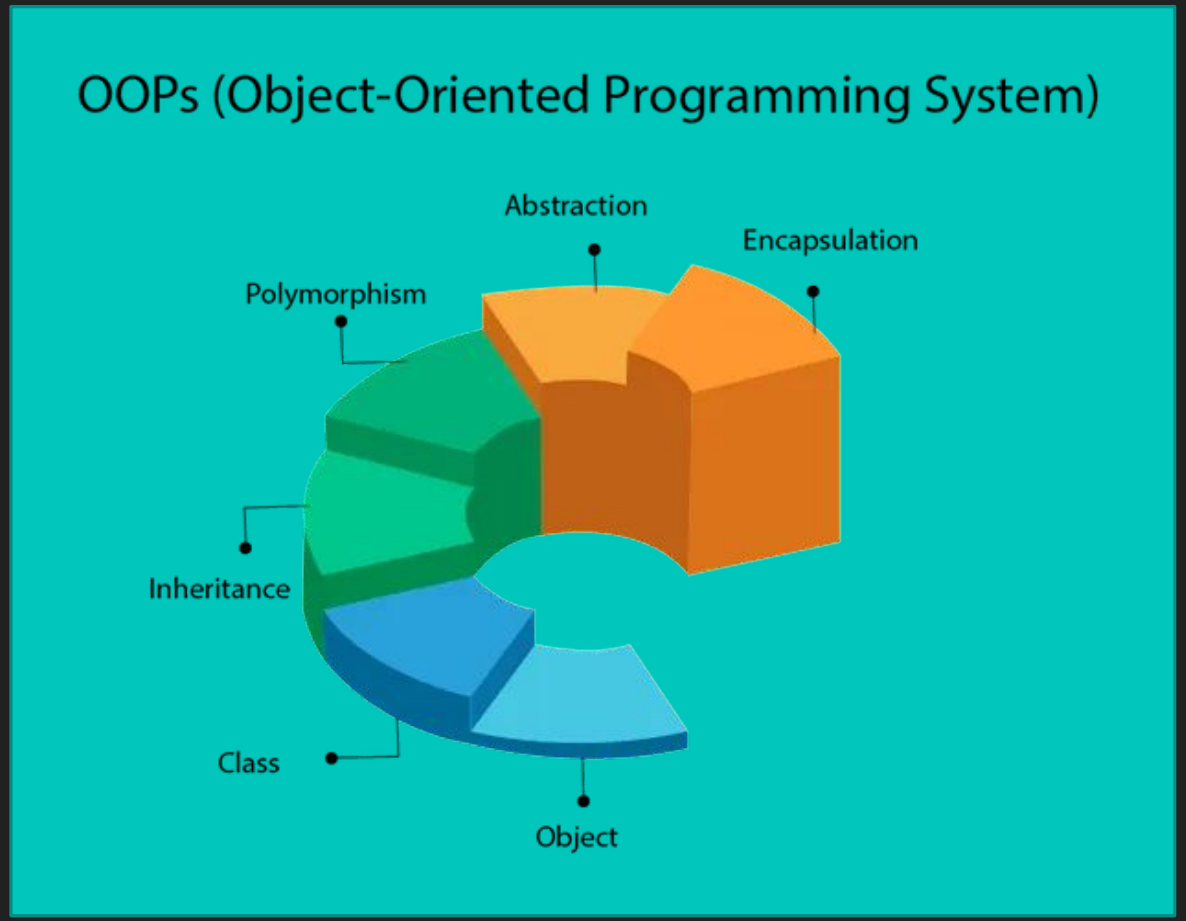
# Особенности Java

Java является языком с Си-подобным синтаксисом и близок в этом отношении к C/C++ и C#. Поэтому, если вы знакомы с одним из этих языков, то овладеть Java будет легче.

Еще одной ключевой особенностью Java является то, что она поддерживает автоматическую сборку мусора. А это значит, что вам не надо освобождать вручную память от ранее использовавшихся объектов, как в C++, так как сборщик мусора это сделает автоматически за вас.

# Особенности Java

Java является объектно-ориентированным языком. Он поддерживает полиморфизм, наследование, статическую типизацию. Объектно-ориентированный подход позволяет решить задачи по построению крупных, но в тоже время гибких, масштабируемых и расширяемых приложений.



# Компилятор и интерпретатор

Компиляторы и интерпретаторы - это трансляторы, которые преобразуют исходный код в машинный код, только разными способами. Интерпретатор читает исходный код программы и выполняет его. Преобразование исходного кода в бинарный и выполнение происходит построчно.

# Компилятор и интерпретатор

Достоинства интерпретаторов:

- Независимость от ОС (переносимость кода).
- При внесении изменений НЕ требуется перекомпиляция кода.

Недостатки интерпретаторов:

- Для запуска программы требуется наличие интерпретатора.
- Низкая скорость работы.

# Интерпретатор

Достоинства интерпретаторов:

- Независимость от ОС (переносимость кода).
- При внесении изменений НЕ требуется перекомпиляция кода.

Недостатки интерпретаторов:

- Для запуска программы требуется наличие интерпретатора.
- Низкая скорость работы.

# Компилятор

Достоинства компиляторов:

- Быстрота работы программ;
- Отсутствие надобности компилятора на компьютере пользователя.

Недостатки компиляторов:

- Программа зависит от ОС, под которую была скомпилирована.
- При внесении изменений требуется перекомпиляция кода.

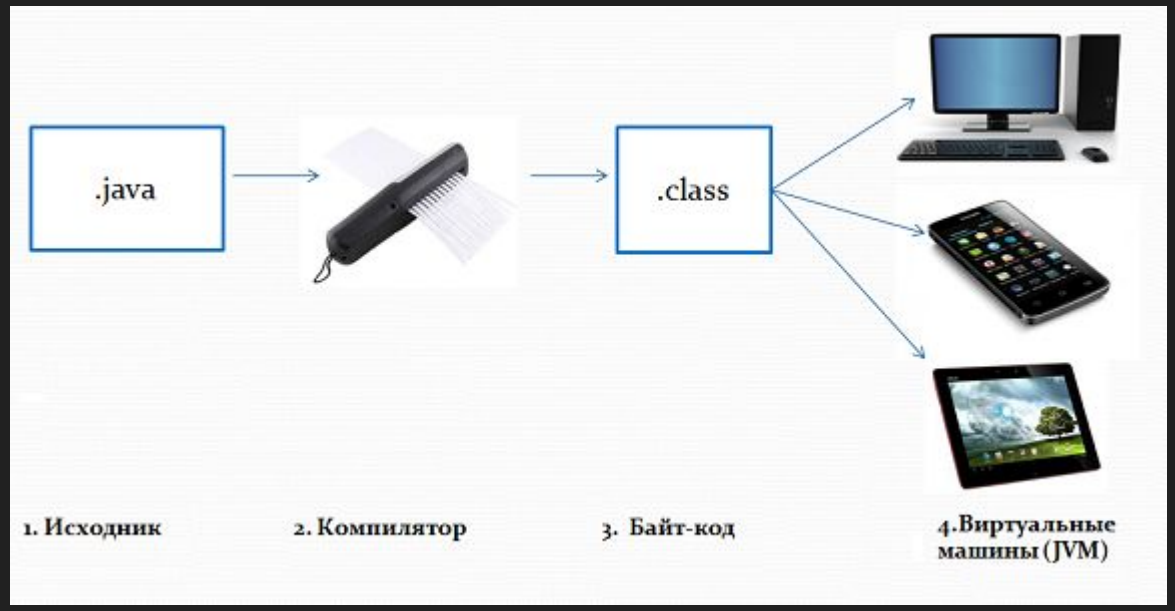
# Байт-код

Все языки можно условно разделить на компилируемые и интерпретируемые. В Java используется третий подход — байт-код. Исходный код Java преобразуется компилятором в байт-код (а не машинный код). А байт-код Java преобразуется в машинный код с помощью специального интерпретатора, называемого виртуальной машиной Java (Java Virtual Machine — JVM).

# Байт-код

Рассмотрим более детально как работает Java:

1. Создается исходный документ (исходник) – файл с расширением .java.
2. Исходник пропускается через компилятор, который проверяет код на ошибки и выдает конечный результат.
3. Компилятор создает новый документ, закодированный с помощью байт-кода. Любое устройство, способное выполнять Java, сможет интерпретировать этот файл в такой формат, который сможет запустить. Скомпилированный байт-код не зависит от платформы.
4. Виртуальная машина считывает и выполняет байт-код.



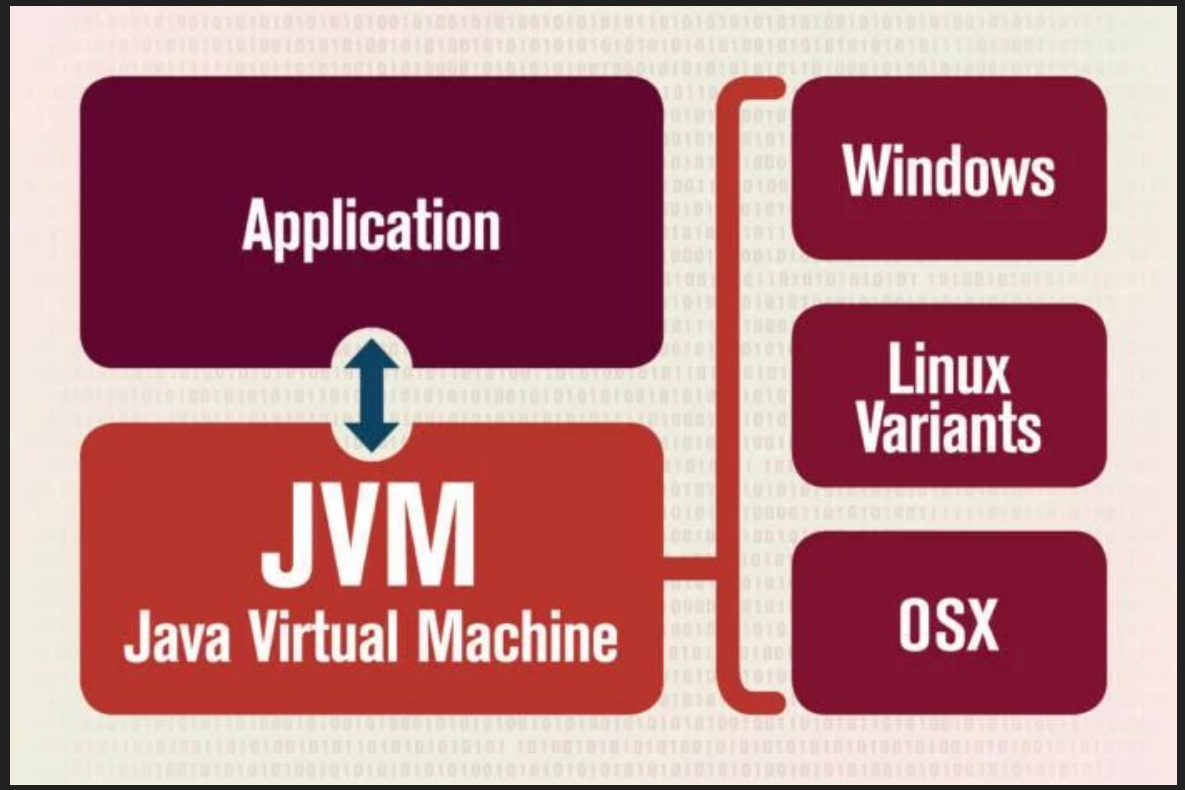
# JVM

JVM имеет две основные функции:

- Позволяет запускать Java приложения на любых устройствах или операционных системах (принцип – «Написал один раз, запускай везде»)
- Управляет и оптимизирует память, используемую приложением

# JVM

В 1995 году, когда Java появилась, все компьютерные программы были написаны под определенные операционные системы и управлять памятью приходилось разработчику программного обеспечения. Так что появление JVM было революцией.



# JVM

Существует техническое определение JVM, а также его повседневная формулировка:

- Техническое определение: JVM – это программное обеспечение, которое выполняет код и предоставляет среду выполнения для этого кода
- Повседневная формулировка: JVM – это способ запуска наших Java приложений. Мы настраиваем параметры JVM, а затем полагаемся на нее автоматическое управление ресурсами программы во время выполнения

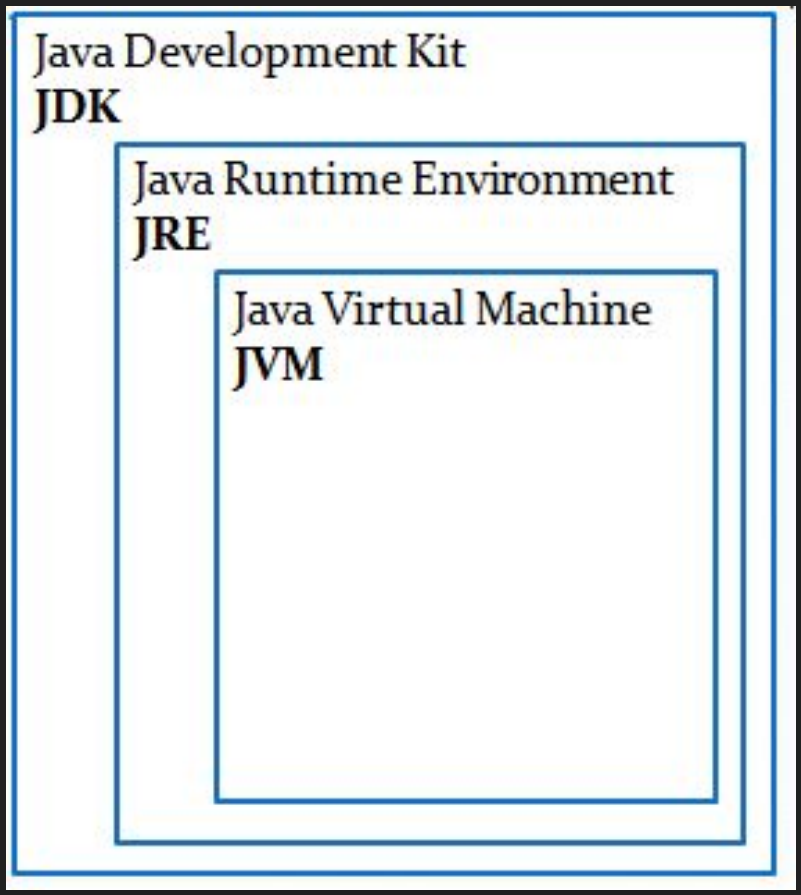
Когда разработчики говорят о JVM, обычно имеют в виду процесс, запущенный на устройстве, который предоставляет и контролирует использование ресурсов Java приложением. Спецификация JVM описывает требования для разработки программ, выполняющих эти задачи.

# JVM

**JVM (Java Virtual Machine)** - виртуальная машина Java - основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). Виртуальная машина Java исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java (javac). JVM обеспечивает платформо-независимый способ выполнения кода. Программисты могут писать код не задумываясь как и где он будет выполняться.

**JRE (Java Runtime Environment)** - минимальная реализация виртуальной машины, необходимая для исполнения Java - приложений, без компилятора и других средств разработки. Состоит из виртуальной машины и библиотек Java классов.

**JDK (Java Development Kit)** - комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов Java, примеры, документацию, различные утилиты и исполнительную систему JRE.



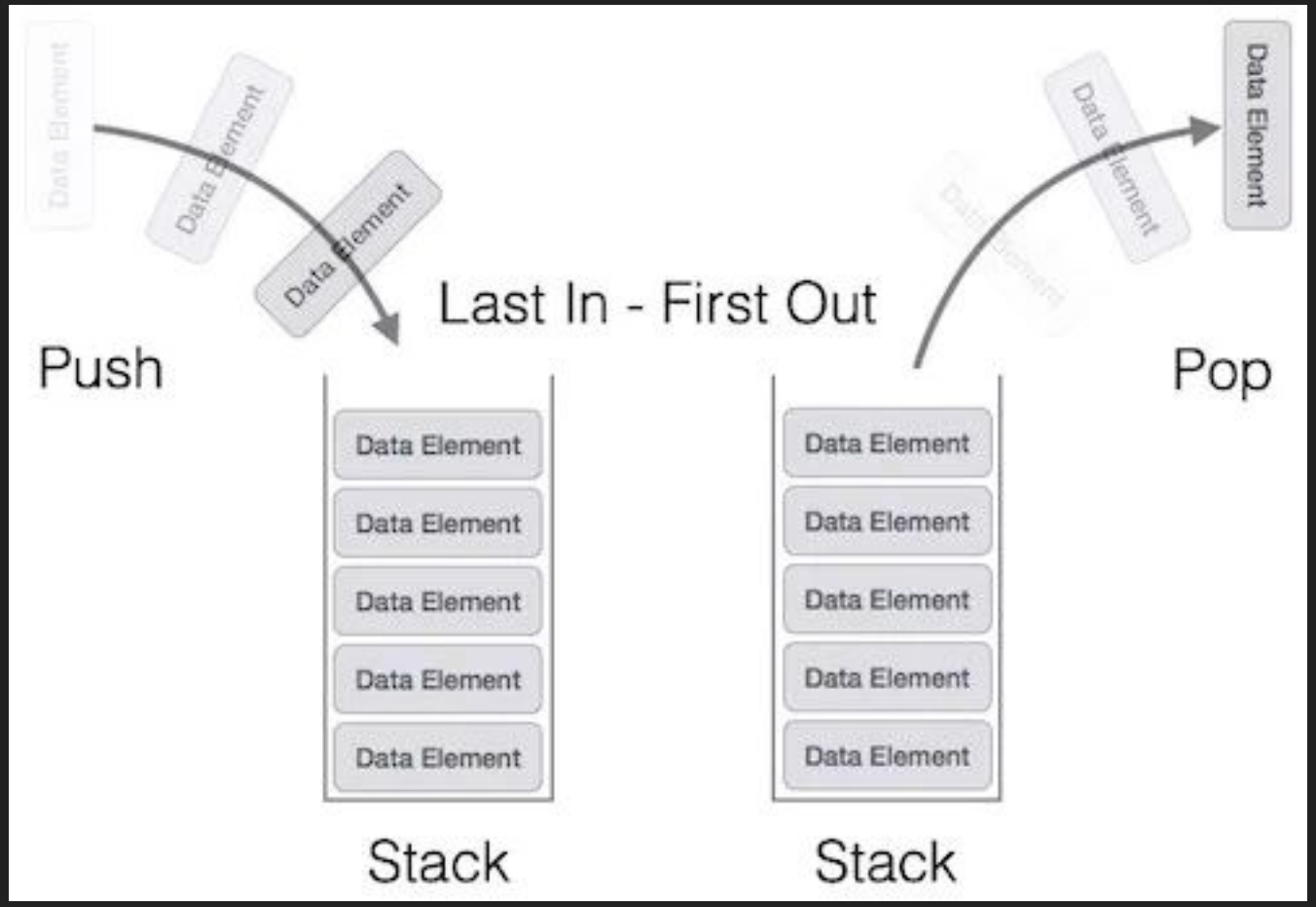
# Стек

Для оптимальной работы приложения JVM делит память на область стека (stack) и область кучи (heap). Всякий раз, когда мы объявляем новые переменные, создаем объекты или вызываем новый метод, JVM выделяет память для этих операций в стеке или в куче.

# Стек

Стек работает по схеме LIFO (последним вошел, первым вышел). Всякий раз, когда вызывается новый метод, содержащий примитивные значения или ссылки на объекты, то на вершине стека под них выделяется блок памяти.

Когда метод завершает выполнение, блок памяти, отведенный для его нужд, очищается, и пространство становится доступным для следующего метода.



# Стек

**Помимо того, что мы рассмотрели, существуют и другие особенности стека:**

- Он заполняется и освобождается по мере вызова и завершения новых методов
- Переменные в стеке существуют до тех пор, пока выполняется метод в котором они были созданы
- Если память стека будет заполнена, Java бросит исключение java.lang.StackOverFlowError
- Доступ к этой области памяти осуществляется быстрее, чем к куче
- является потокобезопасным, поскольку для каждого потока создается свой отдельный стек

# Куча

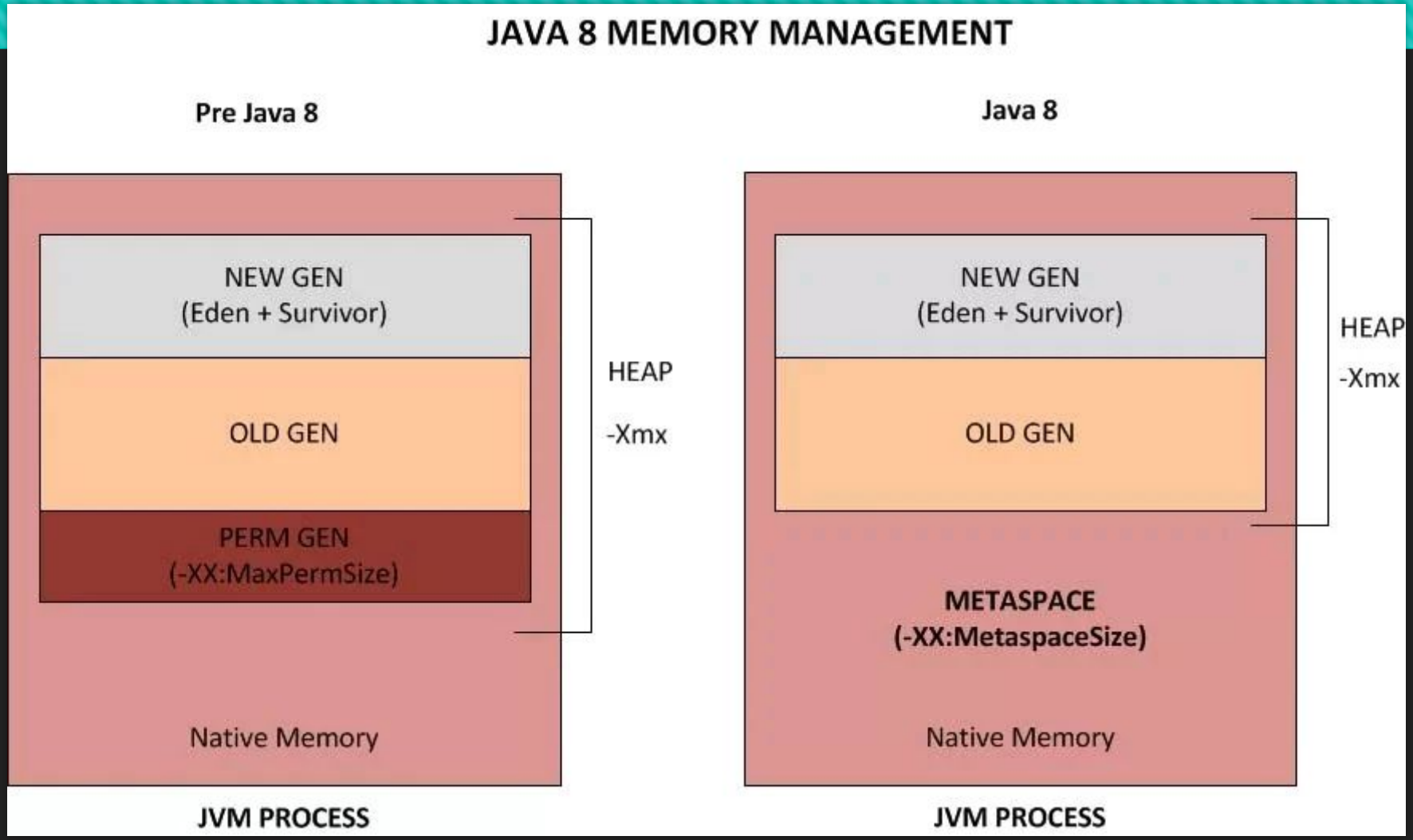
Эта область памяти используется для объектов и классов. Новые объекты всегда создаются в куче, а ссылки на них хранятся в стеке.

Эти объекты имеют глобальный доступ и могут быть получены из любого места программы.

**Эта область памяти разбита на несколько более мелких частей, называемых поколениями:**

- **Young Generation** — область где размещаются недавно созданные объекты. Когда она заполняется, происходит быстрая сборка мусора
- **Old (Tenured) Generation** — здесь хранятся долгоживущие объекты. Когда объекты из Young Generation достигают определенного порога "возраста", они перемещаются в Old Generation
- **Permanent Generation** — эта область содержит метаинформацию о классах и методах приложения, но начиная с Java 8 данная область памяти была упразднена.

# Куча


JAVA 8 MEMORY MANAGEMENT
Pre Java 8
NEW GEN
(Eden + Survivor)
OLD GEN
PERM GEN
(-XX:MaxPermSize)
HEAP
-Xmx
Native Memory
JVM PROCESS
Java 8
NEW GEN
(Eden + Survivor)
OLD GEN
HEAP
-Xmx
METASPACE
(-XX:MetaspaceSize)
Native Memory
JVM PROCESS

# Куча

Eden Space (heap) – в этой области выделятся память под все создаваемые из программы объекты. Большая часть объектов живет недолго (итераторы, временные объекты, используемые внутри методов и т.п.), и удаляются при выполнении сборок мусора это области памяти, не перемещаются в другие области памяти. Когда данная область заполняется (т.е. количество выделенной памяти в этой области превышает некоторый заданный процент), GC выполняет быструю (minor collection) сборку мусора. По сравнению с полной сборкой мусора она занимает мало времени, и затрагивает только эту область памяти — очищает от устаревших объектов Eden Space и перемещает выжившие объекты в следующую область.

# Куча

Survivor Space (heap) – сюда перемещаются объекты из предыдущей, после того, как они пережили хотя бы одну сборку мусора. Время от времени долгоживущие объекты из этой области перемещаются в Tenured Space.

# Куча

Tenured (Old) Generation (heap) — Здесь скапливаются долгоживущие объекты (крупные высокоуровневые объекты, синглтоны, менеджеры ресурсов и проч.). Когда заполняется эта область, выполняется полная сборка мусора (full, major collection), которая обрабатывает все созданные JVM объекты.

# Куча

**Помимо рассмотренных ранее, куча имеет следующие ключевые особенности:**

- Когда эта область памяти полностью заполняется, Java бросает java.lang.OutOfMemoryError
- Доступ к ней медленнее, чем к стеку
- Эта память, в отличие от стека, автоматически не освобождается. Для сбора неиспользуемых объектов используется сборщик мусора
- В отличие от стека, куча не является потокобезопасной и ее необходимо контролировать, правильно синхронизируя код

# Куча

| Свойства | Стек | Куча |
|---|---|---|
| Использование приложением | Для каждого потока используется свой стек | Пространство кучи является общим для всего приложения |
| Размер | Предел размера стека определен операционной системой | Размер кучи не ограничен |
| Хранение | Хранит примитивы и ссылки на объекты | Все созданные объекты хранятся в куче |
| Порядок | Работает по схеме последним вошел, первым вышел (LIFO) | Доступ к этой памяти осуществляется с помощью сложных методов управления памятью, включая Young Generation, Old и Permanent Generation |

# Куча

| Свойства | Стек | Куча |
| --- | --- | --- |
| Существование | Память стека существует пока выполняется текущий метод | Пространство кучи существует пока работает приложение |
| Скорость | Обращение к памяти стека происходит значительно быстрее, чем к памяти кучи | Медленнее, чем стек |
| Выделение и освобождение памяти | Эта память автоматически выделяется и освобождается, когда метод вызывается и завершается соответственно | Память в куче выделяется, когда создается новый объект и освобождается сборщиком мусора, когда в приложении не остается ни одной ссылки на его |

# Hello world

Данный пример находится в исходном файле с именем MyFirstApp.java. Данная программа выводит сообщение "Hello world!!!" на консоль.

```
public class MyFirstApp {
    public static void main(String[] args) {
        System.out.print("Hello world!!!");
    }
}
```

# Основные правила по написанию приложений на Java

1. В Java исходный файл называется единицей компиляции. Он представляет собой текстовый файл, содержащий определения одного или нескольких классов. (Мы будем пока что пользоваться исходными файлами, содержащими только один класс.)
2. Компилятор Java требует, чтобы исходный файл имел расширение .java.
3. По принятому соглашению имя главного класса должно совпадать с именем файла, содержащего исходный код программы.
4. В Java весь код должен размещаться в классе.
5. Java учитывает регистр символов!
6. Все определение класса, в том числе его членов, должно располагаться между открывающей ( { ) и закрывающей ( } ) фигурными скобками.

# Основные правила по написанию приложений на Java

1. Выполнение всех прикладных программ на Java начинается с вызова метода main().
2. Правильные объявления метода *main()*:

```
public static void main(String[] args)
static public void main(String[] args)
public static void main(String... x)
static public void main(String someArgs[])
```

3. Компилятор Java скомпилирует классы, в которых отсутствует метод main(), но загрузчик приложений (java) не сможет выполнить код таких классов.
4. Для передачи любой информации, требующейся методу, служат переменные, указываемые в скобках вслед за именем метода. Эти переменные называются параметрами. Если параметры не требуются методу, то указываются пустые скобки.
5. В языке Java все операторы обычно должны оканчиваться точкой с запятой.
6. Большинство пробелов, табуляций, символов переноса строки и так далее игнорируются.

# Добавление класса в пакет

Обычно проект содержит большое количество классов и держать их в одном каталоге крайне неудобно. Кроме того, может возникнуть ситуация, когда два программиста создали класс с одинаковым названием. Для решения этих проблем в Java существует такой механизм как пакеты. Пакеты по своей сути очень похожи на каталоги файловой системы и должны совпадать с ней.

# Добавление класса в пакет

Существуют также правила для наименования пакетов. Для коммерческих проектов пакет должен начинаться с com, потом следует имя организации и название проекта. Потом пакеты обычно именуются по какому-то функциональному признаку.

Кроме разделения пространств имен классов, пакеты также служат для управления доступностью объектов. В пакете можно определить классы, недоступные для кода за пределами этого пакета. В нем можно также определить члены класса, доступные только другим членам этого же пакета. Благодаря такому механизму классы могут располагать полными сведениями друг о друге, но не предоставлять эти сведения остальному миру.

# Добавление класса в пакет

Хорошей практикой считается добавлять классы в пакеты. Но так как полное имя класса включает в себя имя пакета, в коде это может привести к достаточно длинным строкам, что крайне неудобно. Для решения этой проблемы в Java придуман такой механизм как импорт. Оператор import позволяет импортировать класс, после чего к нему можно обращаться просто по имени.

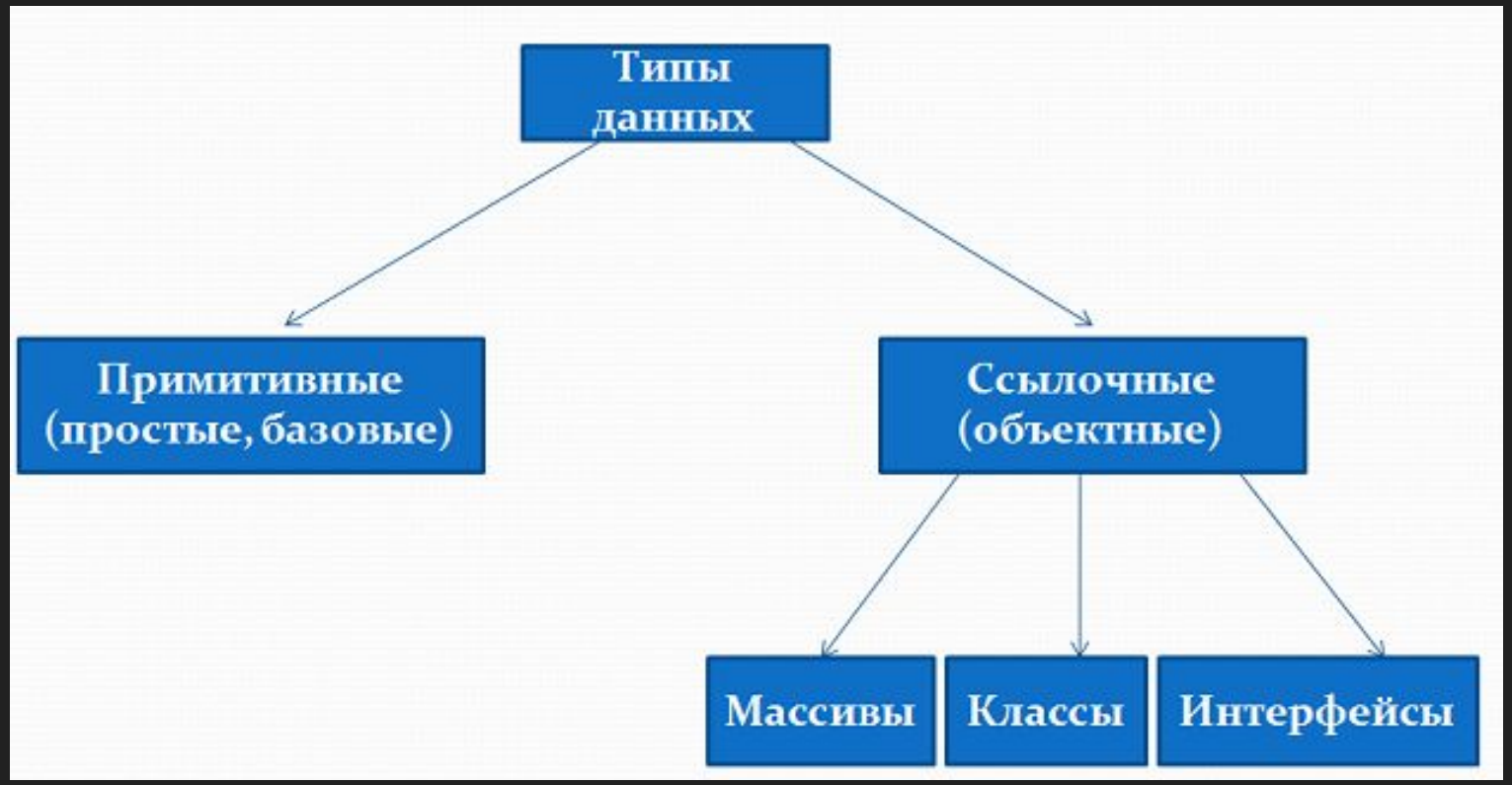
# Добавление класса в пакет

Существует один пакет, классы которого импортируются в код по умолчанию. Это пакет *java.lang*, в котором находятся наиболее часто используемые классы.

Вы можете импортировать два пакета, в которых находятся классы с одинаковыми именами. Но обращаться к ним по короткому имени будет ошибкой - компилятор не сможет определить какой именно класс вам нужен. Поэтому к таким классам придется обращаться только по полному имени.

# Примитивные типы данных

Типы в Java распределены на две категории: примитивные (простые) и ссылочные (объектные). Ссылочные типы - это массивы, классы и интерфейсы.

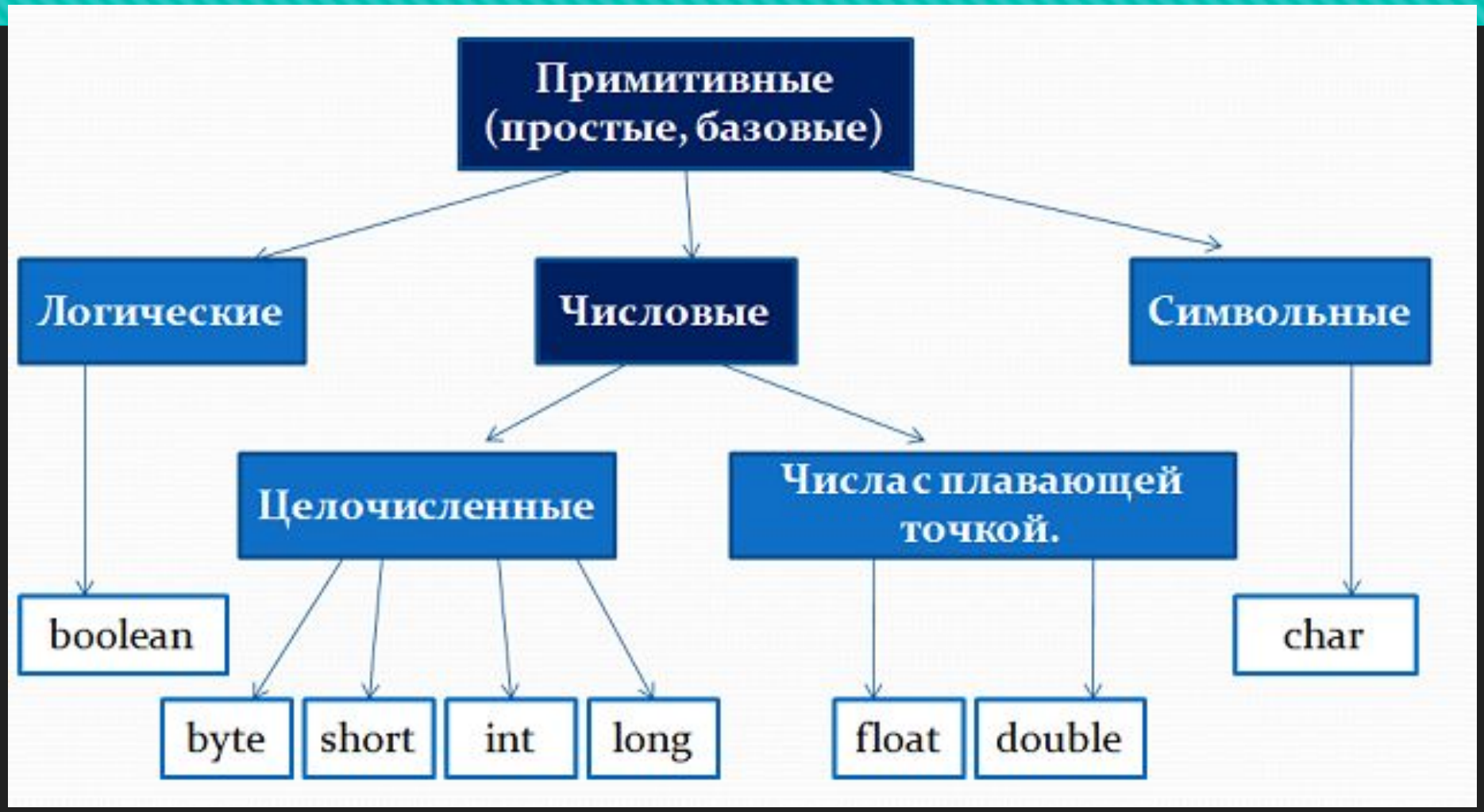


# Примитивные типы данных

Примитивные типы можно разделить на следующие четыре группы:

- Целые числа. Эта группа включает в себя типы данных byte, short, int и long, представляющие целые числа со знаком.
- Числа с плавающей точкой. Эта группа включает в себя типы данных float и double, представляющие числа с точностью до определенного знака после десятичной точки.
- Символы. Эта группа включает в себя тип данных char, представляющий символы, например буквы и цифры, из определенного набора.
- Логические значения. Эта группа включает в себя тип данных boolean, специально предназначенный для представления логических значений.

# Примитивные типы данных



# Примитивные типы данных

**Типы byte, short, int, long**

Целочисленные числа представлены в языке Java четырьмя типами - **byte**, **short**, **int** и **long**.

- **int** — основной целочисленный тип, используемый в Java по умолчанию. Любое целое число будет рассматриваться компилятором как число типа **int**. Используется в качестве счётчика циклов, индексов массивов и индексов символов в строках.
- **long** — целочисленный тип содержащий практически бесконечное количество значений. Используется в случаях, где числа превосходят 2 миллиарда и стандартного int уже не хватает. Используется в повседневной жизни для создания уникальных значений.
- **byte** — используется для передачи данных по сети, записи и чтения из файла. В математических операциях, как правило, не используется.
- **short** — самый редко используемый тип в Java, может использоваться только в целях экономии памяти.

# Примитивные типы данных

| Тип | Размер в байтах | Размер в битах | Возможные значения (от..до) | Значение по умолчанию |
|---|---|---|---|---|
| byte | 1 | 8 | -128..127 | 0 |
| short | 2 | 16 | -32,768..32,767 | 0 |
| int | 4 | 32 | -2,147,483,648..2,147,483,647 | 0 |
| long | 8 | 64 | -9,223,372,036,854,775,808 ..9,223,372,036,854,775,807 | 0 |

# Примитивные типы данных

**Типы byte, short, int, long**

Целочисленные числа представлены в языке Java четырьмя типами - **byte**, **short**, **int** и **long**.

- **int** — основной целочисленный тип, используемый в Java по умолчанию. Любое целое число будет рассматриваться компилятором как число типа **int**. Используется в качестве счётчика циклов, индексов массивов и индексов символов в строках.
- **long** — целочисленный тип содержащий практически бесконечное количество значений. Используется в случаях, где числа превосходят 2 миллиарда и стандартного int уже не хватает. Используется в повседневной жизни для создания уникальных значений.
- **byte** — используется для передачи данных по сети, записи и чтения из файла. В математических операциях, как правило, не используется.
- **short** — самый редко используемый тип в Java, может использоваться только в целях экономии памяти.

# Примитивные типы данных

Числа с плавающей точкой (или действительные числа) представлены типами float и double. Используются для хранения значений с точностью до определенного знака после десятичной точки.

double — это числа с двойной точностью, максимально приближённые к заданным или полученным в результате вычислений значениям. Используется в Java для любых математических вычислений (квадратный корень, синус, косинус,..).

float — менее точный тип с плавающей точкой. Используется очень редко с целью экономии памяти.

| Тип | Размер в байтах | Размер в битах | Возможные значения (от..до) | Значение по умолчанию |
|---|---|---|---|---|
| float | 4 | 32 | -3.4E+38..3.4E+38 (стандарт IEEE 754) | 0.0 |
| double | 8 | 64 | -1.7E+308..1.7E+308 (стандарт IEEE 754) | 0.0 |

# Примитивные типы данных

В языке Java есть три специальных числа плавающей точкой, которые используются для обозначения переполнения и ошибок:

- положительная бесконечность - результат деления положительного числа на 0. Представлены константами Double.POSITIVE_INFINITY и Float.POSITIVE_INFINITY.
- отрицательная бесконечность - результат деления отрицательного числа на 0. Представлены константами Double.NEGATIVE_INFINITY и Float.NEGATIVE_INFINITY.
- NaN (не число) - вычисление 0/0 или извлечение квадратного корня из отрицательного числа. Представлены константами Double.NaN и Float.NAN.

# Примитивные типы данных

**Java тип char**

Символы описываются в языке Java **char** типом. Символы преобразуются по таблице кодировки UTF-16. По большому счёту это все буквы, числа и специальные символы существующие на нашей планете.

*Размер в байтах* - 2 байта

*Возможные значения (от..до)* - 0..65,535

*Значение по умолчанию* - '\u0000'

Тип **char** является псевдоцелочисленным типом, поэтому значения этого типа можно задавать в виде числа - кода символа из таблицы кодировки UTF-16. Каждому символу соответствует определённое число из таблицы и Java при виде этого числа в рамках типа **char** выводит его на экран как символ.

# Примитивные типы данных

**Java тип boolean**

Примитивный тип boolean предназначен для хранения логических значений. Логические переменные этого типа могут принимать только два значения: true – истина и false – ложь. Памяти на переменную такого типа требуется 1 бит.

# Методы в Java

Практически весь код в Java пишется в методах. Рассмотрим синтаксис написания методов.

Общая форма объявления метода:

```
тип имя(список_параметров){
    // тело метода
}
```

Существует также такое понятие как сигнатура метода Java языка - это имя метода и его параметры. Возвращаемый тип не входит в сигнатуру.

# Методы в Java

**Методы в Java не возвращающие значение**

В следующем примере метод print не принимает на вход никаких значений - список параметров у него пустой. Возвращаемый тип у него void - это значит, что он ничего не возвращает.

Метод выводит на консоль сообщение "Print some info".

```
static void print() {
     System.out.println("Print some info");
 }
```

# Методы в Java

**Методы в Java возвращающие значение**

Метод getVolume принимает на вход три параметра типа double, а также возвращает значение типа double. Метод возвращает значение с помощью ключевого слова return:

```
static double getVolume(double width, double height, double depth) {
        return width * height * depth;
 }
```

# Методы в Java

**Тип метода**

Тип обозначает конкретный тип данных, возвращаемых методом. Он может быть любым допустимым типом данных, в том числе и типом созданного класса.

Если метод не возвращает значение, то его возвращаемым типом должен быть void.

Методы, возвращаемый тип которых отличается от void, возвращают значение: return значение;

# Методы в Java

**Имя и параметры метода**

Для указания имени метода служит идентификатор имя. Это может быть любой допустимый идентификатор, кроме тех, которые уже используются другими элементами кода в текущей области действия.

Список параметров обозначает последовательность пар "тип-идентификатор", разделенных запятыми. По существу, параметры - это переменные, которые принимают значения аргументов, передаваемых методу во время его вызова. Если у метода отсутствуют параметры, то список_параметров оказывается пустым.

# Методы в Java

```
public class SquareDemo {
    public static void main(String[] args) {
        int x, y;
        x = square(5);
        System.out.println(x);
        x = square(9);
        System.out.println(x);
        y = 2;
        x = square(y);
        System.out.println(x);
    }

    public static int square(int i) {
        return i * i;
    }
}
```

**Параметр и аргумент**

Важно различать два термина: параметр и аргумент.

Параметр - это переменная, определенная методом, которая принимает значение при вызове метода.

Аргумент - это значение, передаваемое методу при его вызове. Например, square(100) передает 100 в качестве аргумента. Внутри метода square() параметр i получает это значение.

# Java переменные

**Простая форма объявления Java переменных:**

```
тип идентификатор;
```

**Общая форма объявления Java переменных и их инициализации:**

```
тип идентификатор [=значение] [, идентификатор  [=значение] ... ] ;
```

Где параметр тип обозначает один из примитивных типов данных в Java, имя класса или интерфейса. А идентификатор - это имя переменной.

# Java переменные

- Java - строго типизированный язык. Каждая переменная в Java имеет конкретный тип, который определяет размер и размещение её в памяти; диапазон значений, которые могут храниться в памяти; и набор операций, которые могут быть применены к переменной.
- Необходимо объявить все переменные, прежде чем их использовать.
- Если требуется объявить несколько переменных заданного типа, это можно сделать в виде разделенного запятыми списка имен переменных.
- Переменная может быть инициализирована при объявлении или позже.
- Во время выполнения программы значение переменной может изменяться.
- Инициализирующее выражение должно возвращать значение того же самого (или совместимого) типа, что и у переменной.

# Java переменные

**Пример объявления и инициализации переменных**

```
double d = 2.3;// Объявляет число d и инициализирует его
int count = 0, min = 6;// Объявляет два целых числа count и min
int max;// Объявляет число max
max = 89;// Инициализирует число max
```

# Java переменные

**Область видимости переменной в Java**

В Java допускается объявление переменных в любом блоке кода. Блок кода заключается в фигурные скобки, задавая тем самым область видимости или действия. Таким образом, при открытии каждого нового блока кода создается новая область видимости.

- Две основные области видимости в Java определяются классом и методом, хотя такое их разделение несколько искусственно.
- В языке Java видимость переменных, определяемая методом, начинается с его открывающей фигурной скобки. Если у метода имеются параметры, то они также включаются в область видимости метода и действуют точно так же, как и любая другая переменная в методе.
- Переменные, объявленные в области видимости, не доступны из кода за пределами этой области.
- Области видимости могут быть вложенными.
- Переменные создаются при входе в их область действия и уничтожаются при выходе из нее.
- Во внутреннем блоке кода нельзя объявлять переменные с тем же именем, что и во внешней области действия.

```java
public class VarDemo {
    String str1 = "Hello!";
    String str2 = "Hi!";

    public static void main(String[] args) {
        int x; // переменная x доступна всему коду из метода main ( )
        x = 10;
        if (x == 10) { // начало новой области действия,
            int y = 20;
            //int x = 45; // ОШИБКА! Во внутреннем блоке кода нельзя
            // объявлять переменные с тем же именем, что и во внешней области действия.
            //обе переменные x и y доступны в этой области действия
            System.out.println(" x и y : " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // ОШИБКА! переменная y недоступна
        // в этой области действия
        // переменная x доступна и эдесь
        System.out.println(" x равно " + x);
    }
}
```

# Преобразование и приведение примитивных типов в Java

Иногда возникают ситуации, когда необходимо переменной одного типа присвоить значение переменной другого типа. Например:

```
int i = 11;
byte b = 22;
i = b;
```

# Преобразование и приведение примитивных типов в Java

В Java существует два типа преобразований - автоматическое преобразование (неявное) и приведение типов (явное преобразование).

# Преобразование и приведение примитивных типов в Java

**Автоматическое преобразование типов Java**

Рассмотрим сначала автоматическое преобразование. Если оба типа совместимы, их преобразование будет выполнено в Java автоматически. Например, значение типа byte всегда можно присвоить переменной типа int, как это показано в предыдущем примере.

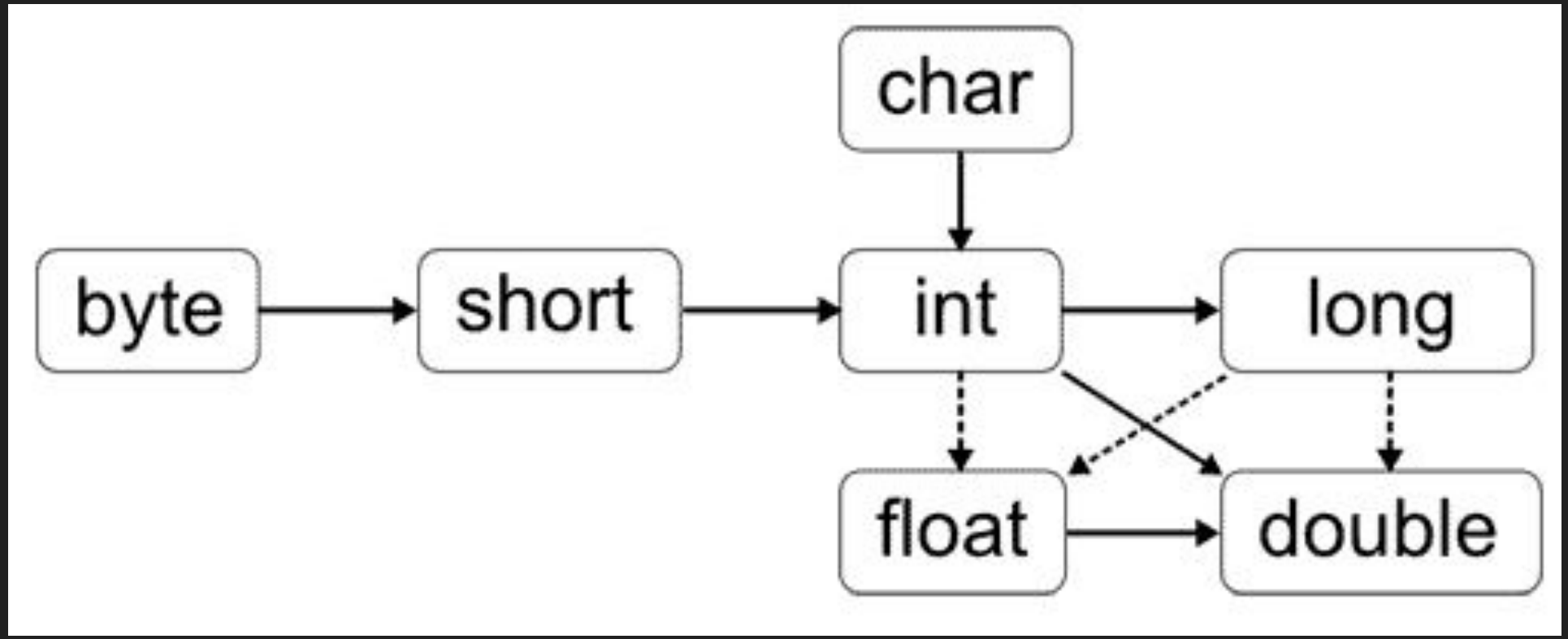
Для автоматического преобразования типа должно выполняться два условия:

- оба типа должны быть совместимы
- длина целевого типа должна быть больше длины исходного типа

В этом случае происходит преобразование с расширением.

# Преобразование и приведение примитивных типов в Java

Следующая схема показывает расширяющее преобразование в Java:



# Преобразование и приведение примитивных типов в Java

Сплошные линии обозначают преобразования, выполняемые без потери данных. Штриховые линии говорят о том, что при преобразовании может произойти потеря точности.

Например, тип данных int всегда достаточно велик, чтобы хранить все допустимые значения типа byte, поэтому никакие операторы явного приведения типов в данном случае не требуются. С точки зрения расширяющего преобразования числовые типы данных, в том числе целочисленные и с плавающей точкой, совместимы друг с другом. В то же время не существует автоматических преобразований числовых типов в тип char или boolean. Типы char и boolean также не совместимы друг с другом.

# Преобразование и приведение примитивных типов в Java

Стоит немного пояснить почему, к примеру тип byte не преобразуется автоматически (не явно) в тип char, хотя тип byte имеет ширину 8 бит, а char - 16, тоже самое касается и преобразования типа short в char. Это происходит потому, что byte и short знаковые типы данных, а char без знаковый. Поэтому в данном случае требуется использовать явное приведение типов, поскольку компилятору надо явно указать, что вы знаете чего хотите и как будет обрабатываться знаковый бит типов byte и short при преобразовании к типу char.

Поведение величины типа char в большинстве случаев совпадает с поведением величины целого типа, следовательно, значение типа char можно использовать везде, где требуются значения int или long. Однако напомним, что тип char не имеет знака, поэтому он ведет себя отлично от типа short, несмотря на то что диапазон обоих типов равен 16 бит.

# Преобразование и приведение примитивных типов в Java

**Приведение типов Java**

Несмотря на все удобство автоматического преобразования типов, оно не в состоянии удовлетворить все насущные потребности. Например, что делать, если значение типа int нужно присвоить переменной типа byte? Это преобразование не будет выполняться автоматически, поскольку длина типа byte меньше, чем у типа int. Иногда этот вид преобразования называется *сужающим преобразованием*, поскольку значение явно сужается, чтобы уместиться в целевом типе данных.

Чтобы выполнить преобразование двух несовместимых типов данных, нужно воспользоваться приведением типов. Приведение - это всего лишь явное преобразование типов. Общая форма приведения типов имеет следующий вид:

```
(целевой_тип) значение
```

где параметр целевой тип обозначает тип, в который нужно преобразовать указанное значение.

# Преобразование и приведение примитивных типов в Java

Например, в следующем фрагменте кода тип int приводится к типу byte:

```
int i = 11;
byte b = 22;
b = (byte) i;
```

Рассмотрим пример преобразования значений с плавающей точкой в целые числа. В этом примере дробная часть значения с плавающей точкой просто отбрасывается (операция усечения):

```
double d = 3.89;
int a = (int) d; //Результат будет 3
```

# Преобразование и приведение примитивных типов в Java

При приведении более емкого целого типа к менее емкому старшие биты просто отбрасываются:

```
int i = 323;
byte b = (byte) i; //Результат будет 67
```

При приведении более емкого значения с плавающей точкой в целое число происходит усечение и отбрасывание старших битов:

```
double d = 389889877779.89;
short s = (short) d; //Результат будет -1
```

# Преобразование и приведение примитивных типов в Java

**Автоматическое продвижение типов в выражениях**

Помимо операций присваивания, определенное преобразование типов может выполняться и в выражениях.

В языке Java действуют следующие правила:

- Если один операнд имеет тип double, другой тоже преобразуется к типу double.
- Иначе, если один операнд имеет тип float, другой тоже преобразуется к типу float.
- Иначе, если один операнд имеет тип long, другой тоже преобразуется к типу long.
- Иначе оба операнда преобразуются к типу int.
- В выражениях совмещенного присваивания (+=,-=,*=,/=) нет необходимости делать приведение.

# Преобразование и приведение примитивных типов в Java

Приведем пример:

При умножении переменной b1 (byte) на 2 (int) результат будет типа int. Поэтому при попытке присвоить результат в переменную b2 (byte) возникнет ошибка компиляции. Но при использовании совмещенной операции присваивания (*=), такой проблемы не возникнет:

```
byte b1 = 1;
byte b2 = 2 * b1; //Ошибка компиляции
int i1 = 2 * b1;
b2 *= 2;
```

# Преобразование и приведение примитивных типов в Java

В следующем примере тоже возникнет ошибка компиляции - несмотря на то, что складываются числа типа byte, результатом операции будет тип int, а не short.

```
public class IntegerDemo1 {
    public static void main(String[] args) {
        byte b1 = 50, b2 = -99;
        short k = b1 + b2; //ошибка компиляции
        System.out.println("k=" + k);
    }
}
```

# Преобразование и приведение примитивных типов в Java

Следующий пример аналогичен предыдущему, но используется операция совмещенного присваивание, в которой приведение происходит автоматически:

```
public class IntegerDemo2 {
    public static void main(String[] args) {
        byte b1 = 50, b2 = -99;
        b1 += b2;
        System.out.println("b1=" + b1);
    }
}
```

# Ввод с клавиатуры Java языка

Для того, чтобы пользователь мог что-то ввести с клавиатуры, существует стандартный поток ввода, представленный объектом System.in. Рассмотрим, как это происходит.

Для ввода данных с клавиатуры в Java используется метод System.in.read() - он возвращает код введенного символа. После его выполнения JVM останавливает программу и ждет пока пользователь введет символ с клавиатуры. Для того, чтобы вывести сам символ на консоль, выполняется его приведение к типу char:

```
public class SystemInDemo {
    public static void main(String[] args) throws IOException {
        int x = System.in.read();
        char c = (char) x;
        System.out.println("Код символа: " + c + " = " + x);
    }
}
```

# Ввод с клавиатуры Java языка

Конечно же, использовать System.in в чистом виде не очень удобно, если нам необходимо ввести не один символ, а целую строку. В этом случае можно воспользоваться классом Scanner. Этот класс находится в пакете java.util, поэтому его надо импортировать:

```
import java.util.Scanner;
```

# Ввод с клавиатуры Java языка

Методы этого класса позволяют считывать строку, значение типа int или double.

Методы класса Scanner:

- hasNextInt() - возвращает true если с потока ввода можно считать целое число.
- nextInt() - считывает целое число с потока ввода.
- hasNextDouble() - проверяет, можно ли считать с потока ввода вещественное число типа double.
- nextDouble() - считывает вещественное число с потока ввода.
- nextLine() - позволяет считывать целую последовательность символов, то есть строку.
- hasNext() - проверяет остались ли в потоке ввода какие-то символы.

# Ввод с клавиатуры Java языка

В следующем примере метод hasNextInt() проверяет, ввел ли пользователь целое число. И с помощью метода nextInt() считываем введенное число. Если пользователь ввел строку, то программа выведет на консоль "Вы ввели не целое число":

```
import java.util.Scanner;

public class ScannerDemo1 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Введите целое число: ");
        if (scanner.hasNextInt()) {
            int i = scanner.nextInt();
            System.out.println(i);
        } else {
            System.out.println("Вы ввели не целое число");
        }
    }
}
```

# Ввод с клавиатуры Java языка

Рассмотрим пример, в котором используется метод nextDouble() для считывания дробного числа. Если же пользователь введет строку, то программа завершится с ошибкой времени выполнения. Чтобы этого не происходило, перед вызовом метода nextDouble(), сделайте проверку с помощью метода hasNextDouble():

```
import java.util.Scanner;

public class ScannerDemo2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // если ввести букву s,
        // то случится ошибка во время исполнения
        double i = scanner.nextDouble();
        System.out.println(i);
    }
}
```

# Ввод с клавиатуры Java языка

Следующий пример использует метод nextLine() для считывания всей строки:

```
import java.util.Scanner;

public class ScannerDemo3 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String s1, s2;
        s1 = scanner.nextLine();
        s2 = scanner.nextLine();
        System.out.println(s1 + s2);
    }
}
```