Лекция 12.
Работа со строками. Работа с датой. Лямбда-функции.
Устройство класса String
1. Методы класса String
У класса String очень много методов. Несколько самых основных из них:

Методы	Описание
int length()	Возвращает количество символов в строке
boolean isEmpty()	Проверяет, что строка == пустая строка
String[] split(String regex)	Разделяет строку на несколько подстрок.

Вспомогательные классы для работы со строками в Java

1. Сравнение строк

Сравнение — одна из самых частых вещей, которая делается со строками. У класса String более десяти различных методов, которые используются для сравнения строк со строками. Ниже мы рассмотрим семь основных.

Методы	Описание
boolean equals(String str)	Строки считаются равными, если все их символы совпадают.
boolean equalsIgnoreCase(String str)	Сравнивает строки, игнорируя регистр (размер) букв

int compareTo(String str)	Сравнивает строки лексикографически. Возвращает 0, если строки равны. Число меньше нуля, если текущая строка меньше строки-параметра. Число больше нуля, если текущая строка больше строки-параметра
int compareToIgnoreCase(String str)	Сравнивает строки лексикографически, игнорирует регистр. Возвращает 0, если строки равны. Число меньше нуля, если текущая строка меньше строки-параметра. Число больше нуля, если текущая строка больше строки-параметра
boolean startsWith(String prefix)	Проверяет, что текущая строка начинается со строки prefix
boolean endsWith(String suffix)	Проверят, что текущая строка заканчивается на строку suffix

2. Поиск подстрок

Вторая по популярности операция после сравнения строк — это поиск одной строки в другой. Для этого у класса String тоже есть немного методов:

Методы	Описание
int indexOf(String str)	Ищет строку str в текущей строке. Возвращает индекс первого символа встретившийся строки.
int indexOf(String str, **int** index)	Ищет строку str в текущей строке, пропустив index первых символов. Возвращает индекс найденного вхождения.
int lastIndexOf(String str)	Ищет строку str в текущей строке с конца. Возвращает индекс первого вхождения.
int lastIndexOf(String str, **int** index)	Ищет строку str в текущей строке с конца, пропустив index первых символов.

3. Создание подстрок

Кроме сравнения строк и поиска подстрок, есть еще одно очень популярное действие — получение подстроки из строки.

Методы	Описание
String substring(**int** beginIndex, **int** endIndex)	Возвращает подстроку, заданную интервалом символов beginIndex..endIndex.
String repeat(**int** count)	Повторяет текущую строку n раз
String replaceAll(String regex, String replacement)	Заменяет в текущей строке все подстроки, совпадающие с регулярным выражением.
String toLowerCase()	Преобразует строку к нижнему регистру
String toUpperCase()	Преобразует строку к верхнему регистру
String trim()	Удаляет все пробелы в начале и конце строки

Вот краткое описание существующих методов:

Метод substring(int beginIndex, int endIndex)

Метод substring возвращает новую строку, которая состоит из символов текущей строки, начиная с символа под номером beginIndex и заканчивая endIndex. Как и во всех интервалах в Java, символ с номером endIndex в интервал не входит.

Если параметр endIndex не указывается (а так можно), подстрока берется от символа beginIndex и до конца строки.

Метод repeat(int n)

Метод repeat просто повторяет текущую строку n раз.

Методы replaceFirst() и replaceAll()

Метод replaceAll() заменяет все вхождения одной подстроки на другую. Метод replaceFirst() заменяет первое вхождение переданной подстроки на заданную подстроку. Строка, которую заменяют, задается регулярным выражением.

Методы toLowerCase() и toUpperCase()

Методы приводят всю строку к нижнему и верхнему регистру.

Метод trim()

Метод trim() удаляет у строки пробелы с начала и с конца строки. Пробелы внутри строки никто не трогает.

StringBuilder

1. Изменения строк

Строки в Java — это неизменяемые объекты (immutable). Так было сделано для того, чтобы класс-строку можно было сильно оптимизировать и использовать повсеместно. Однако часто возникают ситуации, когда программисту все же было бы удобнее иметь String-класс, который можно менять. Который не создает новую подстроку при каждом вызове его метода.

Например, у нас есть очень большая строка и мы часто дописываем что-то в ее конец. В этом случае даже коллекция символов (ArrayList<Character>) может быть эффективнее, чем постоянное пересоздание строк и конкатенации объектов типа String.

Именно поэтому в язык Java все же добавили тип String, который можно менять. Называется он StringBuilder.

Создание объекта

Чтобы создать объект StringBuilder на основе существующей строки, нужно выполнить команду вида:

```
StringBuilder имя = new StringBuilder(строка);
```

Чтобы создать пустую изменяемую строку, нужно воспользоваться командой вида:

```
StringBuilder имя = new StringBuilder();
```

Класс StringBuilder имеет два десятка полезных методов, вот самые важные из них:

Метод	Описание
StringBuilder append(obj)	Преобразовывает переданный объект в строку и добавляет к текущей строке
StringBuilder replace(**int** start, **int** end, String str)	Заменяет часть строки, заданную интервалом start..end на переданную строку
StringBuilder delete(**int** start, **int** end)	Удаляет из строки символы, заданные интервалом
int indexOf(String str, **int** index)	Ищет подстроку в текущей строке
int lastIndexOf(String str, **int** index)	Ищет подстроку в текущей строке с конца
char charAt(**int** index)	Возвращает символ строки по его индексу

String substring(**int** start, **int** end)	Возвращает подстроку, заданную интервалом
StringBuilder reverse()	Разворачивает строку задом наперед.
int length()	Возвращает длину строки в символах

2. Краткое описание методов

Добавление к строке

Чтобы что-то добавить к изменяемой строке (StringBuilder), нужно воспользоваться методом append(). Пример:

Код	Описание
StringBuilder builder = **new** StringBuilder("Привет"); builder.append("Пока"); builder.append(123);	Привет ПриветПока ПриветПока123

Преобразование к стандартной строке

Чтобы преобразовать объект StringBuilder к строке типа String, нужно просто вызвать у него метод toString(). Пример

Код	Вывод на экран
StringBuilder builder = **new** StringBuilder("Привет"); builder.append(123); String result = builder.toString(); System.out.println(result);	Привет123

Как заменить часть строки на другую?

Для этого есть метод replace(int begin, int end, String str). Пример:

Код	Вывод на экран
StringBuilder builder = **new** StringBuilder("Привет"); builder.replace(2, 5, "Hello!"); String result = builder.toString(); System.out.println(result);	ПрHello!т

3. Полезные примеры работы со строками

Как развернуть строку задом наперед?

Для этой операции есть специальный метод — reverse(); Пример:

Код	Вывод на экран
String str = "Привет"; StringBuilder builder = **new** StringBuilder(str); builder.reverse(); String result = builder.toString(); System.out.println(result);	тевирП

Класс StringBuffer

Есть еще один класс — StringBuffer — это аналог класса StringBuilder, только его методы имеют модификатор synchronized. А это значит, что к объекту StringBuffer можно одновременно обращаться из нескольких потоков.
Зато он работает гораздо медленнее, чем StringBuilder.

Класс Calendar

1. Переход от класса Date к классу Calendar

Полное имя класса Calendar — java.util.Calendar. Не забывайте добавлять его в import, если решите использовать в своем коде.
Создать объект Calendar можно командой:

```
Calendar date = Calendar.getInstance();
```

Статический метод getInstance() класса Calendar создаст объект Calendar, инициализированный текущей датой. В зависимости от настроек компьютера, на котором выполняется программа, будет создан нужный календарь.
Более правильно было бы сказать — актуальный. Дело в том, что на Земле не один, а много календарей. И почти каждый из них связан с какой-нибудь религией или страной. Класс Calendar поддерживает 3 из них:

Календарь	Описание
GregorianCalendar	Христианский Григорианский календарь
BuddhistCalendar	Буддистский календарь
JapaneseImperialCalendar	Японский Императорский календарь

А ведь есть еще Китайский и Арабский календари.

В Китае на момент написания этой лекции официально 4718 год. А по мусульманскому календарю — 1441.

2. Создание объекта календарь

Мы будем использовать Григорианский календарь как самый распространенный в мире.

Объект календарь с произвольной датой создается командой:

```
Calendar date = new GregorianCalendar(год, месяц, день);
```

Да, каждый раз придется писать GregorianCalendar. Можно и вместо Calendar писать GregorianCalendar: так тоже будет работать. Но просто Calendar писать короче.

Год нужно писать полностью: никаких 19 вместо 2019. Месяцы по-прежнему нумеруются с нуля. А дни месяца по-прежнему нумеруются не с нуля.

Чтобы задать не только дату, но и время, нужно передать их дополнительными параметрами:

```
... = new GregorianCalendar(год, месяц, день, часы, минуты, секунды);
```

Можно даже передать миллисекунды, если это необходимо: их указывают следующим параметром после секунд.

3. Вывод объекта календарь на экран

Если просто вывести объект-календарь на экран, результат вас не сильно порадует.

Код

```
Calendar calendar = new GregorianCalendar(2019, 03, 12);
System.out.println(calendar);
```

Вывод на экран

```
java.util.GregorianCalendar[time=?,areFieldsSet=false,areAllFieldsSet=false,lenient=true,zone=sun.util.calendar.ZoneInfo[id="Europe/Helsinki",offset=7200000,dstSavings=3600000,useDaylight=true,transitions=118,lastRule=java.util.SimpleTimeZone[id=Europe/Helsinki,offset=7200000,dstSavings=3600000,useDaylight=true,startYear=0,startMode=2,startMonth=2,startDay=-1,startDayOfWeek=1,startTime=3600000,startTimeMode=2,endMode=2,endMonth=9,endDay=-1,endDayOfWeek=1,endTime=3600000,endTimeMode=2]],firstDayOfWeek=1,minimalDaysInFirstWeek=1,ERA=?,YEAR=2019,MONTH=3,WEEK_OF_YEAR=?,WEEK_OF_MONTH=?,DAY_OF_MONTH=12,DAY_OF_YEAR=?,DAY_OF_WEEK=?,DAY_OF_WEEK_IN_MONTH=?,AM_PM=0,HOUR=0,HOUR_OF_DAY=0,MINUTE=0,SECOND=0,MILLISECOND=?,ZONE_OFFSET=?,DST_OFFSET=?]
```

Все дело в том, что календарь — это именно календарь, а не дата: у него много всяких настроек, и они все выводятся на экран.
Правильно будет отображать объект календарь с помощью класса SimpleDateFormat.

4. Работа с фрагментами даты

Чтобы получить фрагмент даты (год, месяц, ...), у класса Calendar есть специальный метод — get(). Метод один, зато с параметрами:

```
int month = calendar.get(Calendar.MONTH);
```

Где calendar — это переменная типа Calendar, а MONTH — это переменная-константа класса Calendar.
В метод get в качестве параметра вы передаете специальную константу класса Calendar, и в результате получаете нужное значение.

Код	Описание
`Calendar calendar = Calendar.getInstance();`	
`int era = calendar.get(Calendar.ERA);`	эра (до нашей эры или после)
`int year = calendar.get(Calendar.YEAR);`	год
`int month = calendar.get(Calendar.MONTH);`	месяц
`int day = calendar.get(Calendar.DAY_OF_MONTH);`	день месяца
`int dayOfWeek = calendar.get(Calendar.DAY_OF_WEEK);`	день недели
`int hour = calendar.get(Calendar.HOUR);`	часы
`int minute = calendar.get(Calendar.MINUTE);`	минуты
`int second = calendar.get(Calendar.SECOND);`	секунды

Для изменения фрагмента даты используется метод set:

```
calendar.set(Calendar.MONTH, значение);
```

Где calendar — это переменная типа Calendar, а MONTH — это переменная-константа класса Calendar.

В метод set в качестве первого параметра вы передаете специальную константу класса Calendar, а в качестве второго параметра — новое значение.

Код	Описание

Calendar calendar = **new** GregorianCalendar();
 	
calendar.set(Calendar.YEAR, 2021);
calendar.set(Calendar.MONTH, 6);
calendar.set(Calendar.DAY_OF_MONTH, 4);
calendar.set(Calendar.HOUR_OF_DAY, 12);
calendar.set(Calendar.MINUTE, 15);
calendar.set(Calendar.SECOND, 0);
 	
System.out.println(calendar.getTime());
	
	год = 2021
	месяц = Июль (нумерация с 0)
	4 число
	часы
	минуты
	секунды

5. Константы класса Calendar

В классе Calendar есть константы не только для названия фрагментов даты.

Calendar date = **new** GregorianCalendar(2021, Calendar.JANUARY, 31);

Например, есть константы для обозначения месяцев:

Или, например, для дней недели:

```
Calendar calendar = new GregorianCalendar(2021, Calendar.JANUARY, 31);
if (calendar.get(Calendar.DAY_OF_WEEK) == Calendar.FRIDAY)
{
  System.out.println("Это пятница");
}
```

Использование констант позволяет сделать код более читабельным, поэтому их и добавили.

6. Изменение даты в объекте Calendar

У класса Calendar есть метод, который позволяет проводить с датой более умные операции. Например, добавить к дате год, месяц или несколько дней. Или отнять. Называется этот метод add(). Выглядит работа с ним примерно так:

```
calendar.add(Calendar.MONTH, значение);
```

Где calendar — это переменная типа Calendar, а MONTH — это переменная-константа класса Calendar.

В метод add в качестве первого параметра вы передаете специальную константу класса Calendar, и в качестве второго параметра — значение, которое нужно добавить.

Особенность этого метода в том, что он умный. Давайте сами посмотрим, насколько:

Код

```
Calendar calendar = new GregorianCalendar(2021, Calendar.FEBRUARY, 27);
calendar.add(Calendar.DAY_OF_MONTH, 2);
System.out.println(calendar.getTime());
```

Вывод на экран

```
Mon Mar 01 00:00:00 EET 2021
```

Этот метод понимает, что в феврале 2021 года всего 28 дней, и итоговая дата — 1 марта.
Чтобы выполнить действие, уменьшающее дату, нужно в метод add() передать значение с отрицательным знаком.
Этот метод учитывает длины месяцев и високосные годы.

Предыстория появления Лямбда-выражений

1. Сортировка

Чтобы сортировать коллекцию строк в алфавитном порядке, в Java есть отличный метод — Collections.sort(коллекция);

Этот статический метод выполняет сортировку переданной коллекции, и в процессе сортировки попарно сравнивает ее элементы: чтобы понять, менять элементы местами или нет.
Сравнение элементов в процессе сортировки выполняется с помощью метода compareTo(), который есть у всех стандартных классов: Integer, String, ...
Метод compareTo() класса Integer сравнивает значения двух чисел, а метод compareTo() класса String смотрит на алфавитный порядок строк.
Таким образом, коллекция чисел будет отсортирована в порядке их возрастания, а коллекция строк — в алфавитном порядке.

Альтернативная сортировка

А если мы хотим сортировать строки не по алфавиту, а по их длине? И числа хотим сортировать в порядке убывания. Как быть в этой ситуации?
Для этого у класса Collections есть еще один метод sort(), но уже с двумя параметрами:

```
Collections.sort(коллекция, компаратор);
```

Где компаратор — это специальный объект, который знает, как сравнивать объекты в коллекции в процессе сортировки. Компаратор происходит от английского слова Comparator (сравнитель), а Comparator — от слова Compare — сравнивать.

Интерфейс Comparator

На самом деле все очень просто. Тип второго параметра метода sort() — Comparator<T>

Где T — это тип-параметр, такой же, как и тип элементов коллекции, а Comparator — это интерфейс, который имеет единственный метод int compare(T obj1, T obj2);
Другими словами, объект-компаратор — это любой объект класса, который реализует интерфейс Comparator. Выглядит интерфейс Comparator очень просто:

```
public interface Comparator<Тип>
{
  public int compare(Тип obj1, Тип obj2);
}
```

Код интерфейса Comparator

Метод compare() сравнивает два параметра, которые в него передают.
Если метод возвращает отрицательное число, то obj1 < obj2. Если метод возвращает положительно число, то obj1 > obj2. Если метод возвращает 0, то считается, что obj1 == obj2.
Вот как будет выглядеть объект компаратор, который сравнивает строки по их длине:

```
public class StringLengthComparator implements Comparator<String>
{
  public int compare (String obj1, String obj2)
  {
    return obj1.length() – obj2.length();
  }
}
```

Код класса StringLengthComparator

Для того, чтобы сравнить длины строк, достаточно просто вычесть одну длину из другой. Полный код программы, которая сортирует строки по длине, будет выглядеть вот так:

```
public class Solution
{
  public static void main(String[] args)
  {
    ArrayList<String> list = new ArrayList<String>();
    Collections.addAll(list, "Привет", "как", "дела?");
    Collections.sort(list, new StringLengthComparator());
  }
}

class StringLengthComparator implements Comparator<String>
{
  public int compare (String obj1, String obj2)
  {
    return obj1.length() – obj2.length();
  }
}
```

Сортировка строк по длине

А как вы думаете, можно ли записать данный код короче? По сути, тут только одна строка, которая несет полезную информацию — obj1.length() – obj2.length();.
Но ведь код не может существовать вне метода, поэтому пришлось добавить метод compare(), а для метода пришлось добавить новый класс – StringLengthComparator. И еще типы переменных нужно указывать... В общем, все правильно.
Однако, есть способы записать этот код короче.

Анонимный внутренний класс

Вы можете записать код компаратора прямо внутри метода main(), а компилятор сам сделает все остальное. Пример:

```
public class Solution
{
   public static void main(String[] args)
   {
      ArrayList<String> list = new ArrayList<String>();
      Collections.addAll(list, "Привет", "как", "дела?");

      Comparator<String> comparator = new Comparator<String>()
      {
         public int compare (String obj1, String obj2)
         {
            return obj1.length() – obj2.length();
```

```
        }
    };

    Collections.sort(list, comparator);
  }
}
```

Сортировка строк по длине

Вы можете создать объект наследник интерфейса Comparator, не создавая сам класс! Компилятор создаст его автоматически и даст ему какое-нибудь временное имя. Сравните:

```
Comparator<String> comparator = new Comparator<String>()
{
  public int compare (String obj1, String obj2)
  {
    return obj1.length() – obj2.length();
  }
};
```

Анонимный внутренний класс

```
Comparator<String> comparator = new StringLengthComparator();

class StringLengthComparator implements Comparator<String>
```

```
{
   public int compare (String obj1, String obj2)
   {
      return obj1.length() – obj2.length();
   }
}
```

Класс StringLengthComparator

Одинаковым цветом раскрашены одинаковые блоки кода в двух разных случаях. Отличия совсем небольшие на самом деле.

Когда компилятор встретит в коде первый блок кода, он просто сгенерирует для него второй блок кода и даст классу какое-нибудь случайное имя.

2. Лямбда-выражения в Java

Допустим, вы решили использовать в вашем коде анонимный внутренний класс. В этом случае у вас будет блок кода типа такого:

```
Comparator<String> comparator = new Comparator<String>()
{
   public int compare (String obj1, String obj2)
   {
      return obj1.length() – obj2.length();
   }
```

```
};
```
Анонимный внутренний класс

Тут и объявление переменной, и создание анонимного класса — все вместе. Однако есть способ записать этот код короче. Например, так:

```
Comparator<String> comparator = (String obj1, String obj2) ->
{
   return obj1.length() – obj2.length();
};
```

Точка с запятой нужна, т.к. у вас тут не только скрытое объявление класса, но и создание переменной.

Такая запись называется **лямбда-выражением.**

Если компилятор встретит такую запись в вашем коде, он просто сгенерирует по ней полную версию кода (с анонимным внутренним классом).

Обратите внимание: при записи лямбда-выражения мы опустили не только имя класса Comparator<String>, но и имя метода int compare().

У компилятора **не возникнет проблем с определением метода**, т.к. лямбда-выражение можно писать **только для интерфейсов, у которых метод один.** Впрочем есть способ обойти это правило, но сейчас не об этом.

Давайте еще раз посмотрим на полную версию кода, только раскрасим серым цветом ту ее часть, которую можно опустить при записи лямбда выражения:

```
Comparator<String> comparator = new Comparator<String>()
{
```

```
    public int compare (String obj1, String obj2)
   {
     return obj1.length() – obj2.length();
   }
};
```

Анонимный внутренний класс

Вроде бы ничего важного не упустили. Действительно, если у интерфейса Comparator есть только один метод compare(), по оставшемуся коду компилятор вполне может восстановить серый код.

Сортировка

Кстати, код вызова сортировки теперь можно записать так:

```
Comparator<String> comparator = (String obj1, String obj2) ->
{
  return obj1.length() – obj2.length();
};
Collections.sort(list, comparator);
```

Или даже так:

```
Collections.sort(list, (String obj1, String obj2) ->
   {
     return obj1.length() – obj2.length();
   }
);
```

Мы просто подставили вместо переменной comparator сразу то значение, которое присваивали переменной comparator.

Выведение типов

Но и это еще не все. Код в этих примерах можно записать еще короче. Во-первых, компилятор может сам определить, что у переменных obj1 и obj2 тип String. А во-вторых, фигурные скобки и оператор return тоже можно не писать, если у вас в коде метода всего одна команда.

Сокращенный вариант будет таким:

```
Comparator<String> comparator = (obj1, obj2) ->
  obj1.length() – obj2.length();

Collections.sort(list, comparator);
```

А если вместо переменной comparator сразу подставить ее значение, то получим такой вариант:

```
Collections.sort(list, (obj1, obj2) ->  obj1.length() — obj2.length() );
```

Всего одна строка кода, никакой лишней информации — только переменные и код.

3. Как это работает

Лямбда-выражение можно записать там, где используется **тип-интерфейс с одним-единственным методом.**

Например, в этом коде Collections.sort(list, (obj1, obj2) -> obj1.length() — obj2.length()); можно записать лямбда-выражение, т.к. сигнатура метода sort() имеет вид:

sort(Collection<T> colls, Comparator<T> comp)

Когда мы передали в метод sort в качестве первого параметра коллекцию ArrayList<String>, компилятор смог определить тип второго параметра как Comparator<String>. А из этого сделал вывод, что этот интерфейс имеет единственный метод int compare(String obj1, String obj2). Остальное уже дело техники.

Функциональный метод

1. Функциональные методы

Если **у интерфейса есть только один метод**, переменной этого типа-интерфейса можно присвоить значение, заданное лямбда-выражением (лямбда-функцией). Такие интерфейсы стали называть функциональными интерфейсами (после добавления в Java поддержки лямбда-функций).

Например, в Java есть интерфейс Consumer<Тип> (Consumer == Потребитель), который содержит метод accept(Тип obj). Зачем же нужен этот интерфейс?

В Java 8 у коллекций появился метод forEach(), который позволяет для каждого элемента коллекции выполнить какое-нибудь действие. И вот для передачи действия в метод forEach() как раз и используется функциональный интерфейс Consumer<T>.

Вот как можно вывести все элементы коллекции на экран:

```
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, "Привет", "как", "дела?");

list.forEach( (s) -> System.out.println(s) );
```
Вывод всех элементов коллекции (с использованием лямбда-выражения)

Компилятор преобразует этот код в код:

```
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, "Привет", "как", "дела?");

list.forEach(new Consumer<String>()
{
  public void accept(String s)
  {
    System.out.println(s));
  }
});
```
Вывод всех элементов коллекции (запись с использованием анонимного класса)

Первая запись однозначно короче, чем вторая. И хотя читать код с лямбда-выражениями непросто, читать код с анонимными внутренними классами порой еще сложнее.

2. Ссылка на метод

Однако наш код с лямбда-выражением можно записать еще короче.

Во-первых, можно опустить скобки вокруг параметра s:

```
list.forEach( (s) -> System.out.println(s) );
```

Было

```
list.forEach( s -> System.out.println(s) );
```

Стало

Так можно делать только если параметр один. Если параметров несколько, нужно использовать скобки.

Ну а во-вторых, можно записать так:

```
list.forEach( System.out::println );
```

Самая компактная запись

Это все одна и та же запись. Обратите внимание, что после println нет скобок.

Тут записан один и тот же код — вызов метода:

```
объект::метод
```

```
x -> объект.метод(x)
```

Подумайте сами: мы хотели для каждого элемента коллекции list выполнять какое-то действие. Если это действие — вызов одной функции (такой как println()), было бы разумно просто передать функцию в метод в качестве параметра.

А как объяснить компилятору, что функцию нужно именно передать, а не вызвать? Для этого перед именем метода ставим не точку, а два двоеточия: одно двоеточие уже занято в тернарном операторе.
Это и есть самая простая и компактная запись.

3. Конструктор

Ссылки на методы с помощью двойного двоеточия очень удобно использовать, когда мы будем работать с потоками ввода-вывода.
3 популярных способах передачи ссылки на метод:

Ссылка на метод объекта

Чтобы передать ссылку на метод объекта, нужно записать код вида объект::метод.
Этот код эквивалентен коду x -> объект.метод(x).
В качестве объекта могут фигурировать такие специальные переменные как this и super.

Ссылка на метод класса

Чтобы передать ссылку на статический метод, нужно записать код вида класс::метод.
Этот код будет преобразован к коду вида x -> класс.метод(x);

Ссылка на конструктор

Конструктор по своему поведению чем-то похож на статический метод класса, поэтому на него тоже можно передать ссылку. Выглядит это так: класс::new.
Например, можно обойти стирание типов у коллекций и передать в метод toArray() ссылку на конструктор, который создаст нужный массив: toArray(int[]::new);