



Программирование на Java

Лекция 7. Преобразование типов

20 января 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) <gene@itc.mipt.ru>

Copyright © 2003 года [Центр Sun технологий МФТИ, ЦОС и ВТ МФТИ](#)[®], Все права защищены.

Аннотация

Эта лекция посвящена вопросам преобразования типов. Поскольку Java – язык строго типизированный, компилятор и виртуальная машина всегда следят за работой с типами, гарантируя надежность выполнения программы. Однако во многих случаях то или иное преобразование необходимо осуществить для реализации логики программы. С другой стороны, некоторые безопасные переходы между типами Java позволяет осуществлять неявным для разработчика образом, что может привести к неверному пониманию работы программы.

В лекции рассматриваются все виды преобразований, а затем все ситуации в программе, где они могут применяться. В заключение приводится начало классификации типов переменных и типов значений, которые они могут хранить. Этот вопрос будет детализироваться в будущих лекциях.

Оглавление

| | |
|--|----|
| Лекция 7. Преобразование типов | 1 |
| 1. Введение | 1 |
| 2. Виды приведений | 2 |
| 2.1. Тожественное преобразование | 3 |
| 2.2. Преобразование примитивных типов (расширение и сужение) | 3 |
| 2.3. Преобразование ссылочных типов (расширение и сужение) | 7 |
| 2.4. Преобразование к строке | 9 |
| 2.5. Запрещенные преобразования | 10 |
| 3. Применение приведений | 10 |
| 3.1. Присвоение значений | 11 |
| 3.2. Вызов метода | 12 |
| 3.3. Явное приведение | 14 |
| 3.4. Оператор конкатенации строк | 15 |
| 3.5. Числовое расширение | 15 |
| 3.5.1. Унарное числовое расширение | 15 |
| 3.5.2. Бинарное числовое расширение | 16 |
| 4. Тип переменной и тип ее значения | 16 |
| 5. Заключение..... | 18 |
| 6. Контрольные вопросы..... | 18 |

Лекция 7. Преобразование типов

Содержание лекции.

| | |
|--|----|
| 1. Введение | 1 |
| 2. Виды приведений | 2 |
| 2.1. Тожественное преобразование | 3 |
| 2.2. Преобразование примитивных типов (расширение и сужение) | 3 |
| 2.3. Преобразование ссылочных типов (расширение и сужение) | 7 |
| 2.4. Преобразование к строке | 9 |
| 2.5. Запрещенные преобразования | 10 |
| 3. Применение приведений | 10 |
| 3.1. Присвоение значений | 11 |
| 3.2. Вызов метода | 12 |
| 3.3. Явное приведение | 14 |
| 3.4. Оператор конкатенации строк | 15 |
| 3.5. Числовое расширение | 15 |
| 3.5.1. Унарное числовое расширение | 15 |
| 3.5.2. Бинарное числовое расширение | 16 |
| 4. Тип переменной и тип ее значения | 16 |
| 5. Заключение..... | 18 |
| 6. Контрольные вопросы..... | 18 |

1. Введение

Как уже говорилось, Java является строго типизированным языком, что означает, что каждое выражение и каждая переменная имеет строго определенный тип уже на момент компиляции. Тип устанавливается на основе структуры применяемых выражений и типов литералов, переменных и методов, используемых в этих выражениях.

Например:

```
long l=3;
l = 5+'A'+1;
print ("l="+Math.round(l/2F));
```

Опишем, как в этом примере компилятор устанавливает тип каждого выражения, и какие преобразования (conversion) типов необходимо осуществить при каждом действии:

- В первой строке литерал 3 имеет тип по умолчанию, то есть `int`. При присвоении этого значения переменной типа `long` необходимо провести преобразование.
- Во второй строке сначала производится сложение значений типа `int` и `char`. Вторым аргументом также будет преобразован, чтобы операция проводилась с точностью в 32 бита. Вторым оператором сложения опять потребуется преобразование, так как наличие переменной `l` увеличивает точность до 64 бит.
- В третьей строке сначала будет выполнена операция деления, для чего значение `long` надо будет привести к типу `float`, так как вторым операндом - дробный литерал. Результат будет передан в метод `Math.round`, который произведет математическое округление, и вернет целочисленный результат типа `int`. Это значение необходимо преобразовать в текст, чтобы осуществить дальнейшую конкатенацию строк. Как рассматривается ниже, эта операция проводится в два этапа - сначала простой тип приводится к объектному классу-обертке (в данном случае `int` к `Integer`), а затем у полученного объекта вызывается метод `toString()`, что дает преобразование к строке.

Данный пример показывает, что даже простые строки могут содержать многочисленные преобразования, зачастую незаметные для разработчика. Часто бывают и такие случаи, когда программисту необходимо явно изменить тип некоторого выражения или переменной, например, чтобы воспользоваться подходящим методом или конструктором.

Вспомним уже рассмотренный пример:

```
byte b=1;
byte c=(byte)-b;
int i=c;
```

Здесь во второй строке необходимо провести явное преобразование, чтобы присвоить значение типа `byte` переменной типа `int`. В третьей же строке обратное приведение производится автоматически, неявным для разработчика образом.

Рассмотрим сначала, какие переходы между различными типами возможно осуществить.

2. Виды приведений

В Java возможны семь видов приведений:

- тождественное (identity);
- расширение примитивного типа (widening primitive);
- сужение примитивного типа (narrowing primitive);
- расширение объектного типа (widening reference);
- сужение объектного типа (narrowing reference);
- преобразование к строке (String);
- запрещенные преобразования (forbidden).

Рассмотрим их по отдельности.

2.1. Тожественное преобразование

Самым простым является тождественное преобразование. В Java преобразование выражения любого типа к точно такому же типу всегда допустимо и успешно выполняется.

Зачем нужно тождественное приведение? Есть две причины для того, чтобы выделить такое преобразование в особый вид.

Во-первых, с теоретической точки зрения теперь можно утверждать, что любой тип в Java может участвовать в преобразовании, хотя бы в тождественном. Например, примитивный тип `boolean` нельзя привести ни к какому другому типу, кроме него самого.

Вторая причина носит более практический смысл. Иногда в Java могут встретиться такие выражения, как длинный последовательный вызов методов:

```
print(getCity().getStreet().getHouse().getFlat().getRoom());
```

При исполнении такого выражения сначала вызывается первый метод `getCity()`. Можно предположить, что возвращаемым значением будет объект класса `City`. У этого объекта далее будет вызван следующий метод `getStreet()`. Чтобы узнать, значение какого типа он вернет, необходимо посмотреть описание класса `City`. У этого значения будет вызван следующий метод (`getHouse()`), и так далее. Чтобы узнать результирующий тип всего выражения, необходимо просмотреть описание каждого метода и класса.

Компилятор легко справится с такой задачей, однако разработчику может потребоваться большое количество усилий, чтобы проследить всю цепочку. В этом случае можно воспользоваться тождественным преобразованием, сделав приведение к точно такому же типу. Это ничего не изменит в структуре программы, но значительно облегчит чтение кода:

```
print((MyFlatImpl) (getCity().getStreet().getHouse().getFlat()));
```

2.2. Преобразование примитивных типов (расширение и сужение)

Легко видеть, что следующие четыре вида приведений легко представляются в виде таблицы.

| | |
|------------------------|--------------------------|
| простой тип расширение | ссылочный тип расширение |
| простой тип сужение | ссылочный тип сужение |

Что это все означает? Начнем по порядку. Для простых типов расширение означает, что осуществляется переход от менее емкого типа к более емкому. Например, от типа `byte` (длина 1 байт) к типу `int` (длина 4 байта). Такие преобразования безопасны в том смысле, что новый тип всегда гарантированно вмещает в себя все данные, которые хранились в старом типе, и таким образом не происходит потери данных. Именно поэтому компилятор осуществляет его сам, незаметно для разработчика:

```
byte b=3;  
int i=b;
```

В последней строке значение переменной `b` типа `byte` будет преобразовано к типу переменной `i` (то есть, `int`) автоматически, никаких специальных действий для этого предпринимать не надо.

Следующие 19 преобразований являются расширяющими:

- от `byte` к `short`, `int`, `long`, `float`, `double`
- от `short` к `int`, `long`, `float`, `double`
- от `char` к `int`, `long`, `float`, `double`
- от `int` к `long`, `float`, `double`
- от `long` к `float`, `double`
- от `float` к `double`

Обратите внимание, что нельзя провести преобразование к типу `char` от типов меньшей или равной длины (`byte`, `short`) или, наоборот, к `short` от `char` без потери данных. Это связано с тем, что `char` является беззнаковым в отличие от остальных целочисленных типов.

Тем не менее, следует помнить, что даже при расширении данные все-таки могут быть искажены в особых случаях. Они уже рассматривались в прошлой главе, это приведение значений `int` к типу `float` и приведение значений типа `long` к типу `float` или `double`. Хотя эти дробные типы вмещают гораздо большие числа, чем соответствующие целые, но у них меньше значащих знаков.

Повторим этот пример:

```
long l=1111111111111L;
float f = l;
l = (long) f;
print(l);
```

Результатом будет:

```
1111111110656
```

Обратное преобразование - сужение - означает, что переход осуществляется от более емкого типа к менее емкому. При таком преобразовании есть риск потерять данные. Например, если число типа `int` было больше 127, то при приведении его к `byte` значения битов старше восьмого будут потеряны. В Java такое преобразование должно совершаться явным образом, т.е. программист в коде должен явно указать, что он намеревается осуществить такое преобразование и готов идти на потерю данных.

Следующие 23 преобразования являются сужающими:

- от `byte` к `char`
- от `short` к `byte`, `char`
- от `char` к `byte`, `short`
- от `int` к `byte`, `short`, `char`
- от `long` к `byte`, `short`, `char`, `int`

- от float к byte, short, char, int, long
- от double к byte, short, char, int, long, float

При сужении целочисленного типа к более узкому целочисленному все старшие биты, не попадающие в новый тип, просто отбрасываются. Не производится никакого округления или других усилий для получения более корректного результата:

```
print((byte)383);
print((byte)384);
print((byte)-384);
```

Результатом будет:

```
127
-128
-128
```

Видно, что знаковый бит при сужении не оказал никакого влияния, так как был просто отброшен - результат приведения обратных чисел (384 и -384) оказался одинаковым. Стало быть, может быть потеряно не только точное абсолютное значение, но и знак величины.

Это верно и для типа char:

```
char c=40000;
print((short)c);
```

Результатом будет:

```
-25536
```

Сужение дробного типа к целочисленному является более сложной процедурой. Она проводится в два этапа.

На первом шаге дробное значение преобразуется в long, если целевым типом является long, или в int в противном случае (целевой тип byte, short, char или int). Для этого исходное дробное число сначала математически округляется в сторону нуля, то есть дробная часть просто отбрасывается.

Например, число 3.84 будет округлено до 3, а -3.84 превратится в -3. При этом могут возникнуть особые случаи:

- если исходное дробное значение является NaN, то результатом первого шага будет 0 выбранного типа (т.е. int или long);
- если исходное дробное значение является положительной или отрицательной бесконечностью, то результатом первого шага будет соответственно максимально или минимально возможное значение для выбранного типа (т.е. для int или long);
- наконец, если дробное значение было конечной величины, но в результате округления получилось слишком большое по модулю число для выбранного типа (т.е. для int или long), то, также как и в предыдущем пункте, результатом первого шага будет

соответственно максимально или минимально возможное значение этого типа. Если же результат округления укладывается в диапазон значений выбранного типа, то он и будет результатом первого шага.

На втором шаге производится дальнейшее сужение от выбранного целочисленного типа к целевому, если такое требуется, то есть может иметь место дополнительное преобразование от `int` к `byte`, `short` или `char`.

Проиллюстрируем описанный алгоритм преобразованием от бесконечности ко всем целочисленным типам:

```
float fmin = Float.NEGATIVE_INFINITY;
float fmax = Float.POSITIVE_INFINITY;
print("long: " + (long) fmin + ".." + (long) fmax);
print("int: " + (int) fmin + ".." + (int) fmax);
print("short: " + (short) fmin + ".." + (short) fmax);
print("char: " + (int) (char) fmin + ".." + (int) (char) fmax);
print("byte: " + (byte) fmin + ".." + (byte) fmax);
```

Результатом будет:

```
long: -9223372036854775808..9223372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1
```

Значения `long` и `int` вполне очевидны - дробные бесконечности преобразовались в соответственно минимально и максимально возможные значения этих типов. Результат для следующих трех типов (`short`, `char`, `byte`) есть по сути дальнейшее сужение значений, полученных для `int`, согласно второму шагу процедуры преобразования. А делается, как было описано, просто отбрасыванием старших битов. Вспомним, что минимально возможное значение в битовом виде представляется как `1000..000` (всего 32 бита для `int`, то есть единица и 31 ноль). Максимально возможное - `1111..111` (32 единицы). Отбрасывая старшие биты получаем для отрицательной бесконечности результат 0, одинаковый для всех 3 типов. Для положительной же бесконечности получаем результат, все биты которого равняются 1. Для знаковых типов `byte` и `short` такая комбинация рассматривается как `-1`, а для беззнакового `char` - как максимально возможное значение, то есть `65.535`.

Может сложиться впечатление, что для `char` приведение дает точное значение. Однако, это был частный случай - отбрасывание битов в большинстве случаев все же дает искажение. Например сужение дробного значения 2 миллиарда:

```
float f=2e9f;
print((int) (char) f);
print((int) (char) -f);
```

Результатом будет:

```
37888
27648
```

Обратите внимание на двойное приведение для значений типа `char` в двух последних примерах. Понятно, что преобразование от `char` к `int` не приводит к потере точности, но позволяет распечатывать не символ, а его числовой код, что более удобно для анализа.

В заключение еще раз обратим внимание на то, что примитивные значения типа `boolean` могут участвовать только в тождественных преобразованиях.

2.3. Преобразование ссылочных типов (расширение и сужение)

Переходим к ссылочным типам. Преобразование объектных типов лучше всего иллюстрируется с помощью дерева наследования. Рассмотрим небольшой пример наследования:

```
// Объявляем класс Parent
class Parent {
    int x;
}

// Объявляем класс Child, и наследуем
// его от класса Parent
class Child extends Parent {
    int y;
}

// Объявляем второго наследника
// класса Parent - класс Child2
class Child2 extends Parent {
    int z;
}
```

В каждом классе объявлено поле с уникальным именем. Будем рассматривать это поле как пример набора уникальных свойств, присущих некоторому объектному типу.

Объявленных 3 класса могут порождать 3 вида объектов. Объекты класса `Parent` обладают только одним полем `x`, а значит только ссылки типа `Parent` могут ссылаться на такие объекты. Объекты класса `Child` обладают полем `y` и полем `x`, полученным по наследству от класса `Parent`. Стало быть, на такие объекты могут указывать ссылки типа `Child` или `Parent`. Второй случай уже иллюстрировался следующим примером:

```
Parent p = new Child();
```

Обращаем внимание, что с помощью такой ссылки `p` можно обращаться лишь к полю `x` созданного объекта. Поле `y` не доступно, так как компилятор, проверяя корректность выражения `p.y`, не может предугадать, что ссылка `p` будет указывать на объект типа `Child` во время исполнения программы. Он анализирует лишь тип самой переменной, а она

объявлена как `Parent`, но в этом классе нет поля `y`, что и станет причиной возникновения ошибки компиляции.

Аналогично, объекты класса `Child2` обладают полем `z` и полем `x`, полученным по наследству от класса `Parent`. Стало быть, на такие объекты могут указывать ссылки типа `Child2` или `Parent`.

Таким образом, ссылки типа `Parent` могут указывать на объект любого из трех рассматриваемых типов, а ссылки типа `Child` и `Child2` - только на объекты точно такого же типа. Теперь можно перейти к преобразованию ссылочных типов на основе такого дерева наследования.

Расширение означает переход от более конкретного типа к менее конкретному, т.е. переход от детей к родителям. В нашем примере преобразование от любого наследника (`Child`, `Child2`) к родителю (`Parent`) есть расширение, переход к более общему типу. Подобно случаю с примитивными типами, этот переход производится самой JVM при необходимости и незаметен для разработчика, то есть не требует никаких дополнительных усилий, так как он всегда успешен: всегда можно обращаться к объекту, порожденному от наследника, по типу его родителя.

```
Parent p1=new Child();  
Parent p2=new Child2();
```

В обеих строках переменным типа `Parent` присваивается значение другого типа, а значит, происходит преобразование. Поскольку это расширение, оно производится автоматически и всегда успешно.

Обращаем внимание, что при подобном преобразовании с самим объектом ничего не происходит. Не смотря на то, что, например, поле `y` класса `Child` теперь больше не доступно, это не говорит о том, что оно исчезло. Такое существенное изменение структуры объекта невозможно. Он был порожден от класса `Child`, и навсегда сохранит все его свойства. Изменился лишь тип ссылки, через которую идет обращение к объекту. Эту ситуацию можно условно сравнить с рассматриванием некоего предмета через подзорную трубу. Если перейти от трубы с большим увеличением к более слабой, то видимых деталей станет меньше, но сам предмет, конечно, никак от этого не изменится.

Следующие преобразования являются расширяющими:

- от класса `A` к классу `B`, если `A` наследуется от `B` (важным частным случаем является преобразование от любого ссылочного типа к `Object`);
- от `null`-типа к любому объектному типу.

Второй случай иллюстрируется следующим примером:

```
Parent p=null;
```

Пустая ссылка `null` не обладает каким-либо конкретным ссылочным типом, поэтому иногда говорят о специальном `null`-типе. Однако на практике важно, что такое значение можно прозрачно преобразовать к любому объектному типу.

С изучением остальных ссылочных типов (интерфейсов и массивов) этот список будет расширяться.

Обратный переход, то есть движение по дереву наследования вниз, к наследникам, является сужением. Например для рассматриваемого случая, переход от ссылки типа Parent, которая может ссылаться на объекты трех классов, к ссылке типа Child, которая может ссылаться на объекты лишь одного из трех классов, очевидно, является сужением. Такой переход может оказаться невозможным. Если ссылка типа Parent ссылается на объект типа Parent или Child2, то переход к Child невозможен, ведь в обоих случаях объект не обладает полем y, которое объявлено в классе Child. Поэтому при сужении разработчику необходимо явным образом указывать на то, что необходимо попытаться провести такое преобразование. JVM во время исполнения проверит корректность перехода. Если он возможен, преобразование будет проведено. Если же нет - возникнет ошибка.

```
Parent p=new Child();
Child c=(Child)p; // преобразование будет успешным.
Parent p2=new Child2();
Child c2=(Child)p2; // во время исполнения возникнет ошибка!
```

Чтобы проверить, возможен ли желаемый переход, можно воспользоваться оператором instanceof:

```
Parent p=new Child();
if (p instanceof Child) {
    Child c = (Child)p;
}
Parent p2=new Child2();
if (p2 instanceof Child) {
    Child c = (Child)p2;
}
Parent p3=new Parent();
if (p3 instanceof Child) {
    Child c = (Child)p3;
}
```

В данном примере ошибок не возникнет. Первое преобразование возможно, и оно будет осуществлено. Во втором и третьем случаях условия операторов if не сработают, и попыток некорректного перехода не будет.

На данный момент можно назвать лишь одно сужающее преобразование:

- от класса A к классу B, если B наследуется от A (важным частным случаем является сужение типа Object до любого другого ссылочного типа);

С изучением остальных ссылочных типов (интерфейсов и массивов) этот список будет расширяться.

2.4. Преобразование к строке

Это преобразование уже не раз упоминалось. Любой тип может быть приведен к строке, т.е. к экземпляру класса String. Это преобразование является исключительным в силу того, что охватывает абсолютно все типы, в том числе и boolean, про который говорилось, что он не может участвовать ни в каком другом приведении, кроме тождественного.

Напомним, как преобразуются различные типы:

- Числовые типы записываются в текстовом виде без потери точности представления. Формально, такое преобразование происходит в два этапа. Сначала на основе примитивного значения порождается экземпляр соответствующего класса-обертки, а затем у него вызывается метод `toString()`. Но поскольку эти действия никак не заметны снаружи, то многие JVM оптимизируют их и преобразуют примитивные значения в текст напрямую.
- Булевская величина приводится к строке `"true"` или `"false"` в зависимости от значения.
- Для объектных величин вызывается метод `toString()`. Если метод возвращает `null`, то результатом будет строка `"null"`.
- Для `null`-значения генерируется строка `"null"`.

2.5. Запрещенные преобразования

Не все переходы между произвольными типами допустимы. Например, к запрещенным преобразованиям относятся: переходы от любого ссылочного типа к примитивному, от примитивного - к ссылочному (кроме преобразований к строке). Уже упоминавшийся пример - тип `boolean` нельзя привести ни к какому другому типу, отличному от `boolean` (как обычно - за исключением приведения к строке). Затем, невозможно привести друг к другу типы, находящиеся не на одной, а на соседних ветвях дерева наследования. В примере, который рассматривался для иллюстрации преобразований ссылочных типов, переход от `Child` к `Child2` запрещен. В самом деле, ссылка типа `Child` может указывать на объекты, порожденные только от класса `Child` или его наследников. Что исключает вероятность того, что объект будет совместим с типом `Child2`.

На этом список запрещенных преобразований не исчерпывается. Однако, он довольно велик, и в то же время все варианты довольно очевидны, и поэтому подробно рассматриваться не будут. Желающие могут получить полную информацию из спецификации.

Разумеется, попытка осуществить запрещенное преобразование вызовет ошибку компиляции.

3. Применение приведений

Теперь, когда рассмотрены все виды преобразований, перейдем к ситуациям в коде, где могут встретиться или потребоваться приведения.

Такие ситуации могут быть сгруппированы следующим образом:

- присвоение значений переменным (assignment). Не все переходы допустимы при таком преобразовании - ограничения выбраны таким образом, чтобы не могла возникнуть ошибочная ситуация.
- вызов метода. Это преобразование применяется к аргументам вызываемого метода или конструктора. Допускаются почти те же переходы, что и для присвоения значений. Такое приведение никогда не порождает ошибок. Также приведение осуществляется при возвращении значения из метода.

- явное приведение. В этом случае явно указывается, к какому типу требуется привести исходное значение. Допускаются все виды преобразований кроме приведений к строке и запрещенных. Может возникать ошибка времени исполнения программы.
- оператор конкатенации производит преобразование к строке своих аргументов.
- числовое расширение (numeric promotion). Числовые операции могут потребовать изменения типа аргумента(ов). Это преобразование имеет особое название - расширение (promotion), так как выбор целевого типа может зависеть не только от исходного значения, но и от второго аргумента операции.

Рассмотрим все случаи более подробно.

3.1. Присвоение значений

Эти ситуации неоднократно применялись в этой главе для иллюстрации видов преобразования. Приведение может потребоваться, если переменной одного типа присваивается значение другого типа. Возможны следующие комбинации.

Если сочетание этих двух типов образует запрещенное приведение, возникнет ошибка. Например, примитивные значения нельзя присваивать объектным переменным, включая следующие примеры:

```
// пример вызовет ошибку компиляции

// примитивное значение нельзя
// присвоить объектной переменной
Parent p = 3;

// приведение к классу-обертке также запрещено
Long l=5L;

// универсальное приведение к строке
// возможно только для оператора +
String s=true;
```

Далее, если сочетание этих двух типов образует расширение (примитивных или ссылочных типов), то оно будет осуществлено автоматически, неявным для разработчика образом:

```
int i=10;
long l=i;
Child c = new Child();
Parent p=c;
```

Если же сочетание оказывается сужением, то возникает ошибка компиляции, такой переход не может быть проведен неявно:

```
// пример вызовет ошибку компиляции
int i=10;
short s=i; // ошибка! сужение!
```

```
Parent p = new Child();
Child c=p; // ошибка! сужение!
```

Как уже упоминалось, в подобных случаях необходимо преобразование делать явно:

```
int i=10;
short s=(int)i;
Parent p = new Child();
Child c=(Child)p;
```

Более подробно явное сужение рассматривается ниже.

Здесь может вызвать удивление следующая ситуация, которая не порождает ошибок компиляции:

```
byte b=1;
short s=2+3;
char c=(byte)5+'a';
```

В первой строке переменной типа `byte` присваивается значение целочисленного литерала типа `int`, что является сужением. Во второй строке переменной типа `short` присваивается результат сложения двух литералов типа `int`, а тип этой суммы также `int`. Наконец, в третьей строке переменной типа `char` присваивается результат сложения числа 5, приведенного к типу `byte`, и символьного литерала.

Однако, все эти примеры корректны. Для удобства разработчика компилятор проводит дополнительный анализ при присвоении значений переменным типа `byte`, `short` и `char`. Если таким переменным присваивается величина типа `byte`, `short`, `char` или `int`, причем ее значение может быть получено уже на момент компиляции, и оказывается, что это значение укладывается в диапазон типа переменной, то явного приведения не требуется. Если бы такой возможности не было, то пришлось бы писать так:

```
byte b=(byte)1; // преобразование необязательно
short s=(short)(2+3); // преобразование необязательно
char c=(char)((byte)5+'a'); // преобразование необязательно

// преобразование необходимо, так как
// число 200 не укладывается в тип byte
byte b2=(byte)200;
```

3.2. Вызов метода

Это приведение возникает в случае, когда вызывается метод с объявленными параметрами одних типов, а при вызове передаются аргументы других типов. Объявление методов рассматривается в следующих главах курса, однако такой простой пример вполне понятен:

```
// объявление метода с параметром типа long
void calculate(long l) {
    ...
}
```

```
void main() {
    calculate(5);
}
```

Как видно, при вызове метода передается значение типа `int`, а не `long`, как определено в объявлении этого метода.

Здесь компилятор предпринимает ровно те же шаги, что и при приведении при присвоении значений переменным. Если типы образуют запрещенное преобразование, то будет ошибка.

```
// пример вызовет ошибку компиляции

void calculate(long l) {
    ...
}

void main() {
    calculate(new Long(5)); // здесь будет ошибка
}
```

Если сужение, то компилятор не сможет осуществить приведение, и потребуются явные указания.

```
void calculate(int l) {
    ...
}

void main() {
    long l=5;
    // calculate(l); // сужение! так будет ошибка.
    calculate((int)l); // корректный вызов
}
```

Наконец, в случае расширения, компилятор осуществит приведение сам, как и было показано в примере в начале этого раздела.

Надо отметить, что в отличие от ситуации присвоения, при вызове методов компилятор не производит преобразований примитивных значений от `byte`, `short`, `char` или `int` к `byte`, `short` или `char`. Это привело бы к усложнению работы с перегруженными методами. Например:

```
// пример вызовет ошибку компиляции

// объявляем перегруженные методы
// с аргументами (byte, int) и (short, short)
int m(byte a, int b) { return a+b; }
int m(short a, short b) { return a-b; }

void main() {
```

```
print(m(12, 2)); // ошибка компиляции!  
}
```

В этом примере компилятор выдаст ошибку, так как при вызове аргументы имеют тип (int, int), а метода с такими параметрами нет. Если бы компилятор проводил преобразование для целых величин, подобно ситуации с присвоением значений, то пример стал бы корректным, но пришлось бы предпринимать дополнительные усилия, чтобы указать, какой из двух возможных перегруженных методов хотелось бы вызвать.

Затем, аналогичное преобразование требуется при возвращении значения из метода, если тип результата и заявленный тип возвращаемого значения не совпадают.

```
long get() {  
    return 5;  
}
```

Хотя в выражении return указан целочисленный литерал типа int, во всех местах, где будет вызван этот метод, будет получено значение типа long. Для такого преобразования действуют все правила, что и для присвоения значения.

В заключение рассмотрим пример, включающий в себя все рассмотренные случаи преобразования:

```
short get(Parent p) {  
    return 5+'A'; // приведение при возвращении значения  
}  
  
void main() {  
    long l = // приведение при присвоении значения  
    get(new Child()); // приведение при вызове метода  
}
```

3.3. Явное приведение

Явное приведение уже многократно использовалось в примерах. При таком преобразовании слева от выражения, тип значения которого необходимо преобразовать, в круглых скобках указывается целевой тип. Если преобразование пройдет успешно, то результат будет точно указанного типа. Примеры:

```
(byte)5  
(Parent)new Child()  
(Flat)getCity().getStreet().getHouse().getFlat()
```

Если комбинация типов образует запрещенное преобразование, возникает ошибка компиляции. Допускаются тождественные преобразования, расширения простых и объектных типов, сужения простых и объектных типов. Первые три выполняются успешно всегда. Последние два могут стать причиной ошибки исполнения, если значения оказались не совместимыми. Как следствие, выражение null всегда может быть успешно преобразовано к любому ссылочному типу.

Легко найти способ все-таки закодировать запрещенное преобразование.

```
Child c=new Child();  
// Child2 c2=(Child2)c; // запрещенное преобразование  
Parent p=c; // расширение  
Child2 c2=(Child2)p; // сужение
```

Такой код будет успешно скомпилирован, однако, разумеется, при исполнении он всегда будет генерировать ошибку в последней строке. "Обманывать" компилятор смысла нет.

3.4. Оператор конкатенации строк

Этот оператор уже был подробно рассмотрен. Если обоими его аргументами являются строки, то происходит обычная конкатенация. Если же тип `String` имеет лишь один из аргументов, то второй необходимо преобразовать в текст. Это единственная операция, при которой производится универсальное приведение любого значения к типу `String`.

Это одно из свойств, выделяющих класс `String` из всех остальных.

Правила преобразования уже были подробно описаны в этой главе, а оператор конкатенации рассматривался в главе "Типы данных".

Небольшой пример:

```
int i=1;  
double d=i/2.;  
String s="text";  
print("i="+i+", d="+d+", s="+s);
```

Результатом будет:

```
i=1, d=0.5, s=text
```

3.5. Числовое расширение

Наконец, последний вид преобразований применяется при числовых операциях, когда требуется привести аргумент(ы) к типу длиной в 32 или 64 бита для проведения вычислений. Таким образом, при числовом расширении осуществляется только расширение примитивных типов.

Различают унарное и бинарное числовое расширение.

3.5.1. Унарное числовое расширение

Это преобразование расширяет примитивные типы `byte`, `short` или `char` до типов `int` по правилам расширения примитивных типов.

Унарное числовое расширение может происходить при следующих операциях:

- унарные операции `+` и `-`;
- битовое отрицание `~`;
- операции битового сдвига `<<`, `>>`, `>>>`.

Операторы сдвига имеют два аргумента, но они расширяются независимо друг от друга, поэтому это преобразование является унарным. Таким образом, результат выражения `5<<3L` имеет тип `int`. Вообще, результат операторов сдвига всегда имеет тип `int` или `long`.

Примеры работы всех этих операторов с учетом расширения подробно рассматривались в предыдущих главах.

3.5.2. Бинарное числовое расширение

Это преобразование расширяет все примитивные числовые типы кроме `double` до типов `int`, `long`, `float`, `double` по правилам расширения примитивных типов. Бинарное числовое расширение происходит при числовых операторах, имеющих два аргумента, по следующим правилам:

- если любой из аргументов имеет тип `double`, то и второй приводится к `double`;
- иначе, если любой из аргументов имеет тип `float`, то и второй приводится к `float`;
- иначе, если любой из аргументов имеет тип `long`, то и второй приводится к `long`;
- иначе оба аргумента приводятся к `int`.

Бинарное числовое расширение может происходить при следующих операциях:

- арифметические операции `+`, `-`, `*`, `/`, `%`;
- операции сравнения `<`, `<=`, `>`, `>=`, `==`, `!=`;
- битовые операции `&`, `|`, `^`;
- в некоторых случаях для операции с условием `?:`.

Примеры работы всех этих операторов с учетом расширения подробно рассматривались в предыдущих главах.

4. Тип переменной и тип ее значения

Теперь, когда были подробно рассмотрены все примеры преобразований, нужно вернуться к вопросу переменной и ее значений.

Как уже говорилось, переменная определяется тремя базовыми характеристиками: имя, тип, значение. Имя дается произвольным образом и никак не сказывается на свойствах переменной. А вот значение всегда имеет некоторый тип, не обязательно совпадающий с типом самой переменной. Поэтому необходимо рассмотреть все возможные типы переменных и выяснить, значения каких типов они могут в себе хранить.

Начнем с переменных примитивных типов. Поскольку эти переменные действительно хранят само значение, то их тип всегда точно совпадает с типом значения.

Проиллюстрируем это правило на примере:

```
byte b=3;
char c='A'+3;
long m=b+c;
double d=m-3F;
```

Здесь переменная `b` будет хранить значение типа `byte` после сужения целочисленного литерала типа `int`. Переменная `c` будет хранить тип `char` после того, как компилятор осуществит сужающее преобразование результата суммирования, который будет иметь тип `int`. Для переменной `l` выполнится расширения результата суммирования типа от `int` к типу `long`. Наконец, переменная `d` будет хранить значение типа `double`, получившееся в результате расширения результата разности, который имеет тип `float`.

Переходим к ссылочным типам. Во-первых, значение любой переменной такого типа - ссылка, которая может указывать только на объекты, порожденные от тех или иных классов, и далее обсуждаются только свойства этих классов. (Также объекты могут порождаться от массивов, эта тема рассматривается в отдельной главе).

Кроме этого, ссылочная переменная любого типа может иметь значение `null`. Большинство действий над такой переменной, например, обращение к полям или методам, приведет к ошибке.

Итак, какова связь между типом ссылочной переменной и ее значения? Здесь главное ограничение - проверка компилятора, который следит, чтобы все действия, выполняющиеся над объектом, были корректны. Компилятор не может предугадать, на объект какого класса будет реально ссылаться та или иная переменная. Все, чем он располагает - тип самой переменной. Именно его и использует компилятор для проверок. А значит, все допустимые значения переменной должны гарантировано обладать свойствами, определенными в классе-типе этой переменной. Такую гарантию дает только наследование. Отсюда получаем правило: ссылочная переменная типа `A` может указывать на объекты, порожденные от самого типа `A` или его наследников.

```
Point p = new Point();
```

В этом примере переменная и ее значение одинакового типа, поэтому над объектом можно совершать все возможные для данного класса действия.

```
Parent p = new Child();
```

Такое присвоение корректно, так как класс `Child` порожден от `Parent`. Однако теперь допустимые действия над переменной `p`, а значит, над объектом, только что созданным на основе класса `Child`, ограничены только теми, которые были унаследованы от класса `Parent`. Например, если в классе `Child` определен некий новый метод `newChildMethod()`, то попытка его вызвать `p.newChildMethod()` будет порождать ошибку компиляции. Необходимо подчеркнуть, что не происходит никаких изменений с самим объектом, ограничение порождается используемым способом доступа к этому объекту - переменной типа `Parent`.

Чтобы показать, что объект не потерял никаких своих свойств, произведем следующее обращение:

```
((Child)p).newChildMethod();
```

Здесь в начале проводится явное сужение к типу `Child`. Во время исполнения программы JVM проверит, что тип объекта, на который ссылается переменная `p` совместим с типом `Child`. В нашем случае это именно так. В результате получается ссылка типа `Child`, поэтому становится допустимым вызов метода `newChildMethod()`, который вызывается у объекта, созданного в предыдущей строке.

Обратим внимание на важный частный случай - переменная типа `Object` может ссылаться на объекты любого типа.

В дальнейшем, с изучением новых типов (абстрактных классов, интерфейсов, массивов) этот список будет продолжаться, а пока приведем краткое обобщение того, что было рассмотрено в этом разделе.

| Тип переменной | Допустимые типы ее значения |
|---------------------|---|
| Примитивный | В точности совпадает с типом переменной |
| Ссылочный | <ul style="list-style-type: none">• <code>null</code>• совпадающий с типом переменной• классы-наследники от типа переменной |
| <code>Object</code> | <ul style="list-style-type: none">• <code>null</code>• любой ссылочный |

5. Заключение

В этой главе были рассмотрены правила работы с типами данных в строго типизированном языке Java. Поскольку компилятор строго отслеживает тип каждой переменной и каждого выражения, в случае изменения этого типа необходимо четко понимать, какие действия допустимы, а какие нет, с точки зрения компилятора и виртуальной машины.

Были рассмотрены все виды приведения типов в Java, то есть переход от одного типа к другому. Они разбиваются на 7 групп, начиная с тождественного и заканчивая запрещенными. Основные 4 вида определяются сужающими или расширяющими переходами между простыми или ссылочными типами. Важно помнить, что при явном сужении числовых типов, старшие биты просто отбрасываются, что порой приводит к неожиданному результату. Что касается преобразования ссылочных значений, то здесь действует правило – преобразование никогда не порождает новых и не изменяет существующих объектов. Меняется лишь способ работы с ними.

Особенными в Java является преобразование к строке.

Затем были рассмотрены все ситуации в программе, где могут происходить преобразования типов. Во-первых, это присвоение значений, когда зачастую преобразование происходит не заметно для программиста. Вызов метода во многом похож на инициализацию. Явное приведение позволяет осуществить желаемый переход в случае, когда компилятор не позволяет сделать это неявно. Преобразование при выполнении числовых операций оказывает существенное влияние на результат.

В заключение была рассмотрена связь между типом переменной и типом ее значения.

6. Контрольные вопросы

7-1. Корректен ли следующий пример кода, и если да, то сколько преобразований и каких видов будет произведено при его исполнении?

```
byte b=1;
long m=-b;
Object o="";
```

```
print("m"+o+m);
```

a.) Пример корректен. Преобразования:

- первая строка: преобразование при присвоении, сужение от `int` к `byte`
- вторая: числовое расширение при вычислении оператора унарный минус от `byte` к `int`, расширение при присвоении значения от `int` к `long`
- третья: расширение при присвоении `String` к `Object`
- четвертая: приведения к строке: от `Object` к `String` и от `long` к `String` (через `Long`).

7-2. Произойдет ли потеря точности при следующем преобразовании?

```
float f = -16777217;
```

a.) Да. Число 16777217 записывается как 100000000000000000000001 в двоичной системе, всего 25 бит. А тип `float` отводит 24 бита для хранения мантииссы (значащих цифр), причем один из них знаковый. После отбрасывания не уместившихся битов получаем 100000000000000000000000, или `-16777216` в десятичной системе. Таким образом, теряется точность в последнем разряде. (Для получения правильного ответа достаточно запустить подобный пример и увидеть результат.)

7-3. Корректен ли следующий пример? Если нет, то в каких строках какие ошибки будут сгенерированы?

```
byte b=100-100;  
byte b=100+100;  
byte b=100*100;
```

a.) Результат вычитания в первой строке равен 0, что укладывается в тип `byte`, ошибки не будет. Результат вычислений в строках 2 и 3 выходит за рамки `byte` (максимально допустимое значение 127), поэтому неявное сужение не допустимо.

7-4. Для каких значений числовой переменной `x` будет верно следующее выражение?

```
x==--x
```

a.) Для целочисленных типов:

- 0
- самое наименьшее значение для типов `int` и `long`. Например, для типа `int` это `-2147483648`

Для дробных типов:

- значения 0.0 и `-0.0`

7-5. Верны ли следующие выражения для переменной `d` дробного типа?

```
(short) d == (short) (int) d  
(int) d == (int) (long) d
```

- a.) Первое выражение верно, так как при приведении к `short`, дробное значение всегда сначала приводится к `int`.

Второе выражение не верно, в чем можно легко убедиться, вычислив его для любого `d`, большего, чем максимально допустимое значение типа `int`. Левая часть будет равна `231-1`, в то время как правая будет вычислена путем отбрасывания битов, старше 32:

```
double d=3e9;  
System.out.println((int) d);  
System.out.println((int) (long) d);
```

Результатом будет:

```
2147483647  
-1294967296
```

7-6. Корректны ли следующие преобразования?

```
Object o = (String) null;  
String s = o;
```

- a.) Нет. Хотя значение `null` можно привести к любому ссылочному типу, во 2 строке происходит сужение типа ссылки (`Object`) до класса `String`. Такое действие должно происходить явно:

```
String s = (String) o;
```

7-7. Пусть классы `Wolf` и `Rabbit` являются наследниками класса `Animal`. Корректен ли следующий пример?

```
Wolf w = new Wolf();  
Animal a = (Animal) w;  
Rabbit r = (Rabbit) a;
```

- a.) Хотя этот пример будет корректно скомпилирован, при выполнении он всегда будет порождать ошибку. Не смотря на приведение, переменная «а» ссылается на объект типа `Wolf`, созданный в первой строке.

В третьей строке при приведении ссылки на объект класса `Wolf` к типу `Rabbit` будет возникать ошибка преобразования типов, поскольку эти классы не находятся на одной ветви дерева наследования.

7-8. Какие значения не могут участвовать в преобразовании к строке?

а.) Все могут.

7-9. Какие преобразования будут производиться при работе следующего метода?

```
public int add(byte a, byte b) {  
    short x=(short)a;  
    char y=(char)b;  
    return x+y;  
}
```

а.) Преобразования:

- первая строка: явное приведение от byte к short
- вторая: явное приведение от byte к char
- третья: числовое расширение при вычислении бинарного оператора сложения: от short и char к int

7-10. Какое значение появится на консоли после выполнения следующего кода?

```
char c=65;  
print(c);  
print(+c);  
print(""+c);
```

а.) Ответ:

A
65
=A

В первом случае выводится значение типа char, поэтому отображается символ. Во втором случае выводится значение унарного оператора плюс, то есть число типа int. В третьем случае при сложении со строкой происходит преобразование от типа char к String, то есть запишется символ.

7-11. Значение какого типа будет хранить переменная после инициализации?

```
byte b=1+2;
```

а.) Примитивные переменные всегда хранят значения точного того же типа, что и они сами, то есть, в данном примере – byte. Это значение будет получено после неявного сужения при присвоении от типа int.

7-12. Значения какого типа будут хранить переменные после инициализации?

```
Object o = "123";  
Child c = new Child();  
Parent p=(Parent)new Child();
```

```
Child x=null;
```

a.) В порядке следования:

- String
- Child
- Child
- null