1. Обощенные и параметризованные типы. Создание параметризованных классов.

```
class Account<T>{

  private T id;
  private int sum;

  Account(T id, int sum){
    this.id = id;
    this.sum = sum;
  }

  public T getId() { return id; }
  public int getSum() { return sum; }
  public void setSum(int sum) { this.sum = sum; }
}
```

2. Работа с параметризованными методами. Ограничение типа сверху или снизу (PECS?).

```
public class Funs {
    public static void customer(ArrayList<? super Integer> list){
        list.add(2);
    }
    public static Float producer(ArrayList<? extends Float> list){
        return list.get(0);
    }
}
```

3. Класс Number. Классы-оболочки. Автоупаковка и автораспаковка

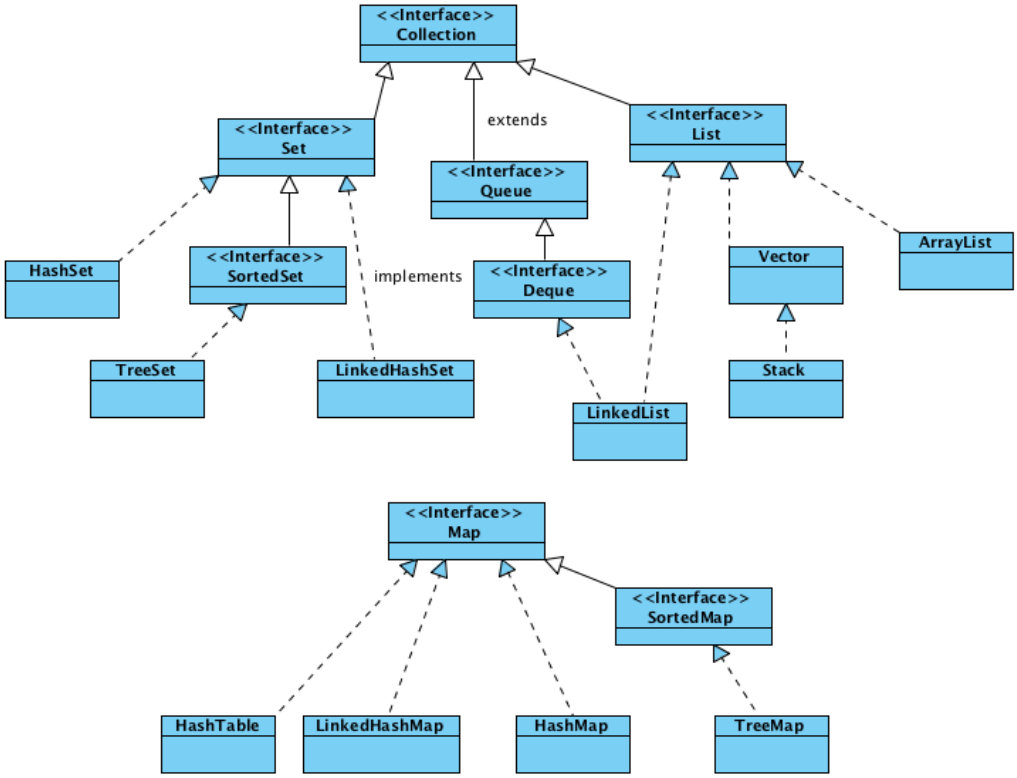
Автоупаковка

```
Integer integer = 9;
```

Распаковка

```
int in = 0;
in = new Integer(9);
```

4. Коллекции. Виды коллекций. Интерфейсы Set, List, Queue и их особенности.



5. Обход элементов коллекции. Интерфейсы Iterable, Iterator и ListIterator

ITERABLE Implementing this interface allows an object to be the target of the "for-each loop" statement.

Реализация интерфейса (iterator) предполагает, что с помощью вызова метода `next()` можно получить следующий элемент. С помощью метода `hasNext()` можно узнать, есть ли следующий элемент, и не достигнут ли конец коллекции. И если элементы еще имеются, то `hasNext()` вернет значение `true`. Метод `hasNext()` следует вызывать перед методом `next()`, так как при достижении конца коллекции метод `next()` выбрасывает исключение `NoSuchElementException`. И метод `remove()` удаляет текущий элемент, который был получен последним вызовом `next()`.

Интерфейс Iterator предоставляет ограниченный функционал. Гораздо больший набор методов предоставляет другой итератор - интерфейс **ListIterator**. Данный итератор используется классами, реализующими интерфейс `List`, то есть классами LinkedList, ArrayList и др.

Интерфейс ListIterator расширяет интерфейс Iterator и определяет ряд дополнительных методов:

- **void add(E obj)**: вставляет объект obj перед элементом, который должен быть возвращен следующим вызовом `next()`
- **boolean hasNext()**: возвращает `true`, если в коллекции имеется следующий элемент, иначе возвращает false
- **boolean hasPrevious()**: возвращает `true`, если в коллекции имеется предыдущий элемент, иначе возвращает false
- **E next()**: возвращает текущий элемент и переходит к следующему, если такого нет, то генерируется исключение `NoSuchElementException`

- **E previous()**: возвращает текущий элемент и переходит к предыдущему, если такого нет, то генерируется исключение `NoSuchElementException`
- **int nextIndex()**: возвращает индекс следующего элемента. Если такого нет, то возвращается размер списка
- **int previousIndex()**: возвращает индекс предыдущего элемента. Если такого нет, то возвращается число -1
- **void remove()**: удаляет текущий элемент из списка. Таким образом, этот метод должен быть вызван после методов `next()` или `previous()`, иначе будет сгенерировано исключение `IlligalStateException`
- **void set(E obj)**: присваивает текущему элементу, выбранному вызовом методов `next()` или `previous()`, ссылку на объект obj

6. Сортировка элементов коллекций. Интерфейсы Comparable и Comparator.
   Comparable – compateTo(element) определяем в классе и сравниваем элемент класса с другим
   Comparator – compare(element1, element2) сравниваем два элемента, реализуем ваще по где

7. Интерфейсы Set и SortedSet, их реализации. Классы HashSet и TreeSet.

Интерфейс **Set** расширяет интерфейс Collection и представляет набор уникальных элементов. Set не добавляет новых методов, только вносит изменения унаследованные.

Класс **HashSet** реализует интерфейс Set, основан на хэш-таблице, а также поддерживается с помощью экземпляра HashMap. В HashSet элементы не упорядочены, нет никаких гарантий, что элементы будут в том же порядке спустя какое-то время.

Обобщенный класс **TreeSet<E>** представляет структуру данных в виде дерева, в котором все объекты хранятся в отсортированном виде по возрастанию. TreeSet является наследником класса AbstractSet и реализует интерфейс NavigableSet, а следовательно, и интерфейс SortedSet.

8. Интерфейс List и его реализации. Классоы ArrayList и LinkedList.

Для создания простых списков применяется интерфейс List, который расширяет функциональность интерфейса Collection.

По умолчанию в Java есть встроенная реализация этого интерфейса - класс ArrayList. Класс ArrayList представляет обобщенную коллекцию, которая наследует свою функциональность от класса AbstractList и применяет интерфейс List. Проще говоря, ArrayList представляет простой список, аналогичный массиву, за тем исключением, что количество элементов в нем не фиксировано.

Обобщенный класс LinkedList<E> представляет структуру данных в виде связанного списка. Он наследуется от класса AbstractSequentialList и

реализует интерфейсы List, Dequeue и Queue. То есть он соединяет функциональность работы со списком и фукциональность очереди.

В LinkedList элементы фактически представляют собой звенья одной цепи. У каждого элемента помимо тех данных, которые он хранит, имеется ссылка на предыдущий и следующий элемент. По этим ссылкам можно переходить от одного элемента к другому.

9. Интерфейсы Map и SortedMap, их реализации. Классы HashMap и TreeMap.

Интерфейс **Map<K, V>** представляет отображение или иначе говоря словарь, где каждый элемент представляет пару "ключ-значение". При этом все ключи уникальные в рамках объекта Map. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта.

Следует отметить, что в отличие от других интерфейсов, которые представляют коллекции, интерфейс Map НЕ расширяет интерфейс Collection.

Интерфейс SortedMap расширяет Map и создает отображение, в котором все элементы отсортированы в порядке возрастания их ключей.

Базовым классом для всех отображений является абстрактный класс AbstractMap, который реализует большую часть методов интерфейса Map. Наиболее распространенным классом отображений является HashMap, который реализует интерфейс Map и наследуется от класса AbstractMap.

Класс TreeMap<K, V> представляет отображение в виде дерева. Он наследуется от класса AbstractMap и реализует интерфейс NavigableMap, а следовательно, также и интерфейс SortedMap. Поэтому в отличие от коллекции HashMap в TreeMap все объекты автоматически сортируются по возрастанию их ключей.

10. Интерфейсы Queue и Deque. Классы PriorityQueue и ArrayDeque.

В Java очереди представлены рядом классов. Одни из низ класс ArrayDeque<E>. Этот класс представляют обобщенную двунаправленную очередь, наследуя функционал от класса AbstractCollection и применяя интерфейс Deque.

11. Классы Collections и Arrays, методы для работы с коллекциями и массивами.

12.Регулярные выражения, Классы Pattern и Matcher.

Pattern - A compiled representation of a regular expression.

A regular expression, specified as a string, must first be compiled into an instance of this class. The resulting pattern can then be used to create a `Matcher` object that can match arbitrary character sequences against the regular expression. All of the state involved in performing a match resides in the matcher, so many matchers can share the same pattern.

Matcher — класс, который представляет строку, реализует механизм согласования (*matching*) с РВ и хранит результаты этого согласования (используя реализацию методов интерфейса **MatchResult**). Не имеет публичных конструкторов, поэтому для создания объекта этого класса нужно использовать метод **matcher** класса **Pattern**:

Но результатов у нас еще нет. Чтобы их получить нужно воспользоваться методом **find**. Можно использовать **matches** — этот метод вернет true только тогда, когда вся строка соответствует заданному РВ, в отличии от **find**, который пытается найти подстроку, которая удовлетворяет РВ. Для более детальной информации о результатах согласования можно использовать реализацию методов интерфейса **MatchResult**, например:

13. Байтовые потоки ввода-вывода. Классы InputStream, OutputStream и их потомки.

Класс	*Назначение*
BufferedInputStream	Буферизированный входной поток.
BufferedOutputStream	Буферизированный выходной поток.
ByteArrayInputStream	Входной поток, читающий из массива байт.
ByteArrayOutputStream	Выходной поток, записывающий в массив байт.
DataInputStream	Входной поток, включающий методы для чтения стандартных типов данных Java.
DataOutputStream	Выходной поток, включающий методы для записи стандартных типов данных Java.
FileInputStream	Входной поток, читающий из файла.
FileOutputStream	Выходной поток, записывающий в файл.
FilterInputStream	Реализация *InputStream*.
FilterOutputStream	Реализация *OutputStream*.
ObjectInputStream	Входной поток для объектов.
ObjectOutputStream	Выходной поток для объектов.
OutputStream	Абстрактный класс, описывающий поток вывода.
PipedInputStream	Входной канал (например, межпрограммный).
PipedOutputStream	Выходной канал.
PrintStream	Выходной поток, включающий *print()* и *println()*.
PushbackInputStream	Входной поток, реализующий операцию pushback (вернуть назад).

14. Символьные потоки ввода-вывода. Классы Reader, Writer и их потомки.

Класс	*Назначение*
BufferedReader	Буферизованный входной символьный поток.
BufferedWriter	Буферизованный выходной символьный поток.
CharArrayReader	Входной поток, который читает из символьного массива.
CharArrayWriter	Выходной поток, который пишет в символьный массив.
FileReader	Входной поток, читающий файл.
FileWriter	Выходной поток, пишущий в файл.
FilterReader	Фильтрующий читатель.
FilterWriter	Фильтрующий писатель.
InputStreamReader	Входной поток, транслирующий байты в символы.
LineNumberReader	Входной поток, подсчитывающий строки.
OutputStreamWriter	Выходной поток, транслирующий байты в символы.
PipedReader	Входной канал.
PipedWriter	Выходной канал.
PrintWriter	Выходной поток, включающий *print()* и *println()*.
PushbackReader	Входной поток, позволяющий возвращать символы обратно в поток.
Reader	Абстрактный класс, описывающий символьный ввод.
StringReader	Входной поток, читающий из строки.
StringWriter	Выходной поток, пишущий в строку.
Writer	Абстрактный класс, описывающий символьный вывод.

16. Работа с файлами в Java. Интерфейс Path. Классы File, Files, Paths.

```
Path testFilePath =
Paths.get("C:\\Users\\Username\\Desktop\\testFile.txt")

Path fileName = testFilePath.getFileName();
System.out.println(fileName);

Path parent = testFilePath.getParent();
System.out.println(parent);

Path root = testFilePath.getRoot();
       System.out.println(root);
```

Класс File, определенный в пакете java.io, не работает напрямую с потоками. Его задачей является управление информацией о файлах и каталогах. Хотя на уровне операционной системы файлы и каталоги отличаются, но в Java они описываются одним классом File.

Files – nio file, with getstream

22. Порядок выполнения и ограничение "happens-before". Модификатор volatile.

В код был добавлен механизм синхронизации локальной памяти нитей, названный «happens before» (дословно «случилось перед»). Был придуман ряд правил/условий, при наступлении которых память синхронизируется – обновляется до актуального состояния.

- В рамках одной нити любая команда **happens-before** (читается «случается перед») любой операцией, следующей за ней в исходном коде.
- Освобождение лока (unlock) **happens-before** захватом того же лока (lock).
- Выход из **synchronized** блока/метода **happens-before** вход в synchronized блок/метод на том же мониторе.
- Запись volatile поля **happens-before** чтение того же самого volatile поля.
- Завершение метода run экземпляра класса Thread **happens-before** выход из метода join() или возвращение false методом isAlive() экземпляром той же нити.
- Вызов метода start() экземпляра класса Thread **happens-before** начало метода run() экземпляра той же нити.
- Завершение конструктора **happens-before** начало метода finalize() этого класса
- Вызов метода interrupt() на нити **happens-before**, когда нить обнаружила, что данный метод был вызван, либо путем выбрасывания исключения InterruptedException, либо с помощью методов isInterrupted() или interrupted()

24. Интерфейсы Lock, ReadWriteLock, Condition.

Интерфейс java.util.concurrent.locks.ReadWriteLock позволяет читать несколько потоков одновременно, но одновременно может писать только один поток.

- **Блокировка чтения** – если ни один поток не заблокировал ReadWriteLock для записи, то несколько потоков могут получить доступ к блокировке чтения.

- **Блокировка записи** – если ни один поток не читает или не пишет, то один поток может получить доступ к блокировке записи.

Блокировка чтения – если ни один поток не заблокировал ReadWriteLock для записи, то несколько потоков могут получить доступ к блокировке чтения.

Блокировка записи – если ни один поток не читает или не пишет, то один поток может получить доступ к блокировке записи.

25. Атомарный доступ к переменным. Пакет java.util.concurrent.atomic.

Concurrenct.atomic -- A small toolkit of classes that support lock-free thread-safe programming on single variables. In essence, the classes in this package extend the notion of `volatile` values, fields, and array elements to those that also provide an atomic conditional update operation of the form:

```
boolean compareAndSet(expectedValue, updateValue);
```

This method (which varies in argument types across different classes) atomically sets a variable to the `updateValue` if it currently holds the `expectedValue`, reporting `true` on success. The classes in this package also contain methods to get and unconditionally set values, as well as a weaker conditional atomic update operation `weakCompareAndSet` described below.

30. Провайдеры служб.

В этом случае может помочь шаблон ServiceLocator. Он подразумевает, что есть некий интерфейс, представляющий сервис — в нашем случае это Knowledge. У него есть несколько реализаций, нам нужно получать объект сервиса универсальным способом. Для этого служит класс ServiceLocator, который пользуется классом InitialContext для поиска конкретных классов, реализующих сервис. ServiceLocator хранит у себя ссылки на найденные объекты и предоставляет их при вызове метода getService().

В Java данную схему реализует класс ServiceLoader. Класс имеет метод load(), который позволяет загрузить конкретную реализацию сервиса, и метод iterator(), возвращающий итератор по загруженным службам. Можно

пройтись по службам в цикле, и выбрать нужную, которая подходит в данный момент.

31. JDBC

JDBC (англ. Java DataBase Connectivity — *соединение с базами данных на Java*) — платформенно независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета `java.sql`, входящего в состав Java SE.

JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает.

JDBC API содержит два основных типа интерфейсов: первый – для разработчиков приложений и второй (более низкого уровня) – для разработчиков драйверов.

Соединение с базой данных описывается классом, реализующим интерфейс `java.sql.Connection`. Имея соединение с базой данных, можно создавать объекты типа Statement, служащие для исполнения запросов к базе данных на языке SQL.

Существуют следующие виды типов Statement, различающихся по назначению:

- `java.sql.Statement` — Statement общего назначения;
- `java.sql.PreparedStatement` — Statement, служащий для выполнения запросов, содержащих подставляемые параметры (обозначаются символом '?' в теле запроса);
- `java.sql.CallableStatement` — Statement, предназначенный для вызова хранимых процедур.

Интерфейс `java.sql.ResultSet` позволяет легко обрабатывать результаты запроса.

Храни́мая проце́дура (англ. *Stored procedure*) — объект базы данных, представляющий собой набор SQL-инструкций, который компилируется один раз и хранится на сервере.

37. Форматирование локализованных числовых данных, текста, даты и времени.

```
public abstract class Format
extends Object
implements Serializable, Cloneable
```

`Format` is an abstract base class for formatting locale-sensitive information such as dates, messages, and numbers.

`Format` defines the programming interface for formatting locale-sensitive objects into `String`s (the `format` method) and for parsing `String`s back into objects (the `parseObject` method).

Generally, a format's `parseObject` method must be able to parse any string formatted by its `format` method. However, there may be exceptional cases where this is not possible. For example, a `format` method might create two adjacent integer numbers with no separator in between, and in this case the `parseObject` could not tell which digits belong to which number.

Для форматирования ввода и вывода даты в ***Java*** до введения нового ***Date Time API*** (https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html), который был введен в *Java 8*, использовался класс ***SimpleDateFormat***.

SimpleDateFormat является подклассом ***DateFormat***, который позволяет форматировать ввод-вывод даты и времени в рамках предопределенных стилей. В отличие от ***DateFormat***, ***SimpleDateFormat*** позволяет создавать собственные настраиваемые форматы ввода-вывода.

Для создания экземпляра класса ***SimpleDateFormat*** используется один из 4 конструкторов:

- `SimpleDateFormat()`
- `SimpleDateFormat(String pattern)`
- `SimpleDateFormat(String pattern, DateFormatSymbols formatSymbols)`
- `SimpleDateFormat(String pattern, Locale locale)`

pattern – шаблон определяющий формат даты и времени
formatSymbols – символы формата даты (например название месяцев или дней недели)
locale — локаль

SimpleDateFormat чувствителен к локали. При создании экземпляра ***SimpleDateFormat*** без параметра ***Locale***, вывод будет форматироваться в соответствии с ***Locale*** по умолчанию.

38. Пакет java.time. Классы для представления даты и времени

Есть два базовых способа представления времени. Один способ представляет время в терминах человека, таких как год, месяц, день, час, минуты и секунды. Второй способ представляет машинное время, измеряя время непрерывно с начала, называемого эпохой, в наносекундах. Пакет Date-Time содержит большое количество классов, представляющий дату и время. Некоторые классы в Date-Time API представляют машинное время, некоторые человеческое.

Package java.time

The main API for dates, times, instants, and durations.

See: Description

Class Summary

Class	Description
Clock	A clock providing access to the current instant, date and time using a time-zone.
Duration	A time-based amount of time, such as '34.5 seconds'.
Instant	An instantaneous point on the time-line.
LocalDate	A date without a time-zone in the ISO-8601 calendar system, such as 2007-12-03.
LocalDateTime	A date-time without a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30.
LocalTime	A time without a time-zone in the ISO-8601 calendar system, such as 10:15:30.
MonthDay	A month-day in the ISO-8601 calendar system, such as --12-03.
OffsetDateTime	A date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00.
OffsetTime	A time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 10:15:30+01:00.
Period	A date-based amount of time in the ISO-8601 calendar system, such as '2 years, 3 months and 4 days'.
Year	A year in the ISO-8601 calendar system, such as 2007.
YearMonth	A year-month in the ISO-8601 calendar system, such as 2007-12.
ZonedDateTime	A date-time with a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00 Europe/Paris.
ZoneId	A time-zone ID, such as Europe/Paris.
ZoneOffset	A time-zone offset from Greenwich/UTC, such as +02:00.

39. Функциональные интерфейсы и λ-выражения. Пакет java.util.function

Что же такое функциональный интерфейс?

Если интерфейс в Java содержит один и только один абстрактный метод, то он называется функциональным. Этот единственный метод определяет назначение интерфейса.

Например, интерфейс Runnable из пакета java.lang является функциональным, потому, что он содержит только один метод run().

```
@FunctionalInterface
interface MyInterface{

    // абстрактный метод
    double getPiValue();
}
public class Main {

    public static void main( String[] args ) {

    // объявление ссылки на MyInterface
    MyInterface ref;

    // лямбда-выражение
    ref = () -> 3.1415;

    System.out.println("Value of Pi = " + ref.getPiValue());
    }
```

Лямбда-выражения, по сути, это анонимный класс или метод. Лямбда-выражение не выполняется само по себе. Вместо этого, оно используется для реализации метода, определенного в функциональном интерфейсе.

41. Конвейерная обработка данных. Пакет java.util.stream.

При работе со Stream API важно понимать, что все операции с потоками бывают либо **терминальными (terminal)**, либо **промежуточными (intermediate)**. Промежуточные операции возвращают трансформированный поток. Например, выше в примере метод `filter` принимал поток чисел и возвращал уже преобразованный поток, в котором только числа больше 0. К возвращенному потоку также можно применить ряд промежуточных операций.

Конечные или терминальные операции возвращают конкретный результат. Например, в примере выше метод `count()` представляет терминальную операцию и возвращает число. После этого никаких промежуточных операций естественно применять нельзя.

Все потоки производят вычисления, в том числе в промежуточных операциях, только тогда, когда к ним применяется терминальная операция. То есть в данном случае применяется отложенное выполнение.

```
1import java.util.stream.*;
2//.....................
3long count = IntStream.of(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5).filter(w -> w > 0).count();
4System.out.println(count);
```

В основе Stream API лежит интерфейс **BaseStream**. Его полное определение:

```
1interface BaseStream<T , S extends BaseStream<T , S>>
```

Здесь параметр `T` означает тип данных в потоке, а `S` - тип потока, который наследуется от интерфейса BaseStream.

FORK / JOIN

Фреймворк fork/join был представлен в Java 7. Он предоставляет инструменты, помогающие ускорить параллельную обработку, пытаясь использовать все доступные процессорные ядра, что достигается **с помощью подхода “разделяй и властвуй”** .

На практике это означает, что **фреймворк сначала “разветвляется”** , рекурсивно разбивая задачу на более мелкие независимые подзадачи, пока они не станут достаточно простыми для асинхронного выполнения.

После этого **начинается часть “соединение”** , в которой результаты всех подзадач рекурсивно объединяются в один результат, или в случае задачи, которая возвращает void, программа просто ждет, пока не будет выполнена каждая подзадача.

Для обеспечения эффективного параллельного выполнения платформа fork/join использует пул потоков, называемый *ForkJoinPool* , который управляет рабочими потоками типа *ForkJoinWorkerThread*

Проще говоря, свободные потоки пытаются “украсть” работу у деков занятых потоков.

По умолчанию рабочий поток получает задачи из головы своего собственного deque. Когда он пуст, поток берет задачу из хвоста deque другого занятого потока или из глобальной очереди ввода, так как именно здесь, вероятно, будут расположены самые большие части работы.

Такой подход сводит к минимуму вероятность того, что потоки будут конкурировать за задачи. Это также сокращает количество раз, когда потоку придется искать работу, так как сначала он работает с самыми большими доступными фрагментами работы.

ГОНКА ПОТОКОВ

Состояние гонки возникает, когда один и тот же ресурс используется несколькими потоками одновременно, и в зависимости от порядка действий каждого потока может быть несколько возможных результатов