| **Класс** | **Можно вызвать flush()?** | **Тип потока** | **Описание** |
| --- | --- | --- | --- |
| **ByteArrayOutputStream** | Да | OutputStream | Поток вывода в память (байтовый массив). flush() не обязателен, но реализован. |
| **CharArrayWriter** | Да | Writer | Поток записи в память (символьный массив). flush() реализован, но фактически не нужен. |
| **BufferedWriter** | Да | Writer | Буферизированный символьный поток. flush() сбрасывает буфер в целевой поток. |
| **BufferedOutputStream** | Да | OutputStream | Буферизированный байтовый поток. flush() сбрасывает буфер в целевой поток. |
| **FileOutputStream** | Да | OutputStream | Поток записи в файл. flush() сбрасывает буфер (если есть) в файл. |
| **PrintWriter** | Да | Writer | Поток форматированного вывода. flush() сбрасывает буфер. |
| **FileWriter** | Да | Writer | Поток записи в файл. flush() сбрасывает буфер. |
| **OutputStreamWriter** | Да | Writer | Преобразует байты в символы. flush() сбрасывает буфер. |
| **BufferedReader** | Нет | Reader | Буферизированный поток ввода. flush() отсутствует, так как это поток чтения. |
| **CharArrayReader** | Нет | Reader | Поток чтения из символьного массива. flush() отсутствует. |
| **ByteArrayInputStream** | Нет | InputStream | Поток чтения из байтового массива. flush() отсутствует. |
| **FileInputStream** | Нет | InputStream | Поток чтения из файла. flush() отсутствует. |
| **InputStreamReader** | Нет | Reader | Преобразует байты в символы для чтения. flush() отсутствует. |

# 2 Вопрос – Коллекции

## 1. **List** — упорядоченные коллекции с доступом по индексу

* Позволяют хранить элементы в порядке добавления.
* Допускают дубликаты.
* Основные реализации: **ArrayList**, **LinkedList**, **Vector**, **Stack**.

| **Коллекция** | **Основные методы** | **Плюсы** | **Минусы** | **Где применяют** |
| --- | --- | --- | --- | --- |
| **ArrayList** | add(E), get(int), remove(int), size() | Быстрый доступ по индексу (O(1)) | Медленные вставки/удаления в середине (O(n)) | Когда нужен быстрый доступ к элементам по индексу[1](https://habr.com/ru/articles/237043/)[4](https://sky.pro/wiki/java/kollekcii-i-spiski-v-java/)[5](https://devmark.ru/article/java-collections) |
| **LinkedList** | add(E), get(int), remove(int), size() | Быстрые вставка и удаление в любом месте (O(1)) | Медленный доступ по индексу (O(n)) | Когда часто добавляют/удаляют элементы в середине[1](https://habr.com/ru/articles/237043/)[4](https://sky.pro/wiki/java/kollekcii-i-spiski-v-java/) |
| **Vector** | add(E), get(int), remove(int), size() | Потокобезопасность (синхронизация) | Сниженная производительность из-за синхронизации | В многопоточных приложениях, где нужна синхронизация[1](https://habr.com/ru/articles/237043/) |
| **Stack** | push(E), pop(), peek() | Реализация стека LIFO | Устаревший, рекомендуется использовать Deque | Для реализации стека (LIFO)[1](https://habr.com/ru/articles/237043/) |

## 2. **Set** — коллекции без дубликатов

* Не допускают повторяющихся элементов.
* Порядок элементов зависит от реализации.
* Основные реализации: **HashSet**, **LinkedHashSet**, **TreeSet**.

| **Коллекция** | **Особенности** | **Плюсы** | **Минусы** | **Где применяют** |
| --- | --- | --- | --- | --- |
| **HashSet** | Неупорядоченная коллекция | Быстрые операции добавления, удаления, поиска (O(1)) | Нет порядка элементов | Когда важна уникальность, порядок не важен |
| **LinkedHashSet** | Сохраняет порядок вставки | Быстрые операции + сохранение порядка | Затраты памяти на поддержание порядка | Когда нужна уникальность и порядок вставки |
| **TreeSet** | Отсортированная коллекция (по возрастанию) | Автоматическая сортировка элементов | Более медленные операции (O(log n)) | Когда важен отсортированный набор уникальных элементов[6](https://university.ylab.io/articles/tpost/2shbb9tgz1-kollektsii-java) |

## 3. **Queue** — коллекции для работы с очередями

* Элементы добавляются в конец и извлекаются из начала (FIFO).
* Основные реализации: **LinkedList** (реализует Queue), **PriorityQueue**, **ArrayDeque**.

| **Коллекция** | **Особенности** | **Плюсы** | **Минусы** | **Где применяют** |
| --- | --- | --- | --- | --- |
| **LinkedList** | Реализует интерфейс Queue | Универсальность, поддержка FIFO и Deque | Медленный доступ по индексу | Очереди, двунаправленные очереди |
| **PriorityQueue** | Элементы с приоритетом | Быстрая сортировка по приоритету | Нет поддержки null, неупорядоченная по вставке | Очереди с приоритетами |
| **ArrayDeque** | Быстрая реализация Deque | Высокая производительность | Нет поддержки ограничения размера | Стек, очередь, двунаправленная очередь |

## 4. **Map** — коллекции для хранения пар «ключ-значение»

* Ключи уникальны, значения могут повторяться.
* Основные реализации: **HashMap**, **LinkedHashMap**, **TreeMap**, **Hashtable**.

| **Коллекция** | **Особенности** | **Плюсы** | **Минусы** | **Где применяют** |
| --- | --- | --- | --- | --- |
| **HashMap** | Неупорядоченная коллекция | Быстрый доступ по ключу (O(1)) | Нет порядка ключей | Когда важна скорость доступа по ключу |
| **LinkedHashMap** | Сохраняет порядок вставки | Быстрый доступ + сохранение порядка | Затраты памяти на порядок | Кэширование, когда важен порядок вставки |
| **TreeMap** | Отсортированная по ключу коллекция | Автоматическая сортировка ключей | Медленнее HashMap (O(log n)) | Когда нужен отсортированный по ключу Map[6](https://university.ylab.io/articles/tpost/2shbb9tgz1-kollektsii-java) |
| **Hashtable** | Потокобезопасная, устаревшая реализация | Синхронизация | Устаревший, медленнее HashMap | В старом коде, где нужна потокобезопасность |

## Основные методы коллекций (интерфейс Collection)

* add(E e) — добавить элемент.
* remove(Object o) — удалить элемент.
* contains(Object o) — проверить наличие элемента.
* size() — размер коллекции.
* isEmpty() — проверка на пустоту.
* clear() — очистить коллекцию.
* Итерация — через Iterator или foreach.

## Плюсы и минусы коллекций

| **Тип коллекции** | **Плюсы** | **Минусы** |
| --- | --- | --- |
| List | Упорядоченность, доступ по индексу | Медленные вставки/удаления в середине (ArrayList), медленный доступ по индексу (LinkedList) |
| Set | Уникальность элементов | Нет доступа по индексу, порядок зависит от реализации |
| Queue | Удобство работы с очередями | Нет доступа по индексу |
| Map | Быстрый доступ по ключу | Ключи должны быть уникальны |

## Где применяются коллекции

* **ArrayList** — когда важен быстрый доступ к элементам по индексу (например, списки, таблицы).
* **LinkedList** — когда часто добавляют/удаляют элементы в середине (например, очереди, стеки).
* **HashSet** — когда нужна уникальность и высокая производительность (например, фильтрация дубликатов).
* **TreeSet** — когда нужна сортировка уникальных элементов (например, сортированные множества).
* **HashMap** — для хранения пар ключ-значение с быстрым доступом (например, кэш, словари).
* **LinkedHashMap** — когда важен порядок вставки (например, кэш с удалением по времени).
* **PriorityQueue** — для очередей с приоритетом (например, планировщики задач).

# 4 вопрос

## Сравнение с интерфейсами

В таблице на изображении есть описание стандартных функциональных интерфейсов Java:

* **Consumer<T>**: (T) → void
* **Predicate<T>**: (T) → boolean
* **Supplier<T>**: () → T
* **Function<T,R>**: (T) → R
* **UnaryOperator<T>**: (T) → T
* **ToIntFunction<T>**: (T) → int

**Наш случай:**

* Аргумент: String
* Результат: int

Это полностью соответствует интерфейсу **ToIntFunction<T>**.

java

@FunctionalInterface

**public** **interface** ToIntFunction<T> {

**int** applyAsInt(T value);

}

## Ответ

**Правильный ответ:**  
**ToIntFunction<T>**

## Почему не другие варианты?

* **Function<T,R>** — возвращает любой тип, но не обязательно int.
* **Consumer<T>** — ничего не возвращает.
* **Predicate<T>** — возвращает boolean.
* **Supplier<T>** — не принимает аргументов.
* **UnaryOperator<T>** — возвращает тот же тип, что и аргумент.

**Вывод:**  
Для лямбда-выражения (String s) -> s.hashCode() правильный интерфейс — **ToIntFunction<T>**.

Related

How can I optimize the loading of large images in Java

What are the best methods for resizing images in Java

How can I add watermarks to images using Java

What are the most efficient algorithms for image compression in Java

How can I convert a byte array back to an image in Java