Занятие №2. Типы данных и отладка

На этом занятии мы, скорее всего,… потеряем время.

Дело в том, что речь пойдет об очень важных вещах, но их важность осознается не сразу.

Первая тема – типы данных, обычно редко вызывает проблемы при решении задач, но иногда именно знание того, как они устроены в Java дает возможность найти неочевидную ошибку. Вторая – отладка – обычно вызывает интерес при поиске конкретной ошибки, а они обычно у всех разные… Но не говорить об этом безусловно нельзя, и, будем надеяться, когда-нибудь знания и советы, которые мы здесь дадим, пригодятся.

Начнем с задачи, которая наглядно демонстрирует, что такое «олимпиадная задача».

Задача «Сумма от 1 до N»

Условие задачи

Дано целое число N. Посчитайте сумму целых чисел от 1 до N.

Гарантируется, что ответ на любом тесте помещается в тип long.

Ограничения по времени и памяти

16 мегабайт,

1 секунда на каждом тесте.

Формат входных данных

Единственное целое число N.

Формат выходных данных

Единственное целое число – ответ на задачу.

Примеры

Ввод	**Вывод**
1	1
3	6

При всей простоте условия, эта задача содержит много подводных камней.

Попробуйте решить ее самостоятельно, без подсказок. И только потом читайте дальше. Интересно, сколько возможных ошибок вам удастся избежать?

Попробуем описать процесс решения задачи.

«Да это же простым циклом делается!»

```
int N = in.nextInt();
int s = 0;
for (int i = 1; i <= N; i++)
{
     s += i;
}
out.println(s);
```

Результат: Test 3: Wrong answer

Читаем условие еще раз внимательно.

«А! Целое, это может быть и отрицательные!»

Учитываем отрицательные. Пишем правильно, не дублируя цикл.

```
int N = in.nextInt();
int s = 0;
int K = N;
if (N < 0) K = -N;
for (int i = 1; i <= K; i++)
{
     s += i;
}
if (N < 0) s = -s + 1;
out.println(s);
```

Для N равного 0 ответ тоже получается верный.

Результат: Test 4: Time Limit

Читаем условие еще раз внимательно.

«Тип long, это же больше 10^{18}, это же очень долго. Надо не циклом, а по формуле!»

Переписываем цикл на известную формулу $S = \frac{N(N+1)}{2}$. Избавляемся от ставшей ненужной переменной K.

```
int N = in.nextInt();
int s = 0;
s = N * (N + 1) / 2;
if (N < 0) s = -s + 1;
```

```
out.println(s);
```

Результат: Test 5: Wrong Answer

Еще раз вспоминаем про long.

«Надо не int, а long!»

Переписываем, заменяя тип int на long.

```
long N = in.nextLong();
long s = 0;
s = N * (N + 1) / 2;
if (N < 0) s = -s + 1;
out.println(s);
```

Результат: Test 6: Wrong Answer

Чтение условие по пятому разу не помогает. Вроде все учтено.

Пробуем потестировать. На маленьких числах, в том числе и отрицательных, работает правильно. На больших работает правильно… На очень больших… Пробуем ввести максимум - 2^{32}-1 - 4294967295. Программа выдает…отрицательное число -2147483648!

Почему? Это очень сложный момент, догадаться очень трудно. Дело в том, что мы сначала умножаем, а потом делим на два. Ответ действительно помещается в long, но на промежуточном этапе происходит переполнение и дальше уже неверные вычисления!

«Сначала делить на 2, потом умножать!»

Ставим скобки

```
long N = in.nextLong();
long s = 0;
s = N * ((N + 1) / 2);
if (N < 0) s = -s + 1;
out.println(s);
```

Результат: Test 8: Wrong Answer

Ну теперь-то почему?

Потому, что деление целых чисел в Java происходит нацело. И если делить нечетное, результат будет неверный. Например, 5 / 2 это ровно 2!

«Делить надо то, что делится!»

Если N четное – то его, если нет – N-1.

```
long N = in.nextLong();
long s = 0;
if (N % 2 == 0) s = (N / 2) * (N + 1));
else s = N * ((N + 1) / 2);
if (N < 0) s = -s + 1;
out.println(s);
```

Результат: OK.

Ура, но с какого раза?!

Из решения этой задачи можно извлечь несколько уроков. Надо внимательно читать условие, не делать «в лоб», а пытаться избавится от лишних циклов, если это возможно, и, наконец, нам здесь пригодилось знание о типах – нужно знать, что такое переполнение, когда оно возникает, как осуществляется деление целых чисел.

Типы данных в Java

На этом занятии мы подробно поговорим только о типах, которые нам будут нужны в дальнейшем.

Прежде всего в Java есть примитивные типы и объекты.

Примитивные типы

Переменные примитивных типов – могут содержать числа, символы и логические значения.

На этом занятии мы подробно поговорим только о типах, которые нам будут нужны в дальнейшем.

О символах мы подробно поговорим на Занятии №8, а о числовых типах нужно, прежде всего, знать диапазон значений, которые они могут принимать. А для дробных (вещественных) точность.

Целые

Тип	**Диапазон**
int	от -2147483648 до 2147483647
long	от -9223372036854775808 до 9223372036854775807

Вещественные

double	$1,7*10^{-308}<\|x\|<1,7*10^{308}$	Точность

		17 цифр

Хотя примитивные **типы инициализируются в Java нулями по умолчанию**, лучше, чтобы не путаться, делать инициализацию явно:

```
int x; // допустимо, в x будет 0
int y = 0; // так лучше, нагляднее
```

Вещественные числа записываются с точкой, например

```
double z1 = -16.2305;
```

С числами можно проводить арифметические операции

сложение + (плюс);

вычитание - (дефис);

умножение * (звездочка);

деление / (наклонная черта — слэш);

взятие остатка от деления (деление по модулю) % (процент);

Существуют операции присваивания, соответствующие этим операциям. Нагляднее писать

```
x += y; // добавить y к x
```

чем

```
x = x + y // сложить x с y, результ ат присвоить x
```

Операции увеличения и уменьшения переменной на единицу удобнее производить при помощи

инкремента (++) и декремента (--):

```
x = x + 1;// возможно, но некрасиво
x += 1; // так лучше!
x++; // а это лучше всего!
```

Отметим, что в языке Java взятие остатка от деления %, инкремент ++ и декремент -- применяются и к вещественным типам.

При арифметических действиях действует приведение типов. Если один из операторов double, второй тоже приводится к типу double. Если long – к long.

Деление целых чисел производится нацело.

```
out.print(7 / 2);
```

выведет 3,

тогда как

```
out.print(7.0 / 2);
```

выведет 3.5

Того же эффекта можно достичь, если привести одно из этих чисел к double явно:

```
out.print((double)7/2);
```

Этот прием нужно использовать, если нужно поделить целочисленную переменную «как в математике»

```
int x = 2;
out.print(7/(double)x);
```

В обоих случаях результат будет 3.5

Переменные логического типа (`bool`) могут принимать только два значения true и false

Пример:

```
bool isPrime = true;
```

Отметим, что потом в коде не следует писать

```
if (isPrime == true)…
```

нагляднее писать просто

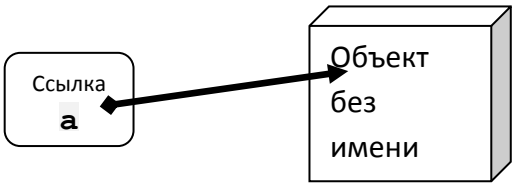
```
if (isPrime)…
```

Кроме примитивных типов при программировании на Java в используются переменные-представители различных классов. Для нас будут важны прежде всего строки и массивы. Подробно мы поговорим о них на Занятиях 6 и 8. Так что, если какие-то моменты сейчас будут непонятны, это нормально. Необходимо понять идею «в целом».

Объекты

Напомним, что с любыми объектами в Java программисты работают при помощи ссылок.

Сами объекты на самом деле не имеют имени, а программист получает доступ к ним при помощи переменных, занимающим малый объем памяти и фактически хранящим только адрес объекта в памяти:



Это означает несколько важных практических вещей.

Чаще всего сравнение объектов при помощи операции равенства (==) является ошибкой!

```
String s1 = "abc";
String s2 = "abcd";
s2 = s2.substring(0, 3); // создается новая строка!
if (s1 == s2) out.println(s1 + " equals " + s2);
else out.println(s1 + " does NOT equal " + s2);
```

выведет

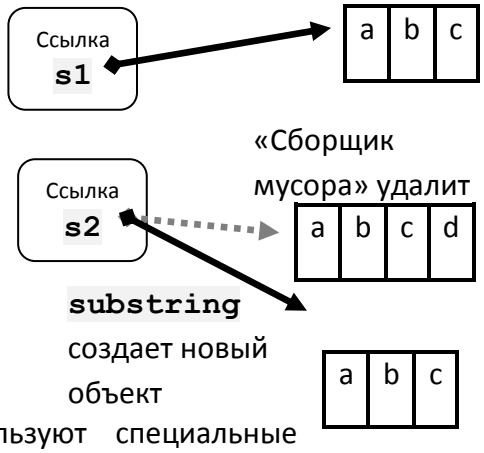
```
abc does NOT equal abc
```

Поясним решение компьютера на картинке

Видно, что s1 и s2 указывают на разные объекты,

то есть имеют разные значения,поэтому а

не равно b!



Для корректного сравнения объектов используют специальные функции. (Обычно для этого перегружают функцию equals класса Object)

Присваивание объектов тоже нужно делать осторожно!

Например, код

```
int[] a = new int[]{1, 2, 3};
int[] b = a; // теперь a и b указывают на ОДИН массив
b[1] = 7;
out.println(a[1]);
```

Выведет 7.

«Вместе» с массивом b меняется массив a!

Нарисуйте соответствующую картинку Сверьтесь с Подсказкой 2.1

Но иногда такое «странное» поведение. Бывает полезно.

Например, можно передавать массив в функцию и там его изменять. Именно переданный массив, а не его копию.

Копировать массивы можно поэлементно, используя цикл. Но удобнее использовать для это статическую функцию arraycopy класса System. Подробно она обсуждается на Занятии «Массивы».

Классы-обертки

Для того, чтобы преодолеть разрыв между примитивными типами и объектами, чтобы получить возможность работать с числами и символами как с объектами, разработчики добавили в язык так называемые классы-обертки. Например, класс `Integer` содержит число типа `int` и при этом позволяет с этим числом как с объектом, например, помещать в коллекции объектов.

Для нас важны статические методы классов-оберток, позволяющие производить манипуляции с переменными соответствующих типов, передаваемыми как параметры. На занятии «Символы и строки» мы подробно рассмотрим возможности класса `Character`, обертки для примитивного типа `char`, позволяющего сделать работу с символами очень удобной.

BigInteger и BigDecimal

Особого внимания заслуживает класс `BigInteger`, объекты которого представляют собой «длинные» целые (дробные длинные числа представляются классом `BigDecimal`). В Java встроена длинная арифметика, работа с числами практически не ограниченного размера.

Необходимо сказать о двух неочевидных технических моментах.

Во-первых, в Java не разрешена перегрузка операций. Для того, чтобы, скажем, сложить два объекта BigInteger, нужно пользоваться не операцией «+», а функцией `add`.

Во-вторых, в результате работы функций реализующих арифметические операции исходное число не меняется, а создается и возвращается новое число – результат операции. Например, код

```
BigInteger b = new BigInteger("1");
b.add(b); // b плюс b
out.println(b);
```

выведет 1.

Правильно записать вторую строку так:

```
b = b.add(b);
```

В этом случае **b** примет новое значение, раезультат операции, то есть 2.

Возможности классов **BigInteger** и **BigDecimal** можно изучить по документации к языку.

Заметим, что возможностями этих классов нужно пользоваться с большой осторожностью. С длинными числами компьютер работает значительно медленнее, чем с обычными примитивными типами. Во многих случаях желание использовать длинную арифметику говорит о неэффективном алгоритме. Часто задачи можно решить без этого.

Отладка

Если при обсуждении типов данных, возникает много технических подробностей, которые пригодятся далеко не сразу, то разговор об отладке еще сложнее.

> Компьютер - устройство разработанное для ускорения и автоматизации человеческих ошибок.
>
> Если отладка - процесс удаления ошибок, то программирование должно быть процессом их внесения.
> *Э.Дейкстра*

Во-первых, у каждого свои ошибки. И поэтому рассказ о том, как искать в чужой программе слушается обычно без энтузиазма.

Во-вторых, многие ошибки «ловятся» внимательным чтением условия, пристальным вглядыванием в код и отладочной печатью. По коду ставятся многочисленные out.println и смотрится где программа начинает вычислять «не то».

В-третьих, некоторые с успехом используют интеллектуальный дебаггер – преподавателя. «А где у меня оши-и-и-бка!» или «Покажи-и-и-те третий тест!». На некоторых преподавателей такое действует…

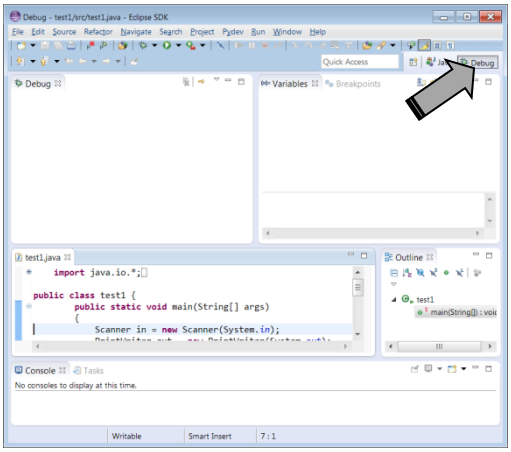
И все же расскажем о мощном средстве среды программирования Eclipse дебаггере или, по-русски, отладчике.

В отличие от отладочной печати, программа под отладчиком не меняется – не надо производить никаких действий после отладки, удалять, комментировать строки. Во-

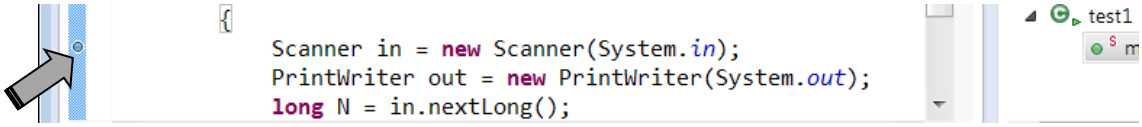
вторых, отладка происходит действительно по шагам. Сразу видно, в какой именно строке программы происходят неполадки.

Выполним в отладчике программу, которую обсуждали в начале занятия - сумму от 1 до N.

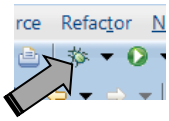
Переходим в окно отладчика, нажимая на кнопку Debug в правом верхнем углу Eclipse. Обратно, к привычной раскладке окон можно вернуться, нажав кнопку Java.



Ставим точку останова на первую строку нашей программы двойным щелчком мыши по голубой полосе слева



И начинаем отладку. Для этого нажимаем не привычную кнопку Play, а ту, которая рядом с ней, с жучком. Или нажимаем на клавиатуре **F11**.

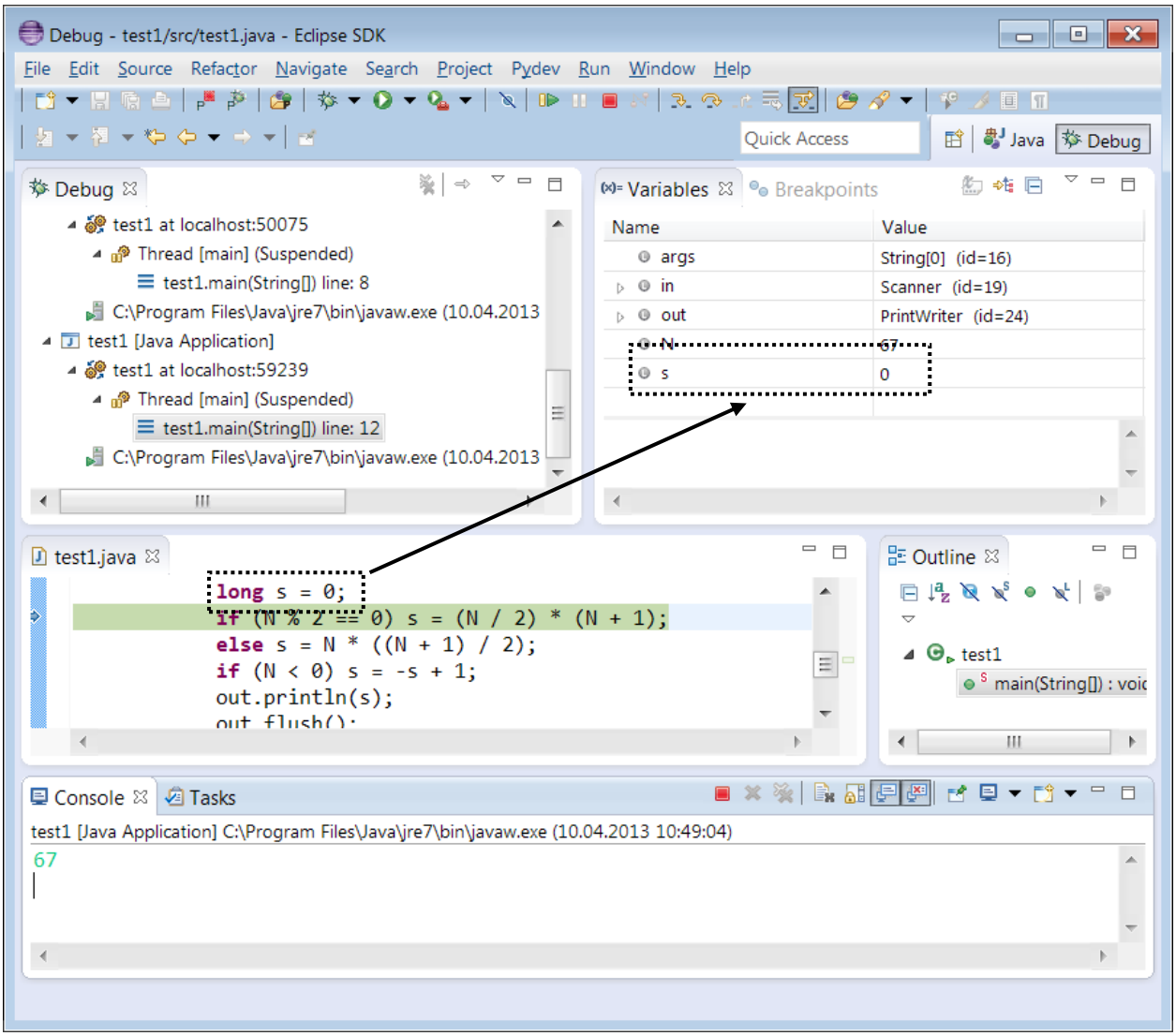


Перед нами окно кода, чтобы следить, какая строка исполняется в данный момент, окно с вкладками **Variables** и **Breakpoints**, в которых показываются состояние переменных программы и информация о точках останова, а внизу привычное окно консоли. Окно **Debug** позволяет контролировать запущенные процессы – Java многопоточна, несколько программ могут работать одновременно. Окно **Outline** позволяет быстро переключаться между функциями программы.

Рассмотрим основные кнопки для управления процессом отладки:

Нажимаем два раза **F6**. Видим в окне переменных, что создались две новые переменные **`in`** и **`out`**.

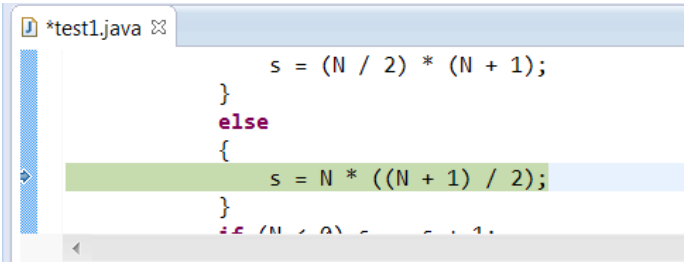
На следующем **F6** программа «зависает». Переменная N будет создана только после того, как произойдет чтения из сканера in. Вводим значение в консоли. Следующим шагом создается переменная s:



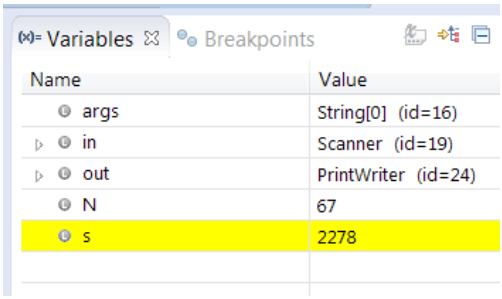
Дальше выполняется условие.

Чтобы было удобнее смотреть, какое действие выполняется по условию, следует писать каждую команду в отдельной строке. Остановим процесс отладки, вернемся к тексту программы, поправим ее, и запустим процесс заново.

Теперь наглядно видно: выполняется ветка «иначе», потому что N – нечетное.

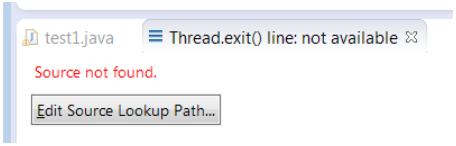


А на следующем шаге изменится переменная s. Это будет видно в окне Variables:



Еще несколько нажатий **F6** и программа завершена – на консоли ответ.

Если нажать **F6** в последней строке программы, то в окне кода может появиться сообщение

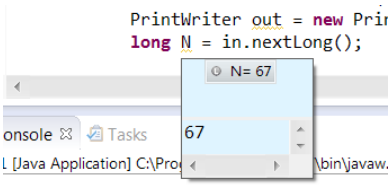


Это происходит потому, что main заканчивает работу и продолжает работу метод exit() класса Thread, исходного текста которого у нас нет. Для корректного завершения программы в этом случае следует нажать кнопку **F8** ( ) чтобы программа нормально завершилась.

Заметим, что мы всегда нажимали кнопку F6. Кнопка F5 нужна, если у нас выполняется функция и мы хотим отладить ее работу. Например, в программе написано `y = factorial(x)` и мы хотим «войти» в функцию `factorial`, проследить за ее работой. **F6** выполняет выполнение функции за один шаг.

А что, если в программе есть длинный цикл? Тогда возможно поставить после него точку останова и пропустить ее при помощи **F8** (кнопка). При нажатии программа выполнится «мгновенно», до следующей точки останова. Если точек больше не будет – закончится.

Осталось заметить, что проверить значение переменной можно не только в окне Varables, но и прямо в окне кода, подведя к ней курсор мыши:



Больше того, его можно прямо там поправить, в нижней части окна, чтобы не останавливать отладку и сразу продолжать искать следующие ошибки.

Подсказки и решения. 2 Занятие.

2.1

Обе ссылки указывают на один и тот же массив!