Национальный Открытый Университет "ИНТУИТ": www.intuit.ru
Вадим Монахов
Лекция 6. Начальные сведения об объектном программировании

В "Объектно-ориентированное проектирование и платформа NetBeans" мы уже познакомились с первым принципом объектного программирования – *инкапсуляцией*. Затем научились пользоваться уже готовыми классами. Это – начальная стадия изучения объектного программирования. Для того чтобы овладеть его основными возможностями, требуется научиться создавать собственные классы, изменяющие и усложняющие поведение существующих классов. Важнейшими элементами такого умения является использование *наследования* и *полиморфизма*.

6.1. Наследование и полиморфизм. UML-диаграммы

Наследование опирается на инкапсуляцию. Оно позволяет строить на основе первоначального класса новые, добавляя в классы новые поля данных и методы. Первоначальный *класс* называется *прародителем* (*ancestor*), новые классы – его *потомками* (*descendants*). От потомков, в свою *очередь*, можно наследовать, получая очередных потомков. И так далее. Набор классов, связанных *отношением наследования*, называется *иерархией классов*. А *класс*, стоящий во главе иерархии, от которого унаследованы все остальные (прямо или опосредованно), называется базовым *классом иерархии*. В *Java* все классы являются потомками класса `Object`. То есть он является базовым для всех классов. Тем не менее, если рассматривается поведение, характерное для объектов какого-то класса и всех потомков этого класса, говорят об иерархии, начинающейся с этого класса. В этом случае именно он является базовым классом иерархии.

Полиморфизм опирается как на инкапсуляцию, так и на *наследование*. Как показывает *опыт* преподавания, это наиболее сложный для понимания принцип. *Слово* "*полиморфизм*" в переводе с греческого означает "имеющий много форм". В объектном программировании под полиморфизмом подразумевается наличие кода, написанного для объектов, имеющих тип базового класса иерархии. При этом такой код должен правильно работать для любого объекта, являющегося экземпляром класса из данной иерархии. Независимо от того, где этот *класс* расположен в иерархии. Такой код и называется *полиморфным*. При написании полиморфного кода заранее неизвестно, для объектов какого типа он будет работать – один и тот же метод будет исполняться *по*-разному в зависимости от типа объекта. Пусть, например, у нас имеется *класс* `Figure` – "фигура", и в нем заданы методы `show()` – показать фигуру на экране, и `hide()` – скрыть ее. Тогда для переменной `figure` типа `Figure` вызовы `figure.show()` и `figure.hide()` будут показывать или скрывать *объект*, на который ссылается эта *переменная*. Причем сам *объект* "знает", как себя показывать или скрывать, а код пишется на уровне абстракций этих действий.

Основное преимущество объектного программирования *по* сравнению с процедурным как раз и заключается в возможности написания полиморфного кода. Именно для этого пишется *иерархия классов*. *Полиморфизм* позволяет резко увеличить *коэффициент повторного использования* программного кода и его *модифицируемость по* сравнению с *процедурным программированием*.

В качестве примера того, как строится *иерархия*, рассмотрим иерархию фигур, отрисовываемых на экране – она показана на рисунке. В ней базовым классом является `Figure`, от которого наследуются `Dot` – "точка", `Triangle` – "треугольник" и `Square` – "квадрат". От `Dot` наследуется *класс* `Circle` – "*окружность*", а от `Circle` унаследуем `Ellipse` – "*эллипс*". И, наконец, от `Square` унаследуем `Rectangle` – "*прямоугольник*".

Отметим, что в иерархии принято рисовать стрелки в направлении от наследника к прародителю. Такое направление называется *Generalization* – "*обобщение*", "генерализация". Оно противоположно направлению наследования, которое принято называть Specialization – "специализация". Стрелки символизируют направление в сторону упрощения.

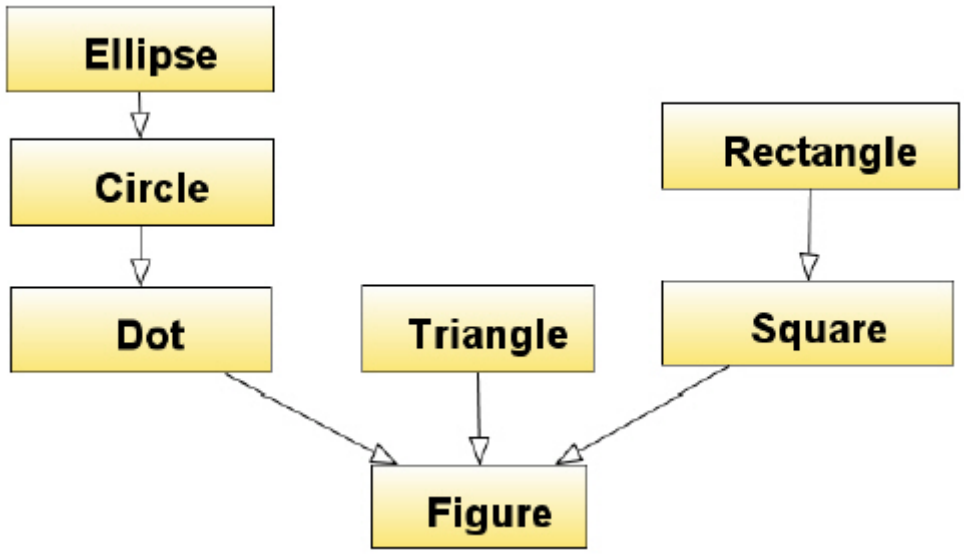


Рис. 6.1. Иерархия фигур, отрисовываемых на экране

Часто *класс*-прародитель называют *суперклассом* (*superclass*), а *класс*-наследник – *субклассом* (*subclass*). Но такая терминология подталкивает начинающих программистов к неверной логике: *суперкласс* пытаются сделать "суперсложным". Так, чтобы его *подклассы* (это неверно воспринимается синонимом выражению "упрощенные разновидности") обладали упрощенным *по* сравнению с ним поведением. На деле же **потомки должны обладать более сложным устройством и поведением по сравнению с прародителем.** Поэтому в данном учебном пособии предпочтение отдается терминам "прародитель" и "наследник".

Чем ближе к основанию иерархии лежит *класс*, тем более общим и универсальным (*general*) он является. И одновременно – более простым. *Класс*, который лежит в основе иерархии, называется *базовым классом* этой иерархии. *Базовый класс* всегда называют именем, которое характеризует все объекты – экземпляры классов-наследников, и которое выражает наиболее общую абстракцию, применимую к таким объектам. В нашем случае это *класс* `Figure`. Любая фигура будет иметь поля данных x и y – *координаты* фигуры на экране.

Класс `Dot` ("точка") является наследником `Figure`, поэтому он будет иметь поля данных x и y, наследуемые от `Figure`. То есть в самом классе `Dot` задавать эти поля не надо. От `Dot` мы наследуем *класс* `Circle` ("*окружность*"), поэтому в нем также имеется поля `x` и `y`, наследуемые от `Figure`. Но появляется дополнительное *поле* данных. У `Circle` это *поле*, соответствующее радиусу. Мы назовем его `r`. Кроме того, для окружности возможна операция изменения радиуса, поэтому в ней может появиться новый метод, обеспечивающий это действие – назовем его `setSize` ("установить размер"). *Класс* `Ellipse` имеет те же поля данных и обеспечивает то же поведение, что и `Circle`, но в этом классе появляется дополнительное *поле* данных `r2` – *длина* второй полуоси эллипса, и возможность регулировать *значение* этого поля. Возможен и другой подход, в некотором роде более логичный: считать *эллипс* сплюснутой или растянутой окружностью. В этом случае необходимо ввести коэффициент растяжения (*aspect ratio*). Назовем его `k`. Тогда *эллипс* будет характеризоваться радиусом `r` и коэффициентом растяжения `k`. Метод, обеспечивающий изменение `k`, назовем `stretch` ("растянуть"). Обратим внимание, что исходя из выбранной логики действий метод `scale` должен приводить к изменению поля `r` и не затрагивать *поле* `k` – поэтому *эллипс* будет масштабироваться без изменения формы.

Каждый из классов этой ветви иерархии фигур можно считать описанием "усложненной точки". При этом важно, что любой *объект* такого типа можно считать "точкой, которую усложнили". Грубо говоря, считать, что круг или *эллипс* – это такая "жирная точка". Аналогичным образом `Ellipse` является "усложненной окружностью"

Аналогично, *класс* `Square` наследует поля `x` и `y`, но в нем добавляется *поле*, соответствующее стороне квадрата. Мы назовем его `a`. У `Triangle` в качестве новых, не унаследованных полей данных могут выступать *координаты* вершин треугольника; либо *координаты* одной из вершин, длины прилегающих к ней сторон и угол между ними, и так далее.

Как располагать классы иерархии, *базовый класс* внизу, а наследники вверху, образуя ветви дерева наследования, или наоборот, *базовый класс* вверху а наследники внизу, образуя "корни" дерева наследования – принципиального значения не имеет. *По*-видимому, на начальном этапе развития объектного программирования применялся первый вариант, почему *базовый класс*, лежащий в основе иерархии, и получил такое название. Такой вариант выбран в данном учебном пособии, поскольку именно он используется в NetBeans *Enterprise* Pack. Хотя в настоящее время чаще используют второй вариант, когда *базовый класс* располагают сверху.

В литературе *по* объектному программированию часто встречается следующий критерий:"если имеются классы `A1` и `A2`, и можно сказать, что `A2` является частным случаем `A1`, то `A2` должен описываться как *потомок* `A1`". Данный критерий не совсем корректен.

Очень часто встречающийся вариант ошибочных рассуждений, основанный на нем, и приводящий к **неправильному построению иерархии**, выглядит так:"поскольку `Circle` является частным случаем `Ellipse` (при равных длинах полуосей), а `Dot` является частным случаем `Circle` (при нулевом радиусе), то *класс* `Ellipse` более общий, чем `Circle`, а `Circle` – более общий, чем `Dot`. Поэтому `Ellipse` должен являться прародителем для `Circle`, а `Circle` должен являться прародителем для `Dot`". Ошибка заключается в неправильном понимании идей"общности" и"специализации", а также характерной путанице, когда объекты не отличают от классов.

Каждый *объект* класса-потомка при **любых** значениях полей должен рассматриваться как экземпляр класса-прародителя, и с тем же поведением на уровне абстракции действий. Но только с некоторыми изменениями на уровне реализации этих действий. В концепции наследования основное внимание уделяется *поведению* объектов. Объекты с разным поведением имеют другой тип. А значения полей данных характеризуют *состояние* объекта, но не его тип.

Мы говорим про абстракции поведения как на те характерные действия, которые могут быть описаны на уровне полиморфного кода, безотносительно к конкретной реализации в конкретном классе.

По своему поведению любой *объект-эллипс* вполне может рассматриваться как экземпляр типа"*Окружность*" и даже вести себя в точности как *окружность*. Но не наоборот – объекты типа *Окружность* не обладает поведением Эллипса. Мы намеренно используем заглавные буквы для того, чтобы не путать классы с объектами. Если для эллипса можно изменить *значение* `aspectRatio` (вызвать метод `setAspectRatio` (*новое значение*)), то для окружности такая операция не имеет смысла или запрещена. Аналогично, и для эллипса, и для окружности имеет смысл операция установки нового размера `setSize` (*новое значение*), а для точки она не имеет смысла или запрещена. И даже если построить неправильную иерархию `Ellipse-Circle-Dot` и унаследовать от `Ellipse` эти методы в `Circle` и `Dot`, возникнет проблема с их переопределением. Если `setAspectRatio` будет менять *отношение* полуосей нашей"окружности" – она перестанет быть окружностью. Аналогично, если `setSize` изменит размер точки – та перестанет быть точкой. Если же сделать эти методы ничего не делающими"заглушками" – экземпляры таких потомков не смогут обладать поведением прародителя. Например, мы не сможем вписать *окружность* в *прямоугольник*, установив нужное *значение* `aspectRatio` – найдутся только три точки, общие для окружности и сторон прямоугольника, а не четыре, как для объекта типа `Ellipse`. То есть *объект* типа `Circle` на уровне абстракции поведения во многих случаях не сможет обладать всеми особенностями поведения объекта типа `Ellipse`. А значит, `Circle` не может быть потомком `Ellipse`.

Можно привести нескончаемое число других примеров того, какие ситуации окажутся нереализуемыми для объектов таких неправильных иерархий. А отдельные хитрости, позволяющие выпутываться из некоторых из таких ситуаций, обычно бывают крайне искусственными, не позволяют решить проблему с очередной внезапно возникшей ситуацией, и только усложняют программу.

Сформулируем критерий того, когда следует использовать *наследование*, более корректно:" *если имеются классы A1 и A2, и можно считать, что A2 является модифицированным (усложненным или измененным) вариантом A1 с сохранением всех особенностей поведения A1, то A2 должен описываться как потомок A1. – На уровне абстракции, описывающей поведение, объект типа A2 должен вести себя, как объект типа A1 при любых значениях полей данных* ".

Специализированный *класс*, вообще говоря, должен быть устроен *более сложно* ("расширенно" – extended) *по* сравнению с прародительским. У него должны иметься дополнительные поля данных и/или дополнительные методы. С этой точки зрения очевидно, что *Окружность* более специализирована, чем Точка, а *Эллипс* более специализирован, чем *Окружность*. Иногда встречаются ситуации, когда *потомок* отличается от прародителя только своим поведением. У него не добавляется новых полей или методов, а только переопределяется часть методов (возможно, только один). Отметим, что поля или методы, имеющиеся в прародителе, не могут отсутствовать в наследнике – они наследуются из прародителя. Даже если *доступ* к ним в классе-наследнике закрыт (так бывает в случае, когда *поле* или метод объявлены с *модификатором видимости* `private` -"закрытый","частный").

Когда про *класс-потомок* можно сказать, что он является специализированной разновидностью класса-прародителя ("В *есть* А"), все очевидно. Но в объектном программировании иногда приходится использовать *отношение*"*Класс* В *похож* на А – имеет те же поля данных, плюс, возможно, дополнительные, но обладает несколько иным поведением".

Любой наш *объект* мы можем назвать фигурой. Поэтому то, что *базовый класс* нашей иерархии называется `Figure`, естественно и однозначно. И однозначно то, что все классы нашей иерархии должны быть его наследниками. А вот остальные элементы иерархии можно было бы устроить совсем *по*-другому. Например, так, как показано на следующем рисунке.

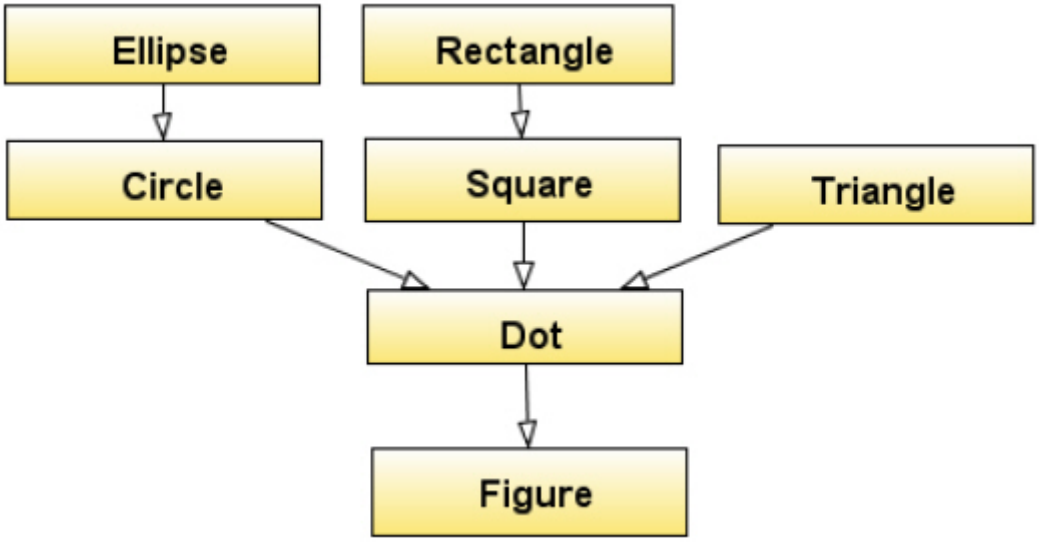


Рис. 6.2. Альтернативный вариант иерархии фигур

Возможно и такое решение: все указанные классы сделать наследниками `Figure` и расположить на одном уровне наследования.

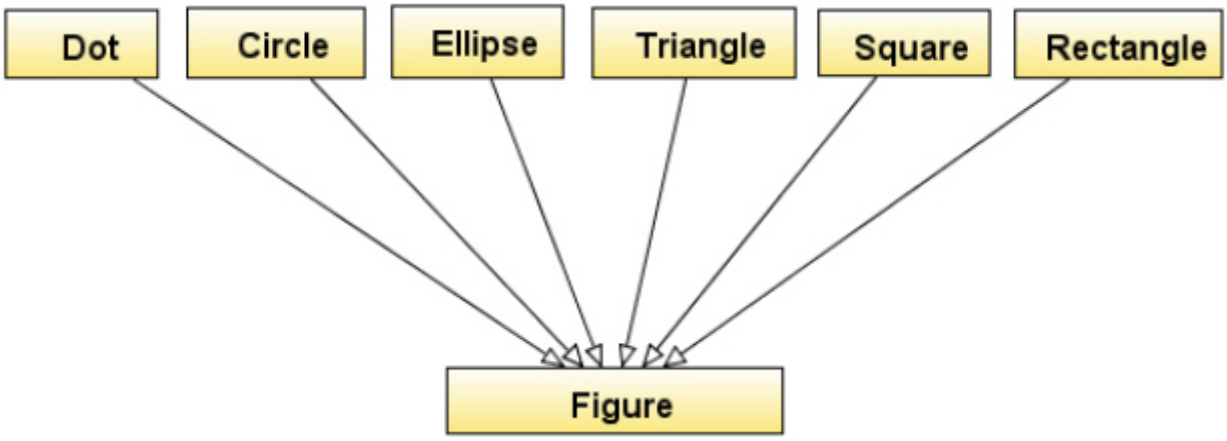


Рис. 6.3. Еще один вариант иерархии фигур

Возможны и другие варианты, ничуть не менее логичные. Какой вариант выбрать?

Уже на этом простейшем примере мы убеждаемся, что проектирование иерархии – очень многовариантная задача. И требуется большой *опыт*, чтобы грамотно построить иерархию. В противном случае при написании кода классов не удается в полной мере обеспечить их функциональность, а код классов становится неуправляемым – внесение исправления в одном месте приводит к возникновению ошибок в совсем других местах. Причем возникает ошибок больше, чем исправляется.

Один из важных принципов при построении таких иерархий – *соответствие представлений из предметной области строящейся иерархии*. В примере, приведенном на первом рисунке, мы имеем вполне логичную с точки зрения идеологии наследования иерархию, показанную на первом рисунке. С точки зрения общности/специализации такая *иерархия* безупречна. *По* этой причине она удобна для написания учебных программ, иллюстрирующих совместимость *объектных типов* и *полиморфизм*. Но в геометрии, из которой мы знаем о свойствах этих фигур, считается, что *окружность* является частным случаем эллипса, а точка – частным случаем окружности (а значит, и эллипса). Так как значения полей данных объекта задают его состояние, в некоторых случаях объекты, являющиеся Эллипсами *по* типу (внутреннему устройству), окажутся в состоянии, когда с точки *предметной области* они будут являться окружностями. Хотя *по* внутреннему устройству и будут отличаться от объектов-Окружностей.

Поэтому данная *иерархия* может вызывать внутренний протест у многих людей. Особенно учитывая сложность различения классов и объектов в обычной речи и при не очень строгих рассуждениях (а можно ли всегда рассуждать абсолютно строго?). Поэтому такое решение может приводить к *логическим ошибкам* в рассуждениях. Вот почему последний из предложенных вариантов иерархий, когда все классы наследуются непосредственно от `Figure`, во многих случаях предпочтителен. Тем более, что никакого выигрыша при написании программного кода увеличение числа поколений наследования не дает: код, написанный для класса `Dot`, вряд ли будет использоваться для объектов классов `Circle` и `Ellipse`. А ведь *наследование* само *по* себе не нужно – это инструмент для написания более экономного полиморфного кода.

Более того, увеличение числа поколений приводит к снижению надежности кода. Так что им не следует злоупотреблять. (Об этом подробнее говорится в одном из параграфов "Наследование: проблемы и альтернативы. Интерфейсы. Композиция").

На выбор варианта иерархии оказывают заметное влияние соображения повторного использования кода – если бы *класс* Ellipse активно использовал часть кода, написанного для класса Circle , а тот, в свою *очередь*, активно пользовался кодом класса Dot , выбор первого варианта мог бы стать предпочтительным *по* сравнению с третьим. Даже несмотря на некоторый *конфликт* с "обыденными" (не принципиальными!) представлениями *предметной области*.

Но имеется одна возможность, которую можно реализовать, попытавшись совместить идеи, возникшие при попытках построить предыдущие варианты нашей иерархии. Мы пришли к выводу, что фигуры могут быть масштабируемы (без изменения формы, оставаясь подобными), а также растягиваемы. Поэтому можно ввести классы ScalableFigure ("масштабируемая фигура") и StretchableFigure ("растягиваемая фигура"). Точка Dot не является ни масштабируемой, ни растягиваемой. Очевидно, что любая растягиваемая фигура должна быть масштабируемая. *Окружность* Circle и квадрат *Square* масштабируемы, но не растягиваемы. А *прямоугольник* Rectangle , *эллипс* Ellipse и треугольник *Triangle* как масштабируемы, так и растягиваемы. Поэтому наша *иерархия* будет выглядеть так:

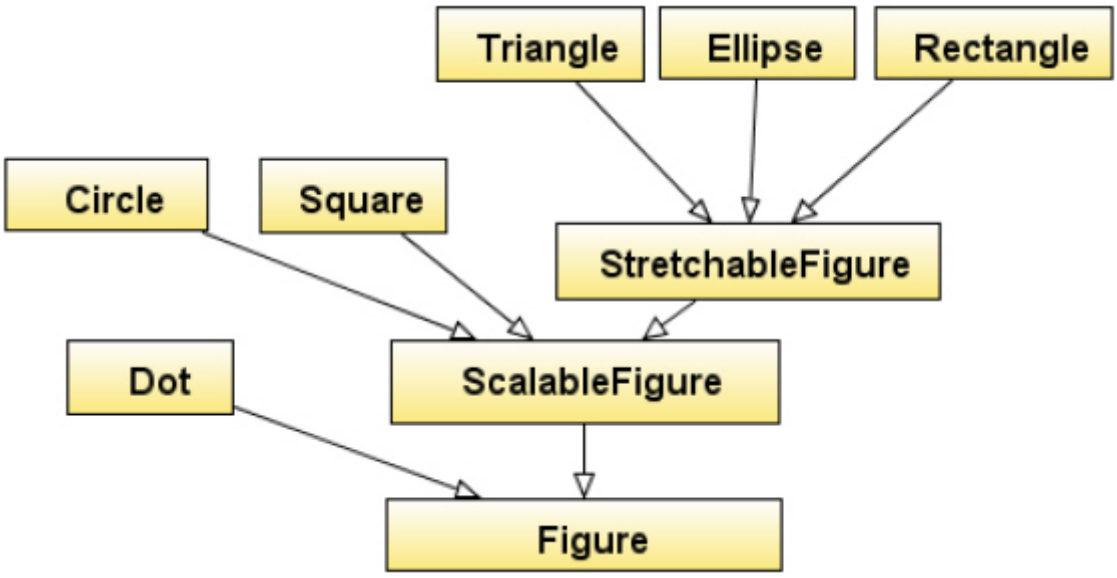


Рис. 6.4. Итоговый вариант иерархии фигур

Основное ее преимущество *по* сравнению с предыдущими – возможность писать полиморфный код для наиболее общих разновидностей фигур. Введение промежуточных уровней наследования, отвечающих соответствующим абстракциям, является характерной чертой объектного программирования. При этом классы *Figure* , ScalableFigure и StretchableFigure будут *абстрактными* – экземпляров такого типа создавать не предполагается. Так как не бывает "фигуры", "масштабируемой фигуры" или "растягиваемой фигуры" в общем виде, без указания ее конкретной формы. Точно так же методы show и hide для этих классов также будут абстрактными.

Еще один важный принцип при построении иерархий на первый взгляд может показаться достаточно странным и противоречащим требованию повторного использования кода. Его можно сформулировать так: *не использовать код неабстрактных классов для наследования*.

Можно заметить, что в приведенной иерархии несколько этапов наследования приходятся именно на *абстрактные классы*, и *ни один из классов, имеющих экземпляры (объекты), не имеет наследников*. Причина такого требования проста: изменение реализации одного класса, проводимое не на уровне абстракции, а относящееся только к одному конкретному классу, не должна влиять на поведение другого класса. Иначе возможны неотслеживаемые труднопонимаемые ошибки в работе *иерархии классов*. Например, если мы попробуем унаследовать *класс* Ellipse от Circle , после исправлений в реализации Circle , обеспечивающих правильную работу объектов этого типа, могут возникнуть проблемы при работе объектов типа Ellipse , которые до того работали правильно. Причем речь идет об особенностях реализации конкретного класса, не относящихся к абстракциям поведения.

Продумывание того, как устроены классы, то есть какие в них должны быть поля и методы (без уточнения об конкретной реализации этих методов), и описание того, какая должна быть *иерархия наследования*, называется *проектированием*. Это сложный процесс, и он обычно гораздо важнее написания конкретных операторов в реализации (*кодирования*).

В языке *Java*, к сожалению, отсутствуют адекватные средства для проектирования классов. Более того, в этом отношении он заметно уступает таким языкам как C++ или *Object PASCAL*, поскольку в *Java*

отсутствует разделение декларации класса (*описание полей* и заголовков методов) и реализации методов. Но в *Sun Java* Studio и NetBeans *Enterprise* Pack имеется средство решения этой проблемы – создание *UML*-диаграмм. *UML* расшифровывается как Universal *Modeling Language* – Универсальный *Язык Моделирования*. Он предназначен для моделирования на уровне абстракций классов и связей их друг с другом – то есть для задач *Объектно-Ориентированного Проектирования* (*OOA – Object- Oriented Architecture*). Приведенные выше рисунки иерархий классов – это *UML*-диаграммы, сделанные с помощью NetBeans *Enterprise* Pack.

Пока в этой среде нет возможности *по UML*-диаграммам создавать заготовки классов *Java*, как это делается в некоторых других средах *UML*-проектирования. Но если создать пустые заготовки классов, то далее можно разрабатывать соответствующие им *UML*-диаграммы, и внесенные изменения на диаграммах будут сразу отображаться в исходном коде. Как это делается будет подробно описано в последнем параграфе данной лекции, где будет обсуждаться технология *Reverse Engineering*.

6.2. Функции. Модификаторы. Передача примитивных типов в функции

Основой создания новых классов является задание полей данных и методов. Но если поля отражают структуру данных, связанных с объектом или классом, то методы задают поведение объектов, а также работу с полями данных объектов и классов.

Формат *объявления функции* следующий:

```
Модификаторы Тип Имя(список параметров){
Тело функции
}
```

Это формат "простой" функции, не содержащей операторов, возбуждающих исключительные ситуации. Про исключительные ситуации и формат *объявления функции*, которая может возбуждать исключительную ситуацию, речь пойдет в одном из следующих параграфов.

Комбинация элементов декларации метода *Модификаторы Тип Имя(список параметров)* называется *заголовком* метода.

Модификаторы – это зарезервированные слова, задающие

- Правила доступа к методу (**private**, **protected**, **public**). Если модификатор не задан, действует доступ по умолчанию – так называемый пакетный.
- Принадлежность к методам класса (**static**). Если модификатор не задан, считается, что это метод объекта.
- Невозможность *переопределения метода* в потомках (**final**). Если модификатор не задан, считается, что это метод можно переопределять в классах-потомках.
- Способ реализации (**native** – заданный во внешней библиотеке DLL, написанной на другом языке программирования; **abstract** – абстрактный, не имеющий реализации). Если модификатор не задан, считается, что это обычный метод.
- Синхронизацию при работе с потоками (**synchronized**) .

В качестве *Типа* следует указать тип результата, возвращаемого методом. В *Java*, как мы уже знаем, все методы являются функциями, возвращающими *значение* какого-либо типа. Если требуется метод, не возвращающий никакого значения (то есть процедура), он объявляется с типом `void` . Возврат значения осуществляется в теле функции с помощью зарезервированного слова `return` .

Для выхода без возврата значения требуется написать

```
return;
```

Для выхода с возвратом значения требуется написать

`return` *выражение* ;

Выражение будет вычислено, после чего полученное *значение* возвратится как результат работы функции.

Оператор `return` осуществляет *прерывание* выполнения подпрограммы, поэтому его обычно используют в ветвях операторов `if-else` или `switch-case` в случаях, когда необходимо возвращать тот или иной результат в зависимости от различных условий. Если в подпрограмме-функции в какой-либо из ветвей не использовать оператор `return` , будет выдана ошибка компиляции с диагностикой "missing *return statement*" – "отсутствие оператора возврата *return*".

Список параметров – это объявление через запятую переменных, с помощью которых можно передавать значения и объекты в подпрограмму снаружи, "из внешнего мира", и передавать объекты из подпрограммы наружу, "во внешний мир".

Объявление параметров имеет вид

тип1 имя1, тип2 имя2,…, типN имяN

Если *список* параметров пуст, пишут круглые скобки без параметров.

Тело функции представляет *последовательность операторов*, реализующую необходимый *алгоритм*. Эта последовательность может быть пустой, в этом случае говорят о *заглушке* – то есть заготовке метода, имеющей только имя и *список* параметров, но с отсутствующей реализацией. Иногда в такой заглушке вместо "правильной" реализации временно пишут *операторы* служебного вывода в консольное окно или в *файл*.

Внутри тела функции в произвольном месте могут задаваться переменные. Они доступны только внутри данной подпрограммы и поэтому называются *локальными*. Переменные, заданные на уровне класса (поля данных класса или объекта), называются *глобальными*.

Данные (значения или объекты) можно передавать в подпрограмму либо через *список* параметров, либо через *глобальные переменные*.

Сначала рассмотрим передачу в подпрограмму через *список* параметров значений примитивного типа. Предположим, что мы написали в классе `MyMath` метод `mult1` умножения двух чисел, каждое из которых перед умножением увеличивается на 1. Он может выглядеть так:

```
double mult1(double x, double y){
 x++;
 y++;
 return x*y;
}
```

Вызов данного метода может выглядеть так:

```
double a,b,c;
…
MyMath obj1=new MyMath();//создали объект типа MyMath
…
c=obj1.mult1(a+0.5,b);
```

Параметры, указанные в заголовке функции при ее декларации, называются *формальными*. А те параметры, которые подставляются во время вызова функции, называются *фактическими*. *Формальные параметры* нужны для того, чтобы указать последовательность действий с *фактическими параметрами* после того, как те будут переданы в подпрограмму во время вызова. Это ни что иное, как особый вид локальных переменных, которые используются для обмена данными с внешним миром.

В нашем случае `x` и `y` являются формальными параметрами, а выражения `a+0.5` и `b` – *фактическими параметрами*. При вызове сначала проводится *вычисление* выражений, переданных в качестве *фактического параметра*, после чего получившийся результат копируется в локальную переменную, используемую в качестве *формального параметра*. То есть в локальную переменную `x` будет скопировано *значение*, получившееся в результате вычисления `a+0.5` , а в локальную переменную `y` – *значение*, хранящееся в переменной `b` . После чего с локальными переменными происходят все те действия, которые указаны в реализации метода. Соответствие фактических и формальных параметров идет в порядке перечисления. То есть первый *фактический параметр* соответствует первому формальному, второй фактический – второму формальному, и так далее. Фактические параметры должны быть совместимы с формальными – при этом действуют все правила, относящиеся к совместимости *примитивных типов по* присваиванию, в том числе – к *автоматическому преобразованию* типов. Например, для `mult1` можно вместо параметров типа `double` в качестве фактических использовать значения типа `int` или `float` . А если бы *формальные параметры* имели тип `float` , то использовать фактические параметры типа `int` было бы можно, а типа `double` – нельзя.

Влияет ли как-нибудь увеличение переменной `y` на 1, происходящее благодаря оператору `y++` , на *значение*, хранящееся в переменной `b` ? Конечно, нет. Ведь действия происходят с локальной переменной `y` , в которую при начале вызова было скопировано *значение* из переменной `b` . С самой переменной `b` в результате вызова ничего не происходит.

А можно ли сделать так, чтобы *подпрограмма* изменяла *значение* в передаваемой в нее переменной? – Нет, нельзя. В *Java* значения примитивного типа наружу, к сожалению, передавать нельзя, в отличие от подавляющего большинства других языков программирования. Применяемый в *Java* способ *передачи параметров* называется передачей *по значению*.

Иногда бывает нужно передать в подпрограмму неизменяемую константу. Конечно, можно проверить, нет ли где-нибудь оператора, изменяющего соответствующую переменную. Но надежней проверка на уровне

синтаксических конструкций. В этих целях используют модификатор `final`. Предположим, что увеличивать на 1 надо только первый *параметр*, а второй должен оставаться неизменным. В этом случае наш метод выглядел бы так:

```
double mult1(double x, final double y){
 x++;
 return x*y;
}
```

А вот при компиляции такого кода будет выдано *сообщение об ошибке*:

```
double mult1(double x, final double y){
 x++;
 y++;
 return x*y;
}
```

6.3. Локальные и глобальные переменные. Модификаторы доступа и правила видимости. Ссылка this

Как уже говорилось, данные в подпрограмму могут передаваться через *глобальные переменные*. Это могут быть поля данных объекта, в методе которого осуществляется вызов, поля данных соответствующего класса, либо поля данных другого объекта или класса. Использование глобальных переменных не рекомендуется *по* двум причинам.

Во-первых, при вызове в списке параметров не видно, что идет обращение к соответствующим переменным, и программа становится "непрозрачной" для программиста. Что делает ее неструктурной.

Во-вторых, при изменении внутри подпрограммы-функции глобальной переменной возникает *побочный эффект*, связанный с тем, что функция не только возвращает вычисленное значение, но и меняет состояние окружения незаметным для программиста образом. Это может являться причиной плохо обнаружимых логических ошибок, не отслеживаемых компилятором.

Конечно, бывают случаи, когда использование глобальных переменных не только желательно, а просто необходимо – иначе их не стали бы вводить как конструкцию языков программирования! Например, при написании метода в каком-либо классе обычно необходимо получать *доступ* к полям и методам этого класса. В *Java* такой *доступ* осуществляется напрямую, без указания имени объекта или класса.

Правила доступа к методам и полям данных (переменным) из других пакетов, классов и объектов задаются с помощью модификаторов **private**, **protected**, **public**. Правила доступа часто называются также правилами видимости, это синонимы. Если *переменная* или *подпрограмма* невидимы в некой области программы, *доступ* к ним запрещен.

private – элемент (*поле* данных или метод) доступен только в методах данного класса. Доступа из объектов нет! То есть если мы создали *объект*, у которого имеется *поле* или метод `private`, то получить *доступ* к этому полю или методу из объекта нельзя.

Модификатор не задан – значит, действует *доступ по* умолчанию – так называемый пакетный, когда соответствующий элемент доступен только из классов своего пакета. Доступа из объектов нет, если они вызываются в операторах, расположенных в классах из других пакетов!

Иногда, *по* аналогии с C++, этот тип доступа называют "дружественным".

protected – элемент доступен только в методах данного класса, данного пакета, а также классах-наследниках (они могут располагаться в других пакетах).

public – элемент доступен из любых классов и объектов (с квалификацией именем пакета, если соответствующий *класс* не импортирован).

Например, в классе

```
class Vis1 {
    private int x=10,y=10;
    int p1=1;
    protected int p2=1;
    public int p3=1;
}
```

заданы переменные `x,y,p1,p2,p3`. Причем `x` и `y` обладают уровнем доступа `private`, `p1` – пакетным, `p2` – `protected`, `p3` – `public`. *Перечисление* однотипных переменных через запятую позволяет использовать для нескольких переменных однократное задание имени типа и модификаторов, без повторений.

Как уже говорилось, локальные переменные можно вводить в любом месте подпрограммы. Их можно использовать в данном методе только после места, где они заданы. Областью существования и видимости локальной переменной является часть программного кода от места объявления переменной до *окончания блока*, в котором она объявлена, обычно – до окончания метода.

А вот переменные, заданные на уровне класса (*глобальные переменные*), создаются при создании объекта для методов объекта, и при первом вызове класса для *переменных класса*. И их можно использовать в методах данного класса как глобальные независимо от того, заданы переменные до метода или после.

Еще одной важной особенностью локальных переменных является время их существования: под них выделяется *память* в момент вызова, а высвобождается сразу после окончания вызова. Рассмотрим функцию, вычисляющую сумму чисел от 1 до `n`:

```
double sum1(int n){
 int i;
 double r=0;
 for(i=1;i<=n;i++){
  r+=i;
 };
 return r;
}
```

Вызов данного метода может выглядеть так:

```
c=obj1.sum1(1000);
```

При этом переменные `i` и `r` существуют только во время вызова `obj1.sum1(1000)`. При следующем аналогичном вызове будет создан, а затем высвобожден из памяти следующий комплект `i` и `r`.

Все сказанное про локальные переменные также относится и к объектным переменным. Но не следует путать переменные и объекты: время жизни объектов гораздо больше. Даже если *объект* создается во время вызова подпрограммы, а после окончания этого вызова *связь* с ним кончается. Уничтожением неиспользуемых объектов занимается сборщик мусора (*garbage collector*). Если же *объект* создан в подпрограмме, и *ссылка* на него передана какой-либо глобальной переменной, он будет существовать после выхода из подпрограммы столько времени, сколько необходимо для работы с ним.

Остановимся на области видимости локальной переменной. Имеются следующие уровни видимости:

- На уровне метода. Переменная видна от места декларации до конца метода.
- На уровне блока. Если переменная задана внутри блока `{…}`, она видна от места декларации до конца блока. Блоки могут быть вложены один в другой с произвольным уровнем вложенности.
- На уровне цикла `for`. Переменная видна от места декларации в *секции инициализации* до конца *тела цикла*.

Глобальные переменные видны во всей подпрограмме.

Каждый *объект* имеет *поле* данных с именем `this` ("этот" – данное не слишком удачное обозначение унаследовано из C++), в котором хранится *ссылка* на сам этот *объект*. Поэтому *доступ* в методе объекта к полям и методам этого объекта может осуществляться либо напрямую, либо через ссылку `this` на этот *объект*. Например, если у объекта имеется *поле* `x` и метод `show()`, то `this.x` означает то же, что `x`, а `this.show()` – то же, `show()`. Но в случае перекрытия области видимости, о чем речь пойдет чуть ниже, *доступ по* короткому имени оказывается невозможен, и приходится использовать *доступ по* ссылке `this`. Отметим, что *ссылка* `this` позволяет обойтись без использования имени *объектной переменной*, что делает код с ее использованием более универсальным. Например, использовать в методах того класса, экземпляром которого является *объект*.

Ссылка `this` не может быть использована в методах класса (то есть заданных с модификатором `static`), поскольку они могут вызываться без существующего объекта.

Встает вопрос о том, что произойдет, если на разных уровнях будет задано две переменных с одним именем. Имеется следующие варианты ситуаций:

- В классе имеется поле с некоторым именем (глобальная переменная), и в списке параметров задается локальная переменная с тем же именем. Такая проблема часто возникает в конструкторах при

инициализации полей данных и в методах установки значений полей данных setИмяПоля. Это разрешено, и доступ к параметру идет по имени, как обычно. Но при этом видимость поля данных (доступ к полю данных по его имени) перекрывается, и приходится использовать ссылку на объект `this`. Например, если имя поля данных `x`, и имя параметра в методе тоже `x`, установка значения поля выглядит так:

```
void setX(double x){
  this.x=x
}
```

В классе имеется поле с некоторым именем (глобальная переменная), и в методе задается локальная переменная с тем же именем. Ситуация разрешена и аналогична заданию локальной переменной в списке параметров. Доступ к полю идет через ссылку `this`.

В классе имеется поле с некоторым именем (глобальная переменная), и в *секции инициализации* цикла `for` или внутри какого-нибудь блока, ограниченного фигурными скобками `{…}`, задается локальная переменная с тем же именем. В Java такая ситуация разрешена. При этом внутри цикла или блока доступна заданная в нем локальная переменная, а глобальная переменная видна через ссылку `this`.

Имеется локальная переменная (возможно, заданная как элемент списка параметров), и в *секции инициализации* цикла `for` или внутри какого-нибудь блока, ограниченного фигурными скобками `{…}`, задается локальная переменная с тем же именем. В Java такая ситуация запрещена. При этом выдается ошибка компиляции с информацией, что переменная с таким именем уже задана ("is already defined").

Имеется метод, заданный в классе, и в другом методе задается локальная переменная с тем же именем. В Java такая ситуация разрешена и не вызывает проблем, так как компилятор отличает вызов метода от обращения к полю данных по наличию после имени метода круглых скобок.

6.4. Передача ссылочных типов в функции. Проблема изменения ссылки внутри подпрограммы

При передаче в подпрограмму ссылочной переменной возникает ряд отличий *по* сравнению со случаем *примитивных типов*, так как в локальную переменную, с которой идет работа в подпрограмме, копируется не сам *объект*, а его *адрес*. Поэтому глобальная *переменная*, ссылающаяся на тот же *объект*, будет получать *доступ* к тем же самым полям данных, что и локальная. В результате чего изменение полей данных объекта внутри метода приведет к тому, что мы увидим эти изменения после выхода из метода (причем неважно, будем мы менять поля непосредственно или с помощью вызова каких-либо методов).

Для примера создадим в нашем пакете *класс* `Location`. Он будет служить для задания объекта соответствующего типа, который будет передаваться через *список* параметров в метод `m1`, вызываемый из нашего приложения.

```
public class Location {
     public int x=0,y=0;
     public Location (int x, int y) {
       this.x=x;
       this.y=y;
     }
}
```

А в *классе приложения* напишем следующий код:

```
Location locat1=new Location(10,20);


 public static void m1(Location obj){
     obj.x++;
     obj.y++;
 }
```

Мы задали переменную `locat1` типа `Location`, инициализировав ее поля `x` и `y` значениями 10 и 20. А в методе `m1` происходит увеличение на 1 значения полей `x` и `y` объекта, связанного с *формальным параметром* `obj`.

Создадим две кнопки с обработчиками событий. Нажатие на первую кнопку будет приводить к выводу информации о значениях полей `x` и `y` объекта, связанного с переменной `locat1`. А нажатие на вторую – к вызову метода `m1`.

```
 private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
  System.out.println("locat1.x="+locat1.x);
  System.out.println("locat1.y="+locat1.y);
 }

 private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
  m1(locat1);
  System.out.println("Прошел вызов m1(locat1)");
 }
```

Легко проверить, что вызов `m1(locat1)` приводит к увеличению значений полей `locat1.x` и `locat1.y`.

При передаче в подпрограмму ссылочной переменной имеется особенность, которая часто приводит к ошибкам – потеря связи с первоначальным объектом при изменении ссылки. Модифицируем наш метод `m1`:

```
 public static void m1(Location obj){
     obj.x++;
     obj.y++;
     obj=new Location(4,4);
     obj.x++;
     obj.y++;
 }
```

После первых двух строк, которые приводили к инкременту полей передаваемого объекта, появилось создание нового объекта и перещелкивание на него локальной переменной `obj`, а затем две точно такие же строчки, как в начале метода. Какие значения полей `x` и `y` объекта, связанного с переменной `locat1` покажет нажатие на кнопку 1 после вызова модифицированного варианта метода? Первоначальный и модифицированный вариант метода дадут одинаковые результаты!

Дело в том, что *присваивание* `obj=new Location(4,4);` приводит к тому, что *переменная* `obj` становится связанной с новым, только что созданным объектом. И изменение полей данных в операторах `obj.x++` и `obj.y++` происходит уже для этого объекта. А вовсе не для того объекта, ссылку на который передали через *список* параметров.

Следует обратить внимание на то, какая терминология используется для описания программы. Говорится "ссылочная *переменная*" и "*объект*, связанный со ссылочной переменной". Эти понятия не отождествляются, как часто делают программисты при описании программы. И именно строгая терминология позволяет разобраться в происходящем. Иначе трудно понять, почему оператор `obj.x++` в одном месте метода дает совсем не тот эффект, что в другом месте. Поскольку если бы мы сказали "изменение поля x объекта *obj*", было бы невозможно понять, что объекты-то разные! А правильная фраза "изменение поля `x` объекта, связанного со ссылочной переменной `obj` " подталкивает к мысли, что эти объекты в разных местах программы могут быть разными.

Способ передачи данных (ячейки памяти) в подпрограмму, позволяющий изменять содержимое внешней ячейки памяти благодаря использованию ссылки на эту ячейку, называется передачей *по ссылке*. И хотя в *Java объект* передается *по* ссылке, *объектная переменная*, в которой хранится *адрес* объекта, передается *по* значению. Ведь этот *адрес* копируется в другую ячейку, локальную переменную. А именно *переменная* является параметром, а не связанный с ней *объект*. То есть *параметры в Java всегда передаются по значению*. *Передачи параметров по* ссылке в языке *Java* нет.

Рассмотрим теперь нетривиальные ситуации, которые часто возникают при передаче ссылочных переменных в качестве параметров.

Мы уже упоминали о проблемах, возникающих при работе со строками. Рассмотрим подпрограмму, которая, *по* идее, должна бы возвращать с помощью переменной `s3` сумму строк, хранящихся в переменных `s1` и `s2`:

```
 void strAdd1(String s1,s2,s3){
  s3=s1+s2;
 }
```

Строки в *Java* являются объектами, и строковые переменные являются ссылочными. Поэтому можно было бы предполагать возврат измененного состояния строкового объекта, с которым связана *переменная* `s3`. Но

все обстоит совсем не так: при вызове

```
obj1.strAdd1(t1,t2,t3);
```

значение строковой переменной `t3` *не изменится*. Дело в том, что в *Java* строки типа `String` являются неизменяемыми объектами, и вместо изменения состояния прежнего объекта в результате вычисления выражения `s1+s2` создается новый *объект*. Поэтому *присваивание* `s3=s1+s2` приводит к перещелкиванию ссылки `s3` на этот новый *объект*. А мы уже знаем, что это ведет к тому, что новый *объект* оказывается недоступен вне подпрограммы – "внешняя" *переменная* `t3` будет ссылаться на прежний *объект*-строку. В данном случае, конечно, лучше сделать функцию *strAdd1* строковой, и возвращать получившийся *строковый объект* как результат вычисления этой функции.

Еще пример: пусть нам необходимо внутри подпрограммы обработать некоторую строку и вернуть измененное *значение*. Допустим, в качестве *входного параметра* передается имя, и мы хотим добавить в конец этого имени порядковый номер – примерно так, как это делает *среда разработки* при создании нового компонента. Следует отметить, что для этих целей имеет смысл создавать подпрограмму, хотя на первый взгляд достаточно выражения name+*count*. Ведь на следующем этапе мы можем захотеть проверить, является ли входное *значение* идентификатором (начинающимся с буквы и содержащее только буквы и цифры). Либо проверить, нет ли уже в списке имен такого имени.

Напишем в классе нашего приложения такой код:

```
String componentName="myComponent";
int count=0;
public void calcName1(String name) {
  count++;
  name+=count;
  System.out.println("Новое значение="+name);
}
```

Создадим в нашем приложении кнопку, при нажатии на которую срабатывает следующий обработчик события:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
 calcName1(componentName);
 System.out.println("componentName="+componentName);
}
```

Многие начинающие программисты считают, что раз строки являются объектами, то при первом нажатии на кнопку *значение* `componentName` станет `"myComponent1"`, при втором – `"myComponent2"`, и так далее. Но *значение* `myComponent` остается неизменным, хотя в методе `calcName1` новое *значение* выводится именно таким, как надо. В чем причина такого поведения программы, и каким образом добиться правильного результата?

Если мы меняем в подпрограмме *значение* полей у объекта, а *ссылка* на *объект* не меняется, то изменение значения полей оказывается наблюдаемым с помощью доступа к тому же объекту через внешнюю переменную. А вот *присваивание* строковой переменной внутри подпрограммы нового значения приводит к созданию нового объекта-строки и перещелкивания на него ссылки, хранящейся в локальной переменной name. Причем глобальная *переменная* `componentName` остается связанной с первоначальным объектом-строкой `"myComponent"`.

Как бороться с данной проблемой? Существует несколько вариантов решения.

Во-первых, в данном случае наиболее разумно вместо подпрограммы-процедуры, не возвращающей никакого значения, написать подпрограмму-функцию, возвращающую *значение* типа `String`:

```
public String calcName2(String name) {
  count++;
  name+=count;
  return name;
}
```

В этом случае не возникает никаких проблем с возвратом значения, и следующий обработчик нажатия на кнопку это демонстрирует:

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
 componentName=calcName2(componentName);
 System.out.println("componentName="+componentName);
}
```

К сожалению, если требуется возвращать более одного значения, данный способ решения проблемы не подходит. А ведь часто из подпрограммы требуется возвращать два или более измененных или вычисленных значения.

Во-вторых, можно воспользоваться глобальной строковой переменной – но это плохой стиль программирования. Даже использование глобальной переменной `count` в предыдущем примере не очень хорошо – но мы это сделали для того, чтобы не *усложнять пример*.

В-третьих, возможно создание оболочечного объекта (*wrapper*), у которого имеется *поле строкового типа*. Такой *объект* передается *по* ссылке в подпрограмму, и у него внутри подпрограммы меняется *значение* строкового поля. При этом, конечно, это *поле* будет ссылаться на новый *объект*-строку. Но так как *ссылка* на оболочечный *объект* внутри подпрограммы не меняется, *связь* с новой строкой через оболочечный *объект* сохранится и снаружи. Такой подход, в отличие от использования подпрограммы-функции *строкового типа*, позволяет возвращать *произвольное* количество значений одновременно, причем произвольного типа, а не только строкового. Но у него имеется недостаток – требуется создавать специальные классы для формирования возвращаемых объектов.

В-четвертых, имеется возможность использовать классы `StringBuffer` или `StringBuilder` . Это наиболее адекватный способ при необходимости возврата более чем одного значения, поскольку в этой ситуации является и самым простым, и весьма эффективным *по* быстродействию и используемым ресурсам. Рассмотрим соответствующий код.

```
public void calcName3(StringBuffer name) {
    count++;
    name.append(count);
    System.out.println("Новое значение="+name);
}

StringBuffer sbComponentName=new StringBuffer();
{sbComponentName.append("myComponent");}

private void jButton8ActionPerformed(java.awt.event.ActionEvent evt){
    calcName3(sbComponentName);
    System.out.println("sbComponentName="+sbComponentName);
}
```

Вместо строкового поля `componentName` мы теперь используем *поле* `sbComponentName` типа `StringBuffer` . Почему-то разработчики этого класса не догадались сделать в нем *конструктор* с параметром *строкового типа*, поэтому приходится использовать блок инициализации, в котором переменной `sbComponentName` присваивается нетривиальное начальное *значение*. В остальном код очевиден. Принципиальное отличие от использования переменной типа `String` – то, что изменение значения строки, хранящейся в переменной `StringBuffer` , не приводит к созданию нового объекта, связанного с этой переменной.

Вообще говоря, с этой точки зрения для работы со строками переменные типа `StringBuffer` и `StringBuilder` подходят гораздо лучше, чем переменные типа `String` . Но метода `toStringBuffer()` в классах не предусмотрено. Поэтому при использовании переменных типа `StringBuffer` обычно приходится пользоваться конструкциями вида `sb.append` (*выражение*). В методы `append` и `insert` можно передавать выражения произвольных примитивных или *объектных типов*. Правда, массивы преобразуются в строку весьма своеобразно, так что для их преобразования следует писать собственные подпрограммы. Например, при выполнении фрагмента

```
int[] a=new int[]{10,11,12};
System.out.println("a="+a);
```

был получен следующий результат:

```
a=[I@15fea60
```

И выводимое *значение* не зависело ни от значений элементов массива, ни от их числа.

Наличие автоматической упаковки-распаковки также приводит к проблемам. Пусть у нас имеется случай, когда в списке параметров указана *объектная переменная*:

```
void m1(Double d){
 d++;
}
```

Несмотря на то, что *переменная* `d` объектная, изменение значения `d` внутри подпрограммы не приведет к изменению снаружи подпрограммы *по* той же причине, что и для переменных типа `String`. При инкременте сначала производится распаковка в тип `double`, для которого выполняется оператор `"++"`. После чего выполняется упаковка в *новый объект* типа `Double`, с которым становится связана *переменная* `d`.

Приведем еще один аналогичный пример:

```
public void proc1(Double d1,Double d2,Double d3){
    d3=d1+sin(d2);
}
```

Надежда на то, что в *объект*, передаваемый через *параметр* `d3`, возвратится вычисленное *значение* `d3=d1+sin(d2)`, является ошибочной, так как при упаковке вычисленного результата создается новый *объект*.

Таким образом, объекты стандартных оболочечных числовых классов не позволяют возвращать измененное числовое *значение* из подпрограмм, что во многих случаях вызывает проблемы. Для этих целей приходится писать собственные оболочечные классы. Например:

```
public class UsableDouble{
    Double value=0;
    UsableDouble(Double value){
       this.value=value;
    }
}
```

Объект `UsableDouble d` можно передавать в подпрограмму *по* ссылке и без проблем получать возвращенное измененное *значение*. Аналогичного рода оболочные классы легко написать для всех *примитивных типов*.

Если бы в стандартных оболочечных классах были методы, позволяющие изменить числовое *значение*, связанное с объектом, без изменения адреса объекта, в такого рода деятельности не было бы необходимости.

Заканчивая разговор о проблемах *передачи параметров* в подпрограмму, *автор* хочет выразить надежду, что разработчики *Java* либо добавят в стандартные оболочечные классы такого рода методы, либо добавят возможность передачи переменных в подпрограммы *по* ссылке, как, к примеру, это было сделано в *Java*-образном языке C#.

6.5. Наследование. Суперклассы и подклассы. Переопределение методов

В объектном программировании принято использовать имеющиеся классы в качестве "заготовок" для создания новых классов, которые на них похожи, но обладают более сложной структурой и/или отличающимся поведением. Такие "заготовки" называются *прародителями* (*ancestors*), а основанные на них новые классы - *потомками* (*descendants*) или наследниками. Классы-потомки получают "в пользование" поля и методы, заданные в классах-прародителях, это называется *наследованием* (*inheritance*) полей и методов.

В C++ и *Java* вместо терминов "прародители" и "потомки" чаще используют неудачные названия "*суперклассы*" (*superclasses*) и "*подклассы*" (*subclasses*). Как уже говорилось, *суперклассы* должны быть примитивнее *подклассов*, но приставка "супер" подталкивает программиста к прямо противоположным действиям.

При задании класса-потомка сначала идут модификаторы, затем после ключевого слова `class` идет имя декларируемого класса, затем идет *зарезервированное слово* `extends` ("расширяет"), после чего требуется указать *имя класса*-родителя (непосредственного прародителя). Если не указывается, от какого класса идет *наследование*, родителем считается *класс* `Object`. Сам *класс-потомок* называется наследником, или дочерним.

В синтаксисе *Java* словом `extends` подчеркивается, что *потомок* расширяет то, что задано в прародителе – добавляет новые поля, методы, усложняет поведение. (Но все это делает *класс* более специализированным, менее общим).

Далее в фигурных скобках идет реализация класса – описание его полей и методов. При этом поля данных и методы, имеющиеся в прародителе, в потомке описывать не надо – они наследуются. Однако в случае, если реализация прародительского метода нас не устраивает, в классе-потомке его можно реализовать *по* другому. В этом случае метод необходимо продекларировать и реализовать в классе-потомке. Кроме того, в потомке можно задавать новые поля данных и методы, отсутствующие в прародителях.

Модификаторы, которые можно использовать:

public – модификатор, задающий публичный (общедоступный) уровень видимости. Если он отсутствует, действует пакетный уровень доступа – класс доступен только элементам того же пакета.

abstract – модификатор, указывающий, что класс является абстрактным, то есть у него не бывает экземпляров (объектов). Обязательно объявлять класс абстрактным в случае, если какой-либо метод объявлен как абстрактный.

final – модификатор, указывающий, что класс является окончательным (`final`) , то есть что у него не может быть потомков.

Таким образом, задание класса-наследника имеет следующий формат:

```
Модификаторы class ИмяКласса extends ИмяРодителя {
 Задание полей;
 Задание подпрограмм - методов класса, методов объекта, конструкторов
}
```

Данный формат относится к классам, не реализующим интерфейсы (`interfaces`). Работе с интерфейсами будет посвящен отдельный раздел.

Рассмотрим в качестве примера *наследование* для классов описанной ранее иерархии фигур. Для простоты выберем вариант, в котором `Figure` – это *класс*-прародитель иерархии, `Dot` его *потомок*, а `Circle` – *потомок* `Dot` (то есть является "жирной точкой"). Напомним, что имена классов принято начинать с заглавной буквы.

Класс `Figure` опишем как абстрактный – объектов такого типа создавать не предполагается, так как фигура без указания конкретного вида – это, действительно, чистая *абстракция*. *По* той же причине методы `show` ("показать") и `hide` ("скрыть") объявлены как абстрактные. Напомним также, что если в классе хоть один метод является абстрактным, это *класс* обязан быть объявлен как абстрактный.

```
public abstract class Figure { //это абстрактный класс

    int x=0;
    int y=0;
    java.awt.Color color;
    java.awt.Graphics graphics;
    java.awt.Color bgColor;
    public abstract void show(); //это абстрактный метод
    public abstract void hide(); //это абстрактный метод

    public void moveTo(int x, int y){
        hide();
        this.x= x;
        this.y= y;
        show();
    };
}
```

Поля `x` и `y` задают *координаты* фигуры, а `color` – ее цвет. Соответствующий тип задан в пакете `java.awt` . *Поле* `graphics` задает ссылку на графическую поверхность, *по* которой будет идти отрисовка фигуры. Соответствующий тип также задан в пакете `java.awt` . В отличии от полей `x` , `y` и `color` для этого поля при написании класса невозможно задать начальное *значение*, и оно будет присвоено при создании объекта. То же относится к полю `bgColor` (от "*background color*") – в нем мы

будем хранить ссылку на *цвет фона* графической поверхности. Цветом фона мы будем выводить фигуру в методе `hide` для того, чтобы она перестала показываться на экране. Это не самый лучший, но зато самый простой способ скрыть фигуру. В дальнейшем при желании реализацию метода можно изменить – это никак не коснется остальных частей программы. В параграфе, посвященном конструкторам, в классе `FilledCircle` мы применим более совершенный способ отрисовки и "скрывания" фигур, основанный на использовании режима рисования `XOR` ("исключающее или"). Установка этого режима производится методом `setXORMode`. Такой режим можно использовать для всех наших фигур.

Метод `moveTo` имеет реализацию несмотря на то, что *класс* абстрактный, и в этой реализации используются имена *абстрактных методов* `show` и `hide`. Этот вопрос будет подробно обсуждаться в следующем параграфе, посвященном полиморфизму.

Рассмотрим теперь, как задается *потомок* класса `Figure` – *класс* `Dot` ("Точка"). Для `Dot` классы `Object` и `Figure` будут являться прародителями (*суперклассами*), причем `Figure` будет непосредственным прародителем. Соответственно, для них *класс* `Dot` будет являться потомком (*подклассом*), причем для класса `Figure` – непосредственным потомком. *Класс* `Dot` расширяет (extends) функциональность класса `Figure`: хотя в нем и не появляется новых полей, зато пишется реализация для методов `show` и `hide`, которые в прародительском классе были абстрактными. В классе `Figure` мы использовали классы пакета `java.awt` без импорта этого пакета. В классе `Dot` используется импорт – обычно это удобнее, так как не надо много раз писать длинные имена.

```
package java_gui_example;

import java.awt.*;
/**
 * @author В.В.Монахов
 */
public class Dot extends Figure{

    /** Создает новый экземпляр типа Dot */
    public Dot(Graphics graphics,Color bgColor) {
        this.graphics=graphics;
        this.bgColor=bgColor;
    }

    public void show(){
       Color oldC=graphics.getColor();
       graphics.setColor(Color.BLACK);
       graphics.drawLine(x,y,x,y);
       graphics.setColor(oldC);
    }
    public void hide(){
       Color oldC=graphics.getColor();
       graphics.setColor(bgColor);
       graphics.drawLine(x,y,x,y);
       graphics.setColor(oldC);
        ;
    }
}
```

Отметим, что в классе `Dot` не задаются поля `x`, `y`, `graphics` и метод `moveTo` – они наследуются из класса `Figure`. А методы `show` и `hide` *переопределяются* (*override*) – для них пишется реализация, соответствующая тому, каким именно образом точка появляется и скрывается на экране.

Конструктор `Dot(Graphics graphics, Color bgColor)` занимается созданием объекта типа `Dot` и инициализацией его полей. В методах `show` и `hide` используются методы объекта `graphics`. В методе `show` сначала во временной переменной `oldC` сохраняется *информация* о текущем цвете рисования. Затем в качестве текущего цвета устанавливается черный цвет (константа `java.awt. Color.BLACK`). Затем вызывается метод, рисующий точку, в качестве него используется рисование линии с совпадающими началом

и концом. После чего восстанавливается первоначальный цвет рисования. Это необходимо для того, чтобы не повлиять на поведение других объектов, пользующихся для каких-либо целей текущим цветом. Такого рода действия являются очень характерными при пользовании *разделяемыми* (shared) ресурсами. Если вам при работе какого-либо метода требуется изменить состояние разделяемых внешних данных, сначала требуется сохранить информацию о текущем состоянии, а в конце вызова восстановить это состояние.

Термин **override** ("переопределить") на русский язык часто переводят как "перекрыть". Это может вводить в заблуждение, так как имеется еще одно понятие – перекрытие области видимости (hiding). Такое перекрытие возникает в случае, когда в классе-потомке задается *поле* с тем же именем, что и в прародителе (но, возможно, другого типа). Для методов совпадение имен разрешено, в том числе с именами глобальных и локальных переменных.

Имя метода в сочетании с числом параметров и их типами называется его *сигнатурой*. А *сигнатура метода* в сочетании с типом возвращаемого значения называется *контрактом* метода. В контракт также входят типы возбуждаемых методом исключений, но о соответствующих правилах будет говориться в отдельном параграфе, посвященном обработке исключительных ситуаций.

Если контракт задаваемого метода совпадает с контрактом прародительского метода, говорят, что метод переопределен. Если у двух методов имена совпадают, но сигнатуры различны – говорят, что производится *перегрузка* (*overloading*) методов. *Перегрузке методов* далее будет посвящен отдельный *параграф*. Если же в одном классе два метода имеют одинаковые сигнатуры, то даже если их контракты отличаются, *компилятор* выдает *сообщение об ошибке*.

В классе нашего приложения создадим на *экранной форме* панель, и будем вести отрисовку *по* ней. Зададим с помощью редактора свойств белый (или какой-нибудь другой) цвет панели – свойство background . Затем зададим переменную dot , которой назначим *объект* в обработчике нажатия на кнопку:

```
Dot dot=new Dot(jPanel1.getGraphics(),jPanel1.getBackground());
```

После создания объекта-точки с помощью переменной dot можно вызывать методы show и hide :

```
dot.show();
dot.hide();
```

Создадим на форме пункты ввода/редактирования текста jTextField1 и jTextField2 . В этом случае становится можно вызывать метод moveTo , следующим образом задавая *координаты*, куда должна перемещаться точка:

```
int newX=Integer.parseInt(jTextField1.getText());
int newY=Integer.parseInt(jTextField2.getText());
dot.moveTo(newX,newY);
```

Наш пример оказывается достаточно функциональным для того, чтобы увидеть работу с простейшим объектом.

Рассмотрим теперь *класс* ScalableFigure ("Масштабируемая фигура"), *расширяющий класс* Figure . Он очень прост.

```
package java_gui_example;
public abstract class ScalableFigure extends Figure{
    int size;

    public void resize(int size) {
        hide();
        this.size=size;
        show();
    }
}
```

Класс ScalableFigure является абстрактным – объектов такого типа создавать не предполагается, так как масштабируемая фигура без указания конкретного вида – это *абстракция*. *По* этой же причине в классе не заданы реализации методов show и hide .

Зато появилось *поле* size ("размер"), и метод resize ("изменить размер"), расширяющий этот *класс по* сравнению с прародителем. Для того, чтобы изменить размер фигуры, отрисовываемой на экране, надо не только присвоить полю *size* новое *значение*, но и правильно перерисовать фигуру. Сначала надо ее

скрыть, затем изменить *значение* `size`, после чего показать на экране – уже нового размера. Следует обратить внимание, что мы пишем данный код на уровне абстракций, для нас не имеет значения, какого типа будет фигура – главное, чтобы она была масштабируемая, то есть являлась экземпляром класса-потомка `ScalableFigure`. О механизме, позволяющем такому коду правильно работать, будет рассказано далее в параграфе, посвященном полиморфизму.

Опишем *класс* `Circle` ("*Окружность*"), *расширяющий класс* `ScalableFigure`.

```
package java_gui_example;
import java.awt.*;

public class Circle extends ScalableFigure {

    Circle(Graphics g,Color bgColor, int r){ //это конструктор
        graphics=g;
        this.bgColor=bgColor;
        size=r;
    }

    public void show(){
        Color oldC=graphics.getColor();
        graphics.setColor(Color.BLACK);
        graphics.drawOval(x,y,size,size);
        graphics.setColor(oldC);
    }

    public void hide(){
        Color oldC=graphics.getColor();
        graphics.setColor(bgColor);
        graphics.drawOval(x,y,size,size);
        graphics.setColor(oldC);
    }
};
```

В классе `Circle` не задается новых полей – в качестве радиуса окружности используется *поле* `size`, унаследованное от класса `ScalableFigure`. Зато введен *конструктор*, позволяющий задавать *радиус* при создании окружности.

Кроме того, написаны новые реализации для методов `show` и `hide`, поскольку *окружность* показывается, скрывается и движется *по* экрану не так, как точка.

Таким образом, усложнение структуры `Circle` *по* сравнением со `ScalableFigure` в основном связано с появлением реализации у методов, которые до этого были абстрактными. Очевидно, *класс* `Circle` является более специализированным *по* сравнению со `ScalableFigure`, не говоря уж о `Figure`.

Поля `x`, `y`, `color`, `bgColor`, `graphics` и метод `moveTo` наследуется в `Circle` из класса `Figure`. А из `ScalableFigure` наследуются *поле* `size` и метод `resize`.

Следует особо подчеркнуть, что *наследование* относится к классам, а не к объектам. Можно говорить, что один *класс* является наследником другого. Но категорически нельзя – что один *объект* является наследником другого объекта. Иногда говорят фразы вроде "*объект* `circle` является наследником `Figure`". Это не страшно, если подразумевается, что "*объект* `circle` является экземпляром класса-наследника `Figure`". Слишком долго произносить правильную фразу. Но следует четко понимать, что имеется в виду, и злоупотреблять такими оборотами не следует. *Класс Circle* является непосредственным (прямым) потомком `ScalableFigure,` а `ScalableFigure` – непосредственным (прямым) прародителем класса `Circle`. То есть для `ScalableFigure` *класс* `Circle` является *подклассом*, а для `Circle` *класс* `ScalableFigure` является *суперклассом*. Аналогично, для `Figure` *подклассами* являются и `ScalableFigure`, и `Circle`. А для `Circle` *суперклассами* являются и `ScalableFigure`, и `Figure`.

Поскольку в *Java* все классы – потомки класса `Object`, то `Object` является прародителем и для `Figure`, и для `ScalableFigure`, и для `Circle`. Но непосредственным прародителем он будет только

для *Figure* .

6.6. Наследование и правила видимости. Зарезервированное слово super

В данном параграфе рассматривается ряд нетривиальных ситуаций, связанных с правилами видимости при наследовании.

Поля и методы, помеченные как `private` ("закрытый, частный") наследуются, но в классах-наследниках недоступны. Это сделано в целях обеспечения безопасности. Пусть, например, некий *класс* `Password1` обеспечивает проверку правильности пароля, и у него имеется строковое *поле* `password` ("*пароль*"), в котором держится *пароль* и с которым сравнивается введенный пользователем *пароль*. Если оно имеет тип `public` , такое *поле* общедоступно, и сохранить его в тайне мы не сможем. При отсутствии *модификатора видимости* или модификаторе `protected` на первый взгляд имеется необходимое *ограничение доступа*. Но если мы напишем *класс* `Password2` , являющийся наследником от `Password1` , в нем легко написать метод, "вскрывающий" *пароль*:

```
public String getPass(){
  return password;
};
```

Если же поставить модификатор `private` , то в потомке до прародительского поля `password` не добраться!

То, что `private` -поля наследуются, проверить достаточно просто: зададим *класс*

```
public class TestPrivate1 {
  private String s="Значение поля private";

  public String get_s(){
    return s;
  }
}
```

и его *потомок*, который просто имеет другое имя, но больше ничего не делает:

```
public class TestPrivate2 extends TestPrivate1 {
}
```

Если из объекта, являющегося экземпляром `TestPrivate2` , вызвать метод `get_s()` , мы получим строку `="Значение поля private"` :

```
TestPrivate2 tst=new TestPrivate2();
System.out.println(tst.get_s());
```

Таким образом, *поле* `s` наследуется. Но если в классе, где оно задано, не предусмотрен *доступ* к нему с помощью каких-либо методов, доступных в наследнике, извлечь информацию из этого поля оказывается невозможным.

Модификатор `protected` предназначен для использования соответствующих полей и методов разработчиками классов-наследников. Он дает несколько большую открытость, чем пакетный вид доступа (*по* умолчанию, без модификатора), поскольку в дополнении к видимости из текущего пакета позволяет обеспечить *доступ* к таким членам в классах-наследниках, находящихся в других пакетах. Модификатором `protected` полезно помечать различного рода служебные методы, ненужные пользователям класса, но необходимые для функциональности этого класса.

Существует "правило хорошего тона": поля данных принято помечать модификатором `private` , а *доступ* к этим полям обеспечивать с помощью методов с тем же именем, но префиксом `get` ("получить" - *доступ по* чтению) и `set` ("установить" - *доступ по* записи). Эти методы называют "геттерами" и "сеттерами". Такие правила основаны на том, что *прямой доступ по* записи к полям данных может разрушить *целостность* объекта.

Рассмотрим следующий пример: пусть у нас имеется фигура, отрисовываемая на экране. Изменение ее координат должно сопровождаться отрисовкой на новом месте. Но если мы напрямую изменили *поле* `x` или `y` , фигура останется на прежнем месте, хотя поля имеют новые значения! Если же *доступ* к полю осуществляется через методы `setX` и `setY` , кроме изменения значений полей будут вызваны необходимые методы, обеспечивающие перерисовку фигуры в новом месте. Также можно обеспечить проверку вводимых значений на допустимость.

Возможен и гораздо худший случай доступа к полям напрямую: пусть у нас имеется *объект-прямоугольник*, у которого заданы поля `x1,y1` – *координаты* левого верхнего угла, `x2,y2` – *координаты* правого нижнего угла, `w` – ширина, `h` – *высота*, `s` – *площадь* прямоугольника. Они не являются независимыми: `w=x2-x1, h=y2-y1, s=w*h`. Поэтому изменение какого-либо из этих полей должно приводить к изменению других. Если же, скажем, изменить только `x2`, без изменения `w` и `s`, части объекта станут несогласованными. Предсказать, как поведет себя в таких случаях *программа*, окажется невозможно!

Еще хуже обстоит дело при наличии наследования в тех случаях, когда в потомке задано *поле* с тем же именем, что и в прародителе, имеющее совместимый с прародительским полем тип. Так как для полей данных *полиморфизм* не работает, возможны очень неприятные ошибки.

Указанные выше правила хорошего тона программирования нашли *выражение* в среде NetBeans при установленном пакете NetBeans *Enterprise* Pack. В ней при разработке *UML*-диаграмм добавление в *класс* поля автоматически приводит к установке ему модификатора `private` и созданию двух *public*-методов с тем же именем, но префиксами `get` и `set`. Эти типы видимости в дальнейшем, конечно, можно менять, как и удалять ненужные методы.

Иногда возникает необходимость вызвать *поле* или метод из прародительского класса. Обычно это бывает в случаях, когда в классе-потомке задано *поле* с таким же именем (но, обычно, другим типом) или *переопределен метод*. В результате видимость прародительского поля данных или метода в классе-потомке утеряна. Иногда говорят, что *поле* или метод *затеняются* в потомке. В этих случаях используют вызов `super.имяПоля` или `super.имяМетода(список параметров)`. *Слово super* в этих случаях означает сокращение от *superclass*. Если метод или *поле* заданы не в непосредственном прародителе, а унаследованы от более далекого прародителя, соответствующие вызовы все равно будут работать. Но комбинации вида `super.super.имя` не разрешены.

Использовать вызовы с помощью слова *super* разрешается только для методов и полей данных объектов. Для методов и *переменных класса* (то есть объявленных с модификатором `static`) вызовы с помощью ссылки *super* запрещены.

6.7. Статическое и динамическое связывание методов. Полиморфизм

Данный *параграф*, несмотря на краткость, является очень важным – практически все профессиональное *программирование* в *Java* основано на использовании полиморфизма. В то же время эта тема является одной из наиболее сложных для понимания учащимися. Поэтому рекомендуется внимательно перечитать этот *параграф* несколько раз.

Методы классов помечаются модификатором `static` не случайно – для них при компиляции программного кода действует *статическое связывание*. Это значит, что в контексте какого класса указано имя метода в исходном коде, на метод того класса в скомпилированном коде и ставится *ссылка*. То есть осуществляется *связывание имени метода* в месте вызова *с исполняемым кодом* этого метода. Иногда *статическое связывание* называют *ранним связыванием*, так как оно происходит на этапе компиляции программы. *Статическое связывание* в *Java* используется еще в одном случае – когда *класс* объявлен с модификатором `final` ("финальный", "окончательный").

Методы объектов в *Java* являются динамическими, то есть для них действует *динамическое связывание*. Оно происходит на этапе выполнения программы непосредственно во время вызова метода, причем на этапе написания данного метода заранее неизвестно, из какого класса будет проведен вызов. Это определяется типом объекта, для которого работает данный код – какому классу принадлежит *объект*, из того класса вызывается метод. Такое *связывание* происходит гораздо позже того, как был скомпилирован код метода. Поэтому такой тип связывания часто называют *поздним связыванием*.

Программный код, основанный на вызове *динамических методов*, обладает свойством *полиморфизма* – один и тот же код работает *по*-разному в зависимости от того, *объект* какого типа его вызывает, но делает одни и те же вещи на уровне абстракции, относящейся к исходному коду метода.

Для пояснения этих не очень понятных при первом чтении слов рассмотрим пример из предыдущего параграфа – работу метода moveTo. Неопытным программистам кажется, что этот метод следует переопределять в каждом классе-наследнике. Это действительно можно сделать, и все будет правильно работать. Но такой код будет крайне избыточным – ведь реализация метода будет во всех классах-наследниках *Figure* совершенно одинаковой:

```
public void moveTo(int x, int y){
 hide();
 this.x=x;
 this.y=y;
 show();
};
```

Кроме того, в этом случае не используются преимущества полиморфизма. Поэтому мы не будем так делать.

Еще часто вызывает недоумение, зачем в *абстрактном классе* `Figure` писать реализацию данного метода. Ведь используемые в нем вызовы методов `hide` и `show`, на первый взгляд, должны быть вызовами *абстрактных методов* – то есть, кажется, вообще не могут работать!

Но методы `hide` и `show` являются динамическими, а это, как мы уже знаем, означает, что *связывание* имени метода и его исполняемого кода производится на этапе выполнения программы. Поэтому то, что данные методы указаны в контексте класса `Figure`, вовсе не означает, что они будут вызываться из класса `Figure`! Более того, можно гарантировать, что методы `hide` и `show` никогда не будут вызываться из этого класса. Пусть у нас имеются переменные `dot1` типа `Dot` и `circle1` типа `Circle`, и им назначены ссылки на объекты соответствующих типов. Рассмотрим, как поведут себя вызовы `dot1.moveTo(x1,y1)` и `circle1.moveTo(x2,y2)`.

При вызове `dot1.moveTo(x1,y1)` происходит вызов из класса `Figure` метода `moveTo`. Действительно, этот метод в классе `Dot` не переопределен, а значит, он наследуется из `Figure`. В методе `moveTo` первый оператор – вызов динамического метода `hide`. Реализация этого метода берется из того класса, экземпляром которого является *объект* `dot1`, вызывающий данный метод. То есть из класса `Dot`. Таким образом, скрывается точка. Затем идет изменение координат объекта, после чего вызывается *динамический метод* `show`. Реализация этого метода берется из того класса, экземпляром которого является *объект* `dot1`, вызывающий данный метод. То есть из класса `Dot`. Таким образом, на новом месте показывается точка.

Для вызова `circle1.moveTo(x2,y2)` все абсолютно аналогично – динамические методы `hide` и `show` вызываются из того класса, экземпляром которого является *объект* `circle1`, то есть из класса `Circle`. Таким образом, скрывается на старом месте и показывается на новом именно *окружность*.

То есть если *объект* является точкой, перемещается точка. А если *объект* является окружностью – перемещается *окружность*. Более того, если когда-нибудь кто-нибудь напишет, например, *класс* `Ellipse`, являющийся наследником `Circle`, и создаст *объект* `Ellipse ellipse=new Ellipse(…)`, то вызов `ellipse.moveTo(…)` приведет к перемещению на новое *место* эллипса. И происходить это будет в соответствии с тем, каким образом в классе `Ellipse` реализуют методы `hide` и `show`. Заметим, что работать будет давным-давно скомпилированный полиморфный код класса `Figure`. *Полиморфизм* обеспечивается тем, что ссылки на эти методы в код метода `moveTo` в момент компиляции не ставятся – они настраиваются на методы с такими именами из класса вызывающего объекта непосредственно в момент вызова метода `moveTo`.

В объектно-ориентированных языках программирования различают две разновидности *динамических методов* – собственно динамические и *виртуальные*. *По* принципу работы они совершенно аналогичны и отличаются только особенностями реализации. Вызов *виртуальных методов* быстрее. Вызов динамических медленнее, но служебная *таблица динамических методов* (*DMT – Dynamic* Methods *Table*) занимает чуть меньше памяти, чем *таблица виртуальных методов* (*VMT – Virtual* Methods *Table*).

Может показаться, что вызов *динамических методов* неэффективен с точки зрения затрат *по* времени из-за длительности поиска имен. На самом деле во время вызова поиска имен не делается, а используется гораздо более быстрый механизм, использующий упомянутую таблицу виртуальных (динамических) методов. Но мы на особенностях реализации этих таблиц останавливаться не будем, так как в *Java* нет различения этих видов методов.

6.8. Базовый класс Object

Класс `Object` является базовым для всех классов *Java*. Поэтому все его поля и методы наследуются и содержатся во всех классах. В классе `Object` содержатся следующие методы:

public Boolean equals(Object obj) – возвращает `true` в случае, когда равны значения объекта, из которого вызывается метод, и объекта, передаваемого через ссылку `obj` в списке параметров. Если объекты не равны, возвращается `false`. В классе `Object` равенство рассматривается как равенство ссылок и эквивалентно оператору сравнения `"=="`. Но в потомках этот метод может быть переопределен, и может сравнивать объекты по их содержимому. Например, так происходит для объектов оболочечных числовых классов. Это легко проверить с помощью такого кода:

```
Double d1=1.0,d2=1.0;
System.out.println("d1==d2 ="+(d1==d2));
System.out.println("d1.equals(d2) ="+(d1.equals(d2)));
```

Первая строка вывода даст `d1==d2 =false`, а вторая `d1.equals(d2) =true`

public int hashCode() – выдает *хэш-код* объекта. *Хэш-кодом* называется условно уникальный числовой идентификатор, сопоставляемый какому-либо элементу. Из соображений безопасности выдавать адрес объекта прикладной программе нельзя. Поэтому в Java *хэш-код* заменяет адрес объекта в тех случаях, когда для каких-либо целей надо хранить таблицы адресов объектов.

protected Object clone() throws CloneNotSupportedException – метод занимается копированием объекта и возвращает ссылку на созданный клон (дубликат) объекта. В наследниках класса `Object` его обязательно надо переопределить, а также указать, что класс реализует интерфейс `Clonable`. Попытка вызова метода из объекта, не поддерживающего *клонирования*, вызывает *возбуждение исключительной ситуации* `CloneNotSupportedException` ("Клонирование не поддерживается"). Про интерфейсы и исключительные ситуации будет рассказано в дальнейшем.

Различают два вида *клонирования*: мелкое (`shallow`), когда в клон один к одному копируются значения полей оригинального объекта, и глубокое (`deep`), при котором для полей *ссылочного типа* создаются новые объекты, клонирующие объекты, на которые ссылаются поля оригинала. При мелком клонировании и оригинал, и клон будут ссылаться на одни и те же объекты. Если объект имеет поля только *примитивных типов*, различия между мелким и глубоким *клонированием* нет. Реализацией *клонирования* занимается программист, разрабатывающий класс, автоматического механизма *клонирования* нет. И именно на этапе разработки класса следует решить, какой вариант *клонирования* выбирать. В подавляющем большинстве случаев требуется *глубокое клонирование*.

public final Class getClass() – возвращает ссылку на метаобъект типа класс. С его помощью можно получать информацию о классе, к которому принадлежит объект, и вызывать его методы класса и *поля класса*.

protected void finalize() throws Throwable – вызывается перед уничтожением объекта. Должен быть переопределен в тех потомках `Object`, в которых требуется совершать какие-либо вспомогательные действия перед уничтожением объекта (закрыть файл, вывести сообщение, отрисовать что-либо на экране, и т.п.). Подробнее об этом методе говорится в соответствующем параграфе.

public String toString() – возвращает строковое представление объекта (настолько адекватно, насколько это возможно). В классе `Object` этот метод реализует выдачу в строку полного имени объекта (с именем пакета), после которого следует символ `'@'`, а затем в шестнадцатеричном виде *хэш-код* объекта. В большинстве стандартных классов этот метод переопределен. Для числовых классов возвращается строковое представление числа, для строковых – содержимое строки, для символьного – сам символ (а не строковое представление его кода!). Например, следующий фрагмент кода

```
Object obj=new Object();
System.out.println(" obj.toString() дает "+obj.toString());
Double d=new Double(1.0);
System.out.println(" d.toString()дает "+d.toString());
Character c='A';
System.out.println("c.toString() дает "+c.toString());
```

обеспечит вывод

```
obj.toString() дает java.lang.Object@fa9cf
d.toString()дает 1.0
c.toString()дает A
```

Также имеются методы **notify()**, **notifyAll()**, и несколько перегруженных вариантов метода **wait**, предназначенные для работы с потоками (threads). О них говорится в разделе, посвященном потокам.

6.9. Конструкторы. Зарезервированные слова super и this. Блоки инициализации

Как уже говорилось, объекты в *Java* создаются с помощью зарезервированного слова `new`, после которого идет *конструктор* – специальная *подпрограмма*, занимающаяся созданием объекта и инициализацией полей создаваемого объекта. Для него не указывается тип возвращаемого значения, и он не является ни методом объекта (вызывается через *имя класса* когда объекта еще нет), ни методом класса (в конструкторе доступен *объект* и его поля через ссылку `this`). На самом деле *конструктор* в сочетании с оператором `new` возвращает ссылку на создаваемый *объект* и может считаться особым видом методов, соединяющим в себе черты методов класса и методов объекта.

Если в объекте при создании не нужна никакая дополнительная *инициализация*, можно использовать *конструктор*, который *по* умолчанию присутствует для каждого класса. Это *имя класса*, после которого ставятся пустые круглые скобки – без списка параметров. Такой *конструктор* при разработке класса задавать не надо, он присутствует автоматически.

Если требуется *инициализация*, обычно применяют конструкторы со списком параметров. Примеры таких конструкторов рассматривались нами для классов `Dot` и `Circle`. Классы `Dot` и `Circle` были унаследованы от *абстрактных классов*, в которых не было конструкторов. Если же идет *наследование* от неабстрактного класса, то есть такого, в котором уже имеется *конструктор* (пусть даже и *конструктор по* умолчанию), возникает некоторая специфика. Первым оператором в конструкторе должен быть вызов конструктора из *суперкласса*. Но его делают не через имя этого класса, а с помощью зарезервированного слова `super` (от "*superclass*"), после которого идет необходимый для прародительского конструктора

список параметров. Этот *конструктор* инициализирует поля данных, которые наследуются от *суперкласса* (в том числе и от всех более ранних прародителей). Например, напишем *класс* `FilledCircle` – наследник от `Circle`, экземпляр которого будет отрисовываться как цветной круг.

```
package java_gui_example;
import java.awt.*;

public class FilledCircle extends Circle{

    /** Creates a new instance of FilledCircle */
    public FilledCircle(Graphics g,Color bgColor, int r,Color color) {
        super(g,bgColor,r);
        this.color=color;
    }

        public void show(){
        Color oldC=graphics.getColor();

        graphics.setColor(color);
        graphics.setXORMode(bgColor);
        graphics.fillOval(x,y,size,size);
        graphics.setColor(oldC);
        graphics.setPaintMode();
    }

    public void hide(){
        Color oldC=graphics.getColor();
        graphics.setColor(color);
        graphics.setXORMode(bgColor);
        graphics.fillOval(x,y,size,size);
        graphics.setColor(oldC);
        graphics.setPaintMode();
    }}
```

Вообще, логика создания сложно устроенных объектов: родительская часть объекта создается и инициализируется первой, начиная от части, доставшейся от класса `Object`, и далее *по* иерархии, заканчивая частью, относящейся к самому классу. Именно поэтому обычно первым оператором конструктора является вызов прародительского конструктора *super*(*список параметров*), так как обращение к неинициализированной части объекта, относящейся к ведению прародительского класса, может привести к непредсказуемым последствиям.

В данном классе мы применяем более совершенный способ отрисовки и "скрывания" фигур *по* сравнению с предыдущими классами. Он основан на использовании режима рисования `XOR` ("исключающее или"). Установка этого режима производится методом `setXORMode`. При этом повторный *вывод* фигуры на то же *место* приводит к восстановлению первоначального изображения в области вывода. Переход в обычный режим рисования осуществляется методом `setPaintMode`.

В конструкторах очень часто используют *зарезервированное слово* `this` для доступа к полям объекта, *видимость имен* которых перекрыта переменными из списка параметров конструктора. Но в конструкторах оно имеет еще одно применение – для обращения из одного варианта конструктора к другому, имеющему другой *список* параметров. Напомним, что наличие таких вариантов называется *перегрузкой* конструкторов. Например, пусть мы первоначально задали в классе `Circle` *конструктор*, в котором *значение* полей `x`, `y` и `r` задается случайным образом:

```
Circle(Graphics g, Color bgColor){
    graphics=g;
    this.bgColor=bgColor;
```

```
    size=(int)Math.round(Math.random()*40);
  }
```

Тогда *конструктор*, в котором случайным образом задаются значения полей `x` и `y`, а *значение* `size` задается через *список* параметров конструктора, можно написать так:

```
Circle(Graphics g, Color bgColor, int r){
  this(g, bgColor);
  size=r;
}
```

При вызове конструктора с помощью слова `this` требуется, чтобы вызов `this` был первым оператором в реализации вызывающего конструктора.

В отличие от языка C++ в *Java* не разрешается использование имени конструктора, отличающегося от имени класса.

Порядок вызовов при создании объекта некого класса (будем называть его дочерним классом):

- Создается объект, в котором все поля данных имеют значения по умолчанию (нули на двоичном уровне представления).
- Вызывается конструктор *дочернего класса*.
  - Конструктор *дочернего класса* вызывает конструктор родителя (непосредственного прародителя), а также по цепочке все прародительские конструкторы и инициализации полей, заданных в этих классах, вплоть до класса `Object`.
  - Проводится инициализация полей родительской части объекта значениями, заданными в декларации *родительского класса*.
  - Выполняется тело конструктора *родительского класса*.
- Проводится инициализация полей дочерней части объекта значениями, заданными в декларации *дочернего класса*.
- Выполняется тело конструктора *дочернего класса*.

Знание данного порядка важно в случаях, когда в конструкторе вызываются какие-либо методы объекта, и надо быть уверенным, что к моменту вызова этих методов *объект* получит правильные значения полей данных.

Как правило, для инициализации полей сложно устроенных объектов используют конструкторы. Но кроме них в *Java*, в отличие от большинства других языков программирования, для этих целей могут также служить блоки инициализации класса и блоки инициализации объекта. *Синтаксис* задания классов с блоками инициализации следующий:

```
Модификаторы class ИмяКласса extends ИмяРодителя {
 Задание полей;
 static {
 тело блока инициализации класса
 }

{
 тело блока инициализации объекта
 }

 Задание подпрограмм - методов класса, методов объекта, конструкторов
}
```

Блоков инициализации класса и блоков инициализации объекта может быть несколько.

Порядок выполнения операторов при наличии блоков инициализации главного *класса приложения* (содержащего метод `main`):

- инициализация полей данных и выполнение блоков инициализации класса (в порядке записи в декларации класса);
- метод `main`;

выполнение блоков инициализации объекта;

выполнение тела конструктора класса.

Для других классов порядок аналогичен, но без вызова метода `main`:

инициализация полей данных и выполнение блоков инициализации класса (в порядке записи в декларации класса);

выполнение блоков инициализации объекта;

выполнение тела конструктора класса.

Чем лучше пользоваться, блоками инициализации или конструкторами? Ответ, конечно, неоднозначен: в одних ситуациях – конструкторами, в других – блоками инициализации. Для придания начальных значений *переменным класса* в случаях, когда для этого требуются сложные алгоритмы, можно пользоваться только статическими блоками инициализации. Для инициализации полей объектов в общем случае лучше пользоваться конструкторами, но если необходимо выполнить какой-либо код инициализации до вызова унаследованного конструктора, можно воспользоваться блоком динамической инициализации.

6.10. Удаление неиспользуемых объектов и метод finalize. Проблема деструкторов для сложно устроенных объектов

Как мы знаем, *конструктор* занимается созданием и рядом дополнительных действий, связанных с *инициализацией объекта*. Уничтожение объекта также может требовать дополнительных действий. В таких языках программирования как C++ или *Object PASCAL* для этих целей используют *деструкторы* – методы, которые уничтожают *объект*, и совершают все сопутствующие этому сопроводительные действия.

Например, у нас имеется *список* фигур, отрисовываемых на экране, и мы хотим удалить из этого списка какую-нибудь фигуру. Перед уничтожением фигура должна исключить себя из списка, затем дать команду списку заново отрисовать содержащиеся в нем фигуры, и только после этого "умереть". Именно такого рода действия характерны для *деструкторов*. Заметим, что возможна другая логика работы: дать списку команду исключить из него фигуру, после чего перерисовать фигуры, содержащиеся в списке. Но желательно, чтобы *язык программирования* поддерживал возможность реализации обоих подходов.

В *Java* имеется метод `finalize()`. Если в классе, который производит завершающие действия перед уничтожением объекта сборщиком мусора, переопределить этот метод, он, как может показаться, может служить некой заменой *деструктора*. Но так как момент уничтожения объекта неопределен и может быть отнесен *по* времени очень далеко от момента потери ссылки на *объект*, метод `finalize` не может служить реальной заменой деструктору. Даже явный вызов сборщика мусора `System.gk()` сразу после вызова метода `finalize()` не слишком удачное решение, так как и в этом случае нет гарантии правильности порядка высвобождения ресурсов. Кроме того, сборщик мусора потребляет много ресурсов и в ряде случаев может приостановить работу программы на заметное время.

Гораздо более простым и правильным решением будет написать в базовом классе разрабатываемой вами иерархии метод `destroy()` – "уничтожить, разрушить", который будет заниматься выполнением всех необходимых вспомогательных действий (можно назвать метод `dispose()` – "избавиться, отделаться", можно `free()` – "освободить"). Причем при необходимости надо будет переопределять этот метод в классах-наследниках. В случае, когда надо вызывать прародительский *деструктор*, следует делать вызов `super.destroy()`. При этом желательно, чтобы он был последним оператором в *деструкторе класса* – в противном случае может оказаться неправильной логика работы *деструктора*. Например, произойдет попытка обращения к объекту, исключенному из списка, или попытка записи в уже *закрытый файл*.

Логика разрушения объектов является обратной той, что используется при их создании: сначала разрушается часть, относящаяся к самому классу. Затем разрушается часть, относящаяся к непосредственному прародителю, и далее *по* иерархии, заканчивая частью, относящейся к базовому классу. Поэтому последним оператором *деструктора* бывает вызов прародительского *деструктора* `super.destroy()`.

6.11. Перегрузка методов

Напомним, что *имя функции* в сочетании с числом параметров и их типами называется *сигнатурой* функции. Тип возвращаемого значения и имена параметров в сигнатуру не входят. Понятие сигнатуры важно при задании подпрограмм с одинаковыми именами, но разными списками параметров – *перегрузке* (*overloading*) подпрограмм. Методы, имеющие одинаковое имя, но разные сигнатуры, разрешается перегружать. Если же сигнатуры совпадают, *перегрузка* запрещена. Для задания *перегруженных методов* в *Java* не требуется никаких дополнительных действий *по* сравнению с заданием обычных методов. Если же *перегрузка* запрещена, *компилятор* выдаст *сообщение об ошибке*.

Чаще всего перегружают конструкторы при желании иметь разные их варианты, так как имя конструктора определяется именем класса. Например, рассмотренные ранее конструкторы

```
Circle(Graphics g, Color bgColor){
  …
}
```

и

```
Circle(Graphics g, Color bgColor, int r){
  …
}
```

отличаются числом параметров, поэтому *перегрузка* разрешена.

Вызов *перегруженных методов* синтаксически не отличается от вызова обычных методов, но все-таки в ряде случаев возникает некоторая специфика из-за неочевидности того, какой вариант метода будет вызван. При разном числе параметров такой проблемы, очевидно, нет. Если же два варианта методов имеют одинаковое число параметров, и отличие только в типе одного или более параметров, возможны *логические ошибки*.

Напишем *класс* `Math1` , в котором имеется *подпрограмма-функция* `product,` вычисляющая *произведение* двух чисел, у которой имеются варианты с разными целыми типами параметров. Пример полезен как для иллюстрации проблем, связанных с вызовом *перегруженных методов*, так и для исследования проблем арифметического переполнения.

```
public class Math1 {

    public static byte product(byte x, byte y){
       return x*y;
    }

    public static short product(short x, short y){
       return x*y;
    }

    public static int product(int x, int y){
       return x*y;
    }

    public static char product(char x, char y){
       return x*y;
    }

    public static long product(long x, long y){
       return x*y;
    }

}
```

Такое задание методов разрешено, так как сигнатуры перегружаемых вариантов различны. Обратим внимание на типы возвращаемых значений – они могут задаваться *по* желанию программиста. Подпрограммы заданы как *методы класса* (`static`) для того, чтобы при их использовании не пришлось создавать *объект*.

Если бы мы попытались задать такие варианты методов:

```
public static byte product(byte x, byte y){
   return x*y;
}

public static int product(byte a, byte b){
```

```
    return a*b;
}
```

то *компилятор* выдал бы *сообщение об ошибке*, так как у данных вариантов одинаковая *сигнатура*. – Ни тип возвращаемого значения, ни имена параметров на сигнатуру не влияют.

Если при вызове метода `product` параметры имеют типы, совпадающие с заданными в одном из перегруженных вариантов, все просто. Но что произойдет в случае, когда в качестве параметра будут переданы значения типов `byte` и `int`? Какой вариант будет вызван? Проверка идет при компиляции программы, при этом перебираются все допустимые варианты. В нашем случае это `product(int x, int y)` и `product(long x, long y)`. Остальные варианты не подходят из-за типа второго параметра – тип подставляемого значения должен иметь *диапазон* значений, "вписывающийся" в *диапазон* вызываемого метода. Из допустимых вариантов выбирается тот, который ближе *по* типу параметров, то есть в нашем случае `product(int x, int y)`.

Если среди *перегруженных методов* среди разрешенных вариантов не удается найти предпочтительный, при компиляции класса, где делается вызов, выдается *диагностика* ошибки. Так бы случилось, если бы мы имели следующую реализацию класса `Math2`

```
public class Math2 {

    public static int product(int x, byte y){
       return x*y;
    }

    public static int product(byte x, int y){
       return x*y;
    }

}
```

и в каком-нибудь другом классе имели переменные `byte b1, b2` и сделали вызов `Math2.product(b1,b2)`. Оба варианта перегруженного метода подходят, и выбрать более подходящий невозможно. Отметим, что *класс* `Math2` при этом компилируется без проблем – в нем самом ошибок нет. Проблема в том классе, который его использует.

Самая неприятная особенность перегрузки – вызов не того варианта метода, на который рассчитывал программист. Особо *опасные ситуации* при этом возникают в случае, когда *перегруженные методы* отличаются типом параметров, и в качестве таких параметров выступают *объектные переменные*. В этом случае близость *совместимых типов* определяется *по* близости в *иерархии наследования* – *по* числу этапов наследования. Отметим, что выбор перегруженного варианта проводится статически, на этапе компиляции. Поэтому тип, используемый для этого выбора, определяется типом *объектной переменной*, передаваемой в качестве параметра, а не типом объекта, который этой переменной назначен.

6.12. Правила совместимости ссылочных типов как основа использования полиморфного кода. Приведение и проверка типов

Мы уже говорили, что полиморфный код обеспечивает основные преимущества объектного программирования. Но как им воспользоваться? Ведь тип объектных переменных задается на этапе компиляции. Решением проблемы является следующее правило:

переменной некоторого объектного типа можно присваивать выражение, имеющее тот же тип или тип класса-наследника.

Аналогичное правило действует при передаче *фактического параметра* в подпрограмму:

В качестве фактического параметра вместо формального параметра некоторого объектного типа можно подставлять выражение, имеющее тот же тип или тип класса-наследника.

В качестве выражения может выступать *переменная объектного типа*, оператор создания нового объекта (*слово* `new`, за которым следует *конструктор*), *функция объектного типа* (в том числе приведения *объектного типа*).

Поэтому если мы создадим переменную базового типа, для которой можно писать полиморфный код, этой переменной можно назначить ссылку на *объект*, имеющий тип любого из классов-потомков. В том числе – еще не написанных на момент компиляции базового класса. Пусть, например, мы хотим написать подпрограмму, позволяющую перемещать фигуры из нашей иерархии не в точку с новыми координатами, как

метод `moveTo`, а на необходимую величину `dx` и `dy` *по* соответствующим осям. При этом у нас отсутствуют исходные коды базового класса нашей иерархии (либо их запрещено менять). Для этих целей создадим *класс* `FiguresUtil` (сокращение от Utilities – утилиты, служебные программы), а в нем зададим метод `moveFigureBy` ("переместить фигуру на").

```
public class FiguresUtil{


  public static void moveFigureBy(Figure figure,int dx, int dy){
    figure.moveTo(figure.x+dx, figure.y+dy);
  }
}
```

В качестве *фактического параметра* такой подпрограммы вместо `figure` можно подставлять *выражение*, имеющее тип любого класса из иерархии фигур. Пусть, например, новая фигура создается *по* нажатию на кнопку в зависимости от того, какой выбор сделал *пользователь* во время работы программы: если в радиогруппе отмечен *пункт* "Точка", создается *объект* типа `Dot`. Если в радиогруппе отмечен *пункт* "*Окружность*", создается *объект* типа `Circle`. Если же отмечен *пункт* "Круг", создается *объект* типа `FilledCircle`. Отметим также, что *класс* `FilledCircle` был написан уже после компиляции классов `Figure`, `Dot` и `Circle`.

Фрагмент кода для класса нашего приложения будет выглядеть так:

```
Figure figure;
java.awt.Graphics g=jPanel1.getGraphics();


//обработчик кнопки создания фигуры
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
  if(jRadioButton1.isSelected() )
    figure=new Dot(g,jPanel1.getBackground());
  if(jRadioButton2.isSelected())
    figure=new Circle(g,jPanel1.getBackground());
  if(jRadioButton3.isSelected())
    figure=new FilledCircle(g,jPanel1.getBackground(),20,
                                    java.awt.Color.BLUE);
  figure.show();
}


//обработчик кнопки передвижения фигуры
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
  int dx= Integer.parseInt(jTextField1.getText());
  int dy= Integer.parseInt(jTextField2.getText());
  FiguresUtil.moveFigureBy(figure,dx,dy);
}
```

При написании программы неизвестно, ни какого типа будет передвигаемый *объект*, ни насколько его передвинут – все зависит от решения пользователя во время работы программы. Именно возможность назначения ссылки на *объект* класса-потомка обеспечивает возможность использования полиморфного кода.

Следует обратить внимание на еще один момент – стиль написания вызова

```
FiguresUtil.moveFigureBy(figure,dx,dy);
```

Можно было бы написать его так:

```
FiguresUtil.moveFigureBy(
  figure,
  Integer.parseInt(jTextField1.getText()),
  Integer.parseInt(jTextField2.getText())
);
```

При этом экономились бы две локальные переменные (аж целых 8 *байт* памяти!), но читаемость, понимаемость и отлаживаемость кода стали бы гораздо меньше.

Часто встречающаяся ошибка: пытаются присвоить переменной типа "наследник" *выражение* типа "прародитель". Например,

```
Figure figure;
Circle circle;
…
figure =new Circle (); //так можно
…
circle= figure; - Так нельзя! Выдастся ошибка компиляции.
```

Несмотря на то, что переменной `figure` назначен *объект* типа `Circle` – ведь проверка на допустимость присваивания делается на этапе компиляции, а не динамически.

Если программист не уверен, что *объект* имеет тип класса-потомка, в таких случаях надо использовать *приведение типа*. Для приведения типа перед выражением или именем переменной в круглых скобках ставят имя того типа, к которому надо осуществить приведение:

```
Figure figure;
Circle circle;
Dot dot;
…
figure =new Circle (); //так можно
…
circle= (Circle)figure; //так можно!
dot=(Dot) figure; //так тоже можно!
```

Отметим, что приведение типа принципиально отличается от преобразования типа, хотя синтаксически записывается так же. Преобразование типа приводит к изменению содержимого ячейки памяти и может приводить к изменению ее размера. А вот приведение типа не меняет ни размера, ни содержимого никаких ячеек памяти – оно меняет только тип, сопоставляемый ячейке памяти. В *Java* приведение типа применяется к *ссылочным типам*, а преобразование – к примитивным. Это связано с тем, что изменение типа ссылочной переменной не приводит к изменению той ячейки, на которую она ссылается. То есть в случае приведения *тип объекта* не меняется – меняется тип ссылки на *объект*.

Приводить тип можно как в сторону генерализации, так и в сторону специализации.

Приведение в сторону генерализации является безопасным, так как *объект* класса-потомка всегда является экземпляром прародителя, хоть и усложненным. А вот приведение в сторону специализации является опасным – вполне допустимо, что во *время выполнения* программы окажется, что *объект*, назначенный переменной, не является экземпляром нужного класса. Например, при приведении `(Circle)figure` может оказаться, что переменной `figure` назначен *объект* типа `Dot`, который не может быть приведен к типу `Circle`. В этом случае возникает *исключительная ситуация* приведения типа (`typecast`).

Возможна программная проверка того, что *объект* является экземпляром заданного класса:

```
if(figure instanceof Circle)
  System.out.println("figure instanceof Circle");
```

Иногда вместо работы с самими классами бывает удобно использовать ссылки на *класс*. Они получаются с помощью доступа к полю `.class` из любого класса.

Возможно создание переменных типа "*ссылка* на *класс*":

```
Class c=Circle.class;
```

Их можно использовать для обращения к *переменным класса* и методам класса. Кроме того, переменных типа "*ссылка* на *класс*" можно использовать для создания экземпляров этого класса с помощью метода `newInstance()`:

```
Circle circle=(Circle)c.newInstance();
```

Возможна программная проверка соответствия объекта нужному типу с помощью ссылки на *класс*:

```
if(figure.getClass()==Circle.class)
        circle= (Circle)figure;
…;
```

Но следует учитывать, что при такой проверке идет сравнение на точное *равенство* классов, а не на допустимость приведения типов. А вот оператор `isInstance` позволяет проверять, является ли *объект* `figure` экземпляром класса, на который ссылается `c`:

```
if(c.isInstance(figure))
    System.out.println("figure isInstance of Circle");
```

6.13. Рефакторинг

Одним из важных элементов современного программирования является *рефакторинг* – изменение структуры существующего проекта без изменения его функциональности.

Приведем три наиболее часто встречающихся примера рефакторинга.

Во-первых, это переименование элементов программы – классов, переменных, методов.

Во-вторых, перемещение элементов программы с одного места на другое.

В-третьих, инкапсуляция полей данных.

В сложных проектах, конечно, возникают и другие варианты рефакторинга (например, выделение части кода в отдельный метод – `"Extract method"`) , но с упомянутыми приходится встречаться постоянно. Поэтому рассмотрим эти три случая подробнее.

Первый случай – переименование элементов программы.

Для того, чтобы в среде NetBeans переименовать элемент, следует щелкнуть *по* его имени правой кнопкой мыши. Это можно сделать в исходном коде программы, а можно и в окне **Projects** или **Navigator**. В появившемся всплывающем *меню* следует выбрать **Refactor/Rename…** После чего ввести новое имя и нажать кнопку **"Next>"**.

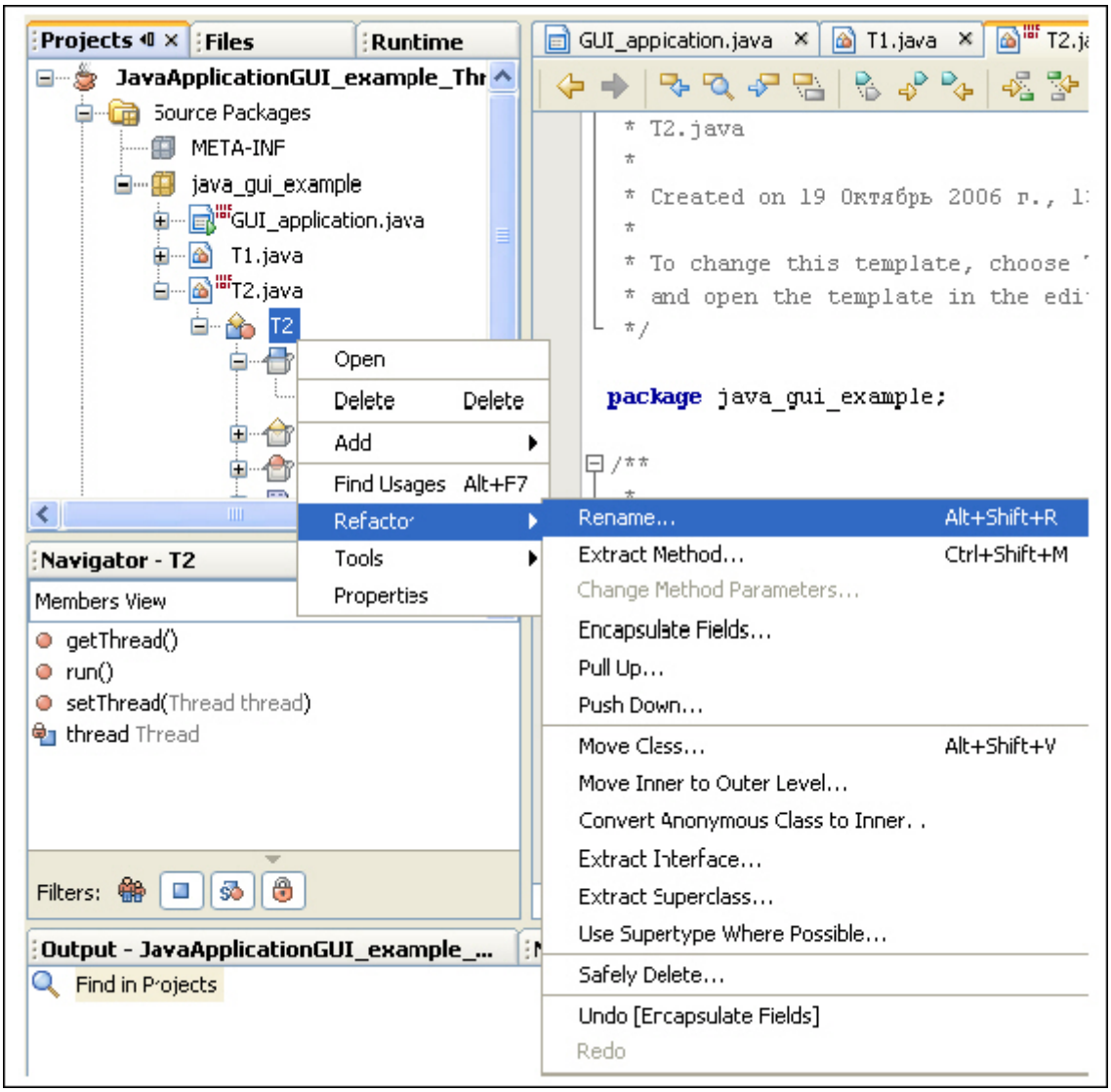


Рис. 6.5. Переименование класса. Шаг 1

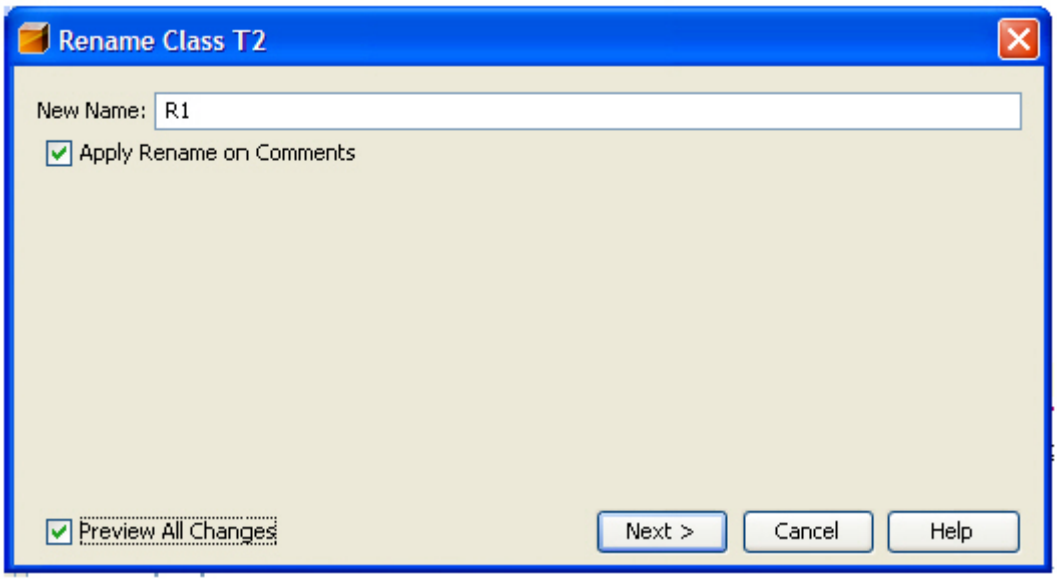
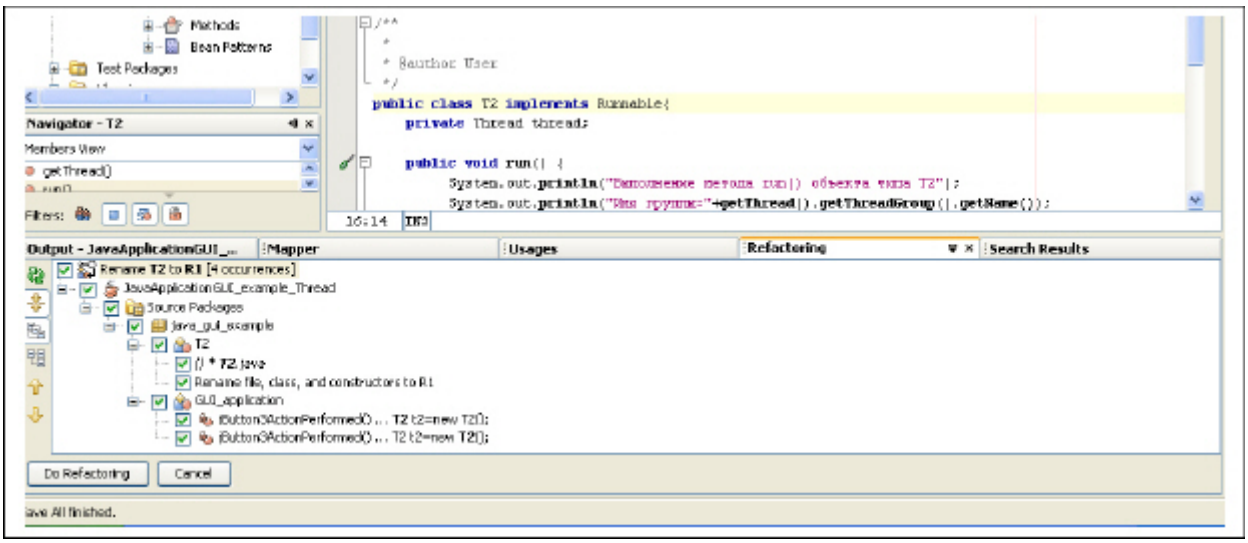


Рис. 6.6. Переименование класса. Шаг 2

Если галочка **"Preview All Changes"** ("Предварительный просмотр всех изменений") не снята, в самом нижнем окне, **Output** ("*Вывод*"), появится *дерево* со списком мест, где будут проведены исправления. В случае необходимости галочки можно снять, и в этих местах переименование проводиться не будет. При нажатии на кнопку **"Do Refactoring"** ("Провести рефакторинг") проводится операция переименования в выбранных местах программы. В отличие от обычных *текстовых процессоров* переименование происходит с учетом синтаксиса программы, так что элементы, не имеющие отношения к переименовываемому, но имеющие такие же имена, не затрагиваются. Что в выгодную сторону отличает NetBeans от многих других сред разработки, не говоря уж об обычных текстовых редакторах.



увеличить изображение

Рис. 6.7. Переименование класса. Шаг 3

Требуется быть внимательными: довольно часто начинающие программисты не замечают появления в окне **Output** списка изменений и кнопки **"Do Refactoring"**. Особенно если *высота* этого окна сделана очень малой. Если в диалоге переименования (шаг 2) флажок **"Preview all Changes"** снят, при нажатии на кнопку **"Next>"** сразу происходит рефакторинг.

Следует также отметить, что после проведения рефакторинга возможен возврат к первоначальному состоянию ("*откат*", операция undo). Обычно такая операция осуществляется с помощью главного *меню* проекта (кнопка **Undo** или *пункт меню* **Edit/Undo**), но в случае рефакторинга требуется правой клавишей мыши вызвать всплывающее окно и выбрать *пункт* **Refactor/Undo**. *Откат* может быть на несколько шагов назад путем повторения данного действия. При необходимости отказа от отката в *меню* рефакторинга следует выбрать *пункт* **Redo**.

Второй случай – перемещение элементов программы с одного места на другое.

Например, мы хотим переместить *класс* из одного пакета в другой. Для выполнения этого действия достаточно перетащить мышью в окне **Projects** узел, связанный с данным классом, в соответствующий пакет. При таком перемещении там, где это необходимо, автоматически добавляются *операторы* импорта.

Если при перемещении возникают проблемы, о них выдается сообщение. Как правило, проблемы бывают связаны с неправильными уровнями видимости. Например, если указан пакетный уровень видимости метода, он доступен другим классам этого пакета. А при переносе класса в другой пакет в месте исходного кода, где осуществляется такой *доступ*, в новом варианте кода возникает ошибка доступа. Перенос класса в отдельный пакет, отличающийся от пакета приложения – хороший способ проверить правильности выбранных уровней доступа для членов класса.

Аналогичным образом перемещаются пакеты. При этом все пакеты в дереве элементов показываются на одном уровне вложенности, но у вложенных пакетов имена квалифицируются именем родительского пакета.

Третий случай – инкапсуляция полей данных.

Напрямую давать *доступ* к полю данных – дурной тон программирования. Поэтому рекомендуется давать полям уровень видимости `private`, а *доступ* к ним *по* чтению и записи осуществлять с помощью методов `get` *ИмяПоля* и `set` *ИмяПоля* – получить и установить *значение* этого поля. Такие методы в *Java* называют геттерами (*getters*) и сеттерами (*setters*).

Но при введении в *класс* новых полей на первом этапе часто бывает удобнее задать поля с модификатором `public` и обеспечивать чтение значения полей напрямую, а изменение значения – путем присваивания полям новых значений. А затем можно исправить данный недостаток программы с помощью инкапсуляции полей данных. Это делается просто: в дереве элементов программы окна **Projects** в разделе **Fields** ("поля") щелкнем правой кнопкой мыши *по* имени поля и выберем в появившемся всплывающем *меню* **Refactor/Encapsulate Fields…** ("Провести рефакторинг"/ "Инкапсулировать поля…").В появившемся диалоге нажмем на кнопку **"Next>"** и проведем рефакторинг. При этом каждое *поле* приобретет *модификатор видимости* `private`, а во всех местах программы, где напрямую шел *доступ* к этому полю, в коде будет проведена замена на вызовы геттеров и сеттеров.

Более подробную информацию *по* идеологии и методах рефакторинга проектов, написанных на языке *Java*, можно найти в монографии [7]. Правда, эта книга уже несколько устарела – среда NetBeans позволяет делать в автоматическом режиме многие из описанных в [7] действий.

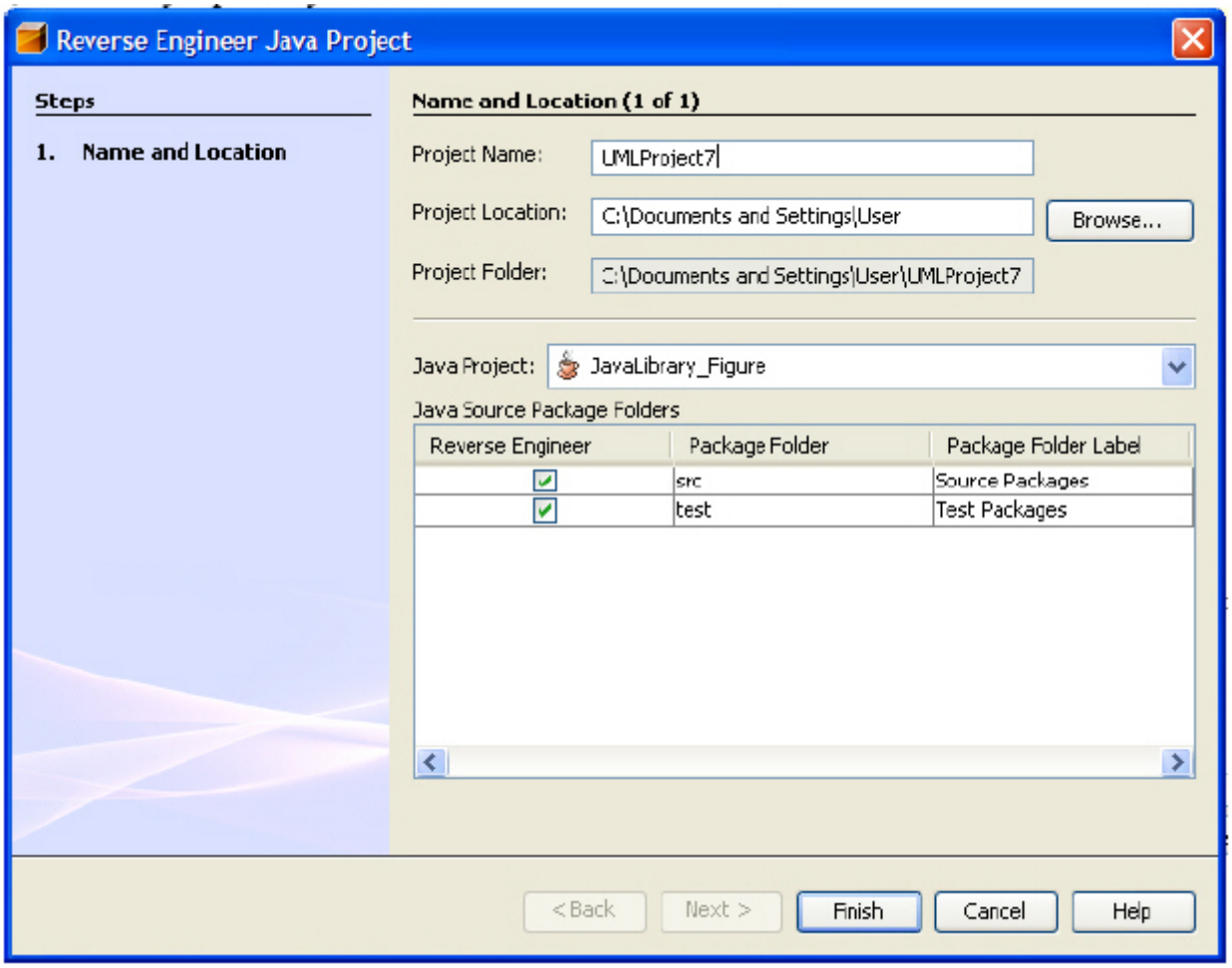
6.14. Reverse engineering – построение UML-диаграмм по разработанным классам

Среда NetBeans при установленном пакете NetBeans *Enterprise* Pack позволяет *по* имеющемуся исходному коду построить *UML*-диаграммы. Для этого следует открыть проект и нажать на главной панели среды разработки кнопку

"Reverse Engineer…"

Рис. 6.8. Кнопка "Reverse Engineering"

Появится диалоговая форма задания параметров создаваемого проекта, в которой следует изменить название проекта на осмысленное, *по* которому легко можно будет определить, к какому проекту *Java* он относится.



увеличить изображение

Рис. 6.9. Диалоговая форма задания параметров создаваемого UML-проекта

В нашем случае `UMLProject7` мы заменим на `UML_Figure`. После нажатия на кнопку **Finish** ("Закончить") будет выдана форма с ненужной вспомогательной информацией, и в ней следует нажать кнопку **Done** ("Сделано"). В результате чего мы получим новый *UML*-проект, в котором можно просмотреть параметры, относящиеся к каждому классу:

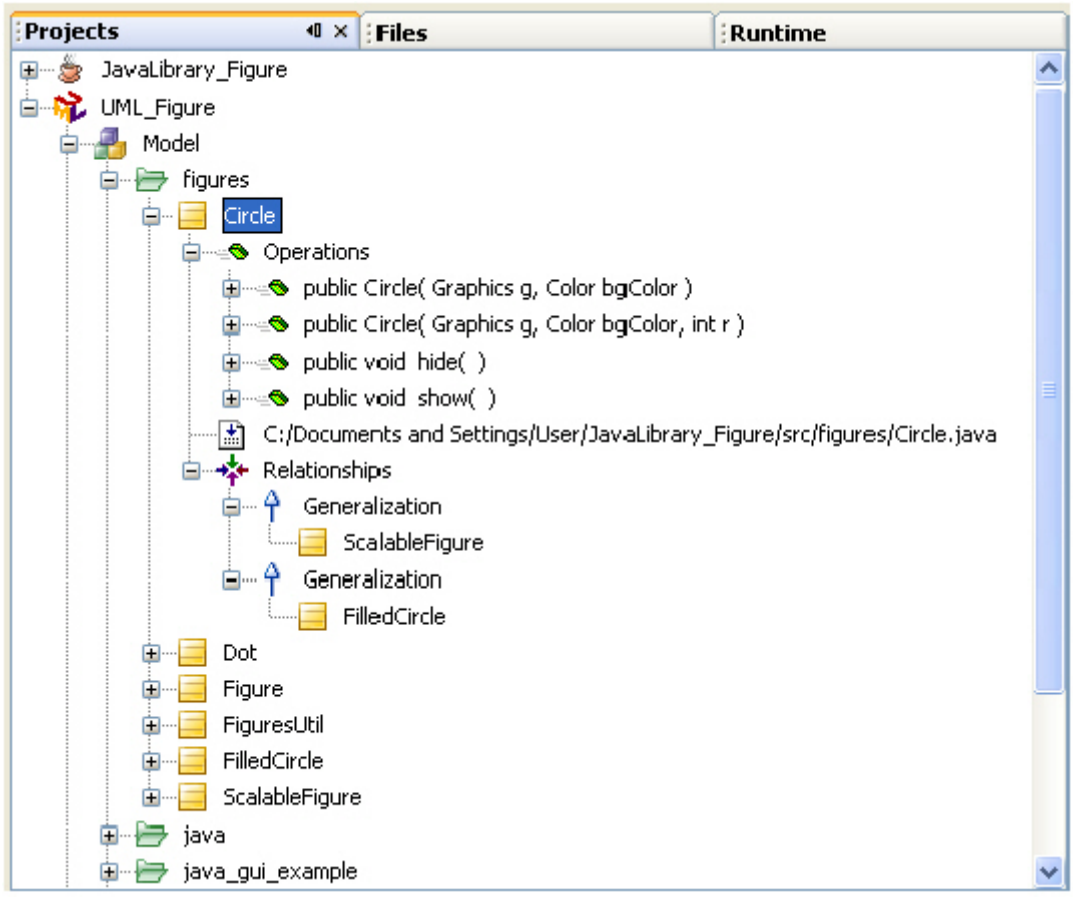


Рис. 6.10. Параметры UML-проекта, относящиеся к классу Circle

Для класса показываются конструкторы и обычные методы (узел `Operations`), а также *отношения наследования* и другие варианты отношений (узел *`Relationships`*).

В *UML*-проекте можно сгенерировать *UML*-диаграммы, щелкнув правой кнопкой мыши *по* имени соответствующего класса:

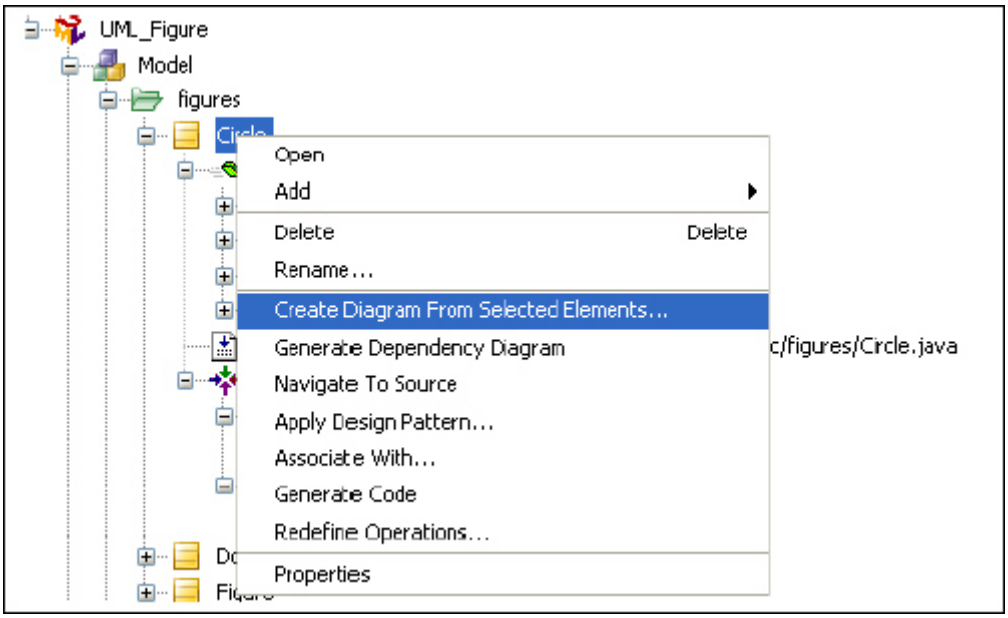


Рис. 6.11. Всплывающее меню действий с классом в UML-проекте

Если выбрать *пункт* **"Create Diagram From Selected Elements"** ("Создать диаграмму из выбранных элементов"), и далее выбрать тип диаграммы **"Class Diagram"**,

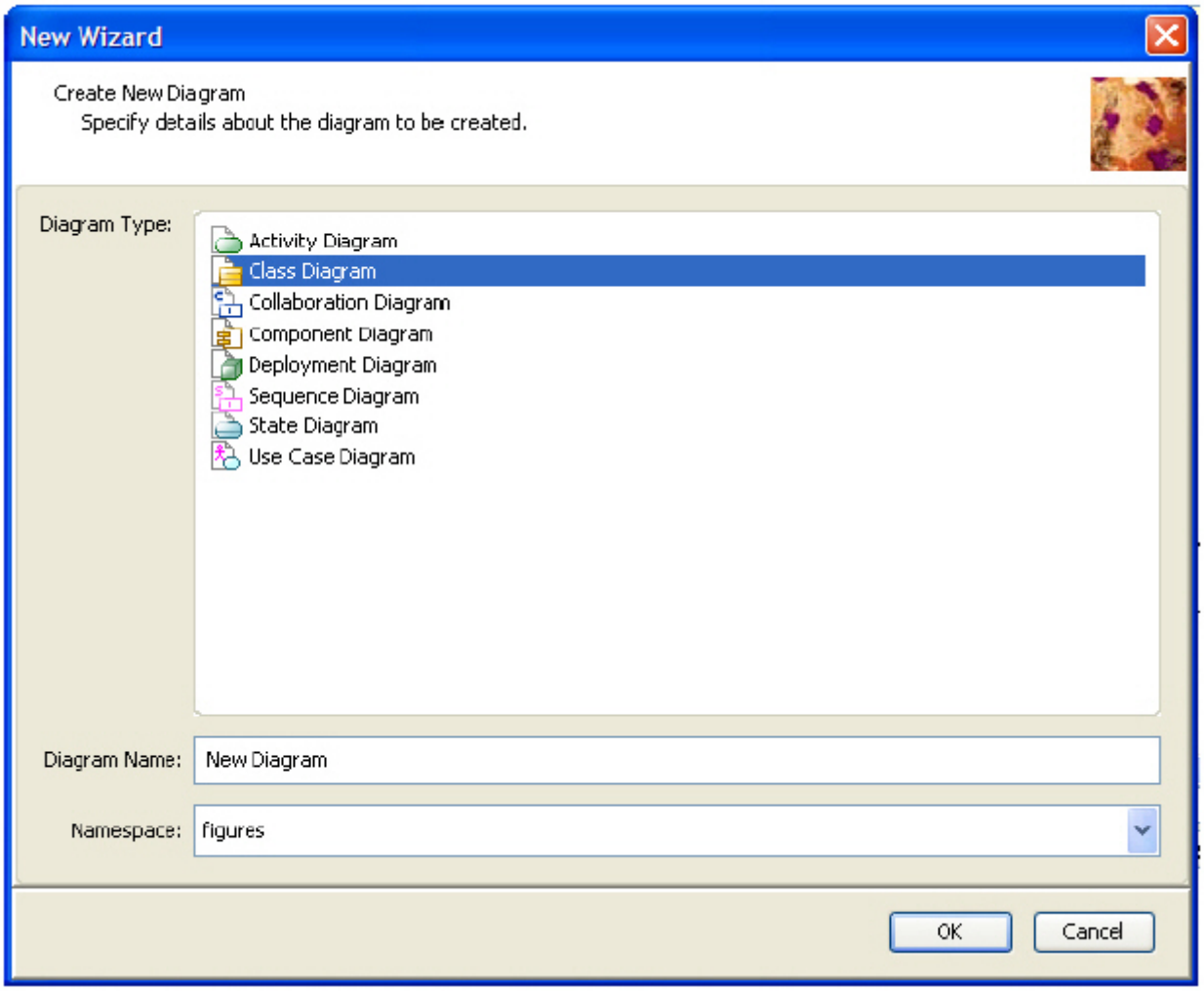


Рис. 6.12. Выбор типа создаваемой диаграмы

можно получить диаграмму такого вида:

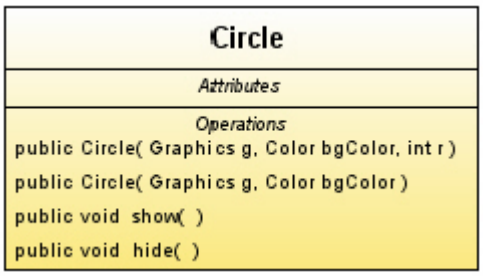


Рис. 6.13. Диаграмма для класса Circle

При этом лучше заменить имя создаваемой диаграммы, например, на *Circle Diagram* . Переименование можно сделать и позже, щелкнув правой кнопкой мыши *по* имени диаграммы и выбрав в появившемся всплывающем *меню пункт* **Rename…** ("Переименовать…").

Если же выделить *Circle*,*Dot*,*Figure*, ScalableFigure, мы получим диаграмму наследования, которой можно дать имя *Inheritance Diagram*.

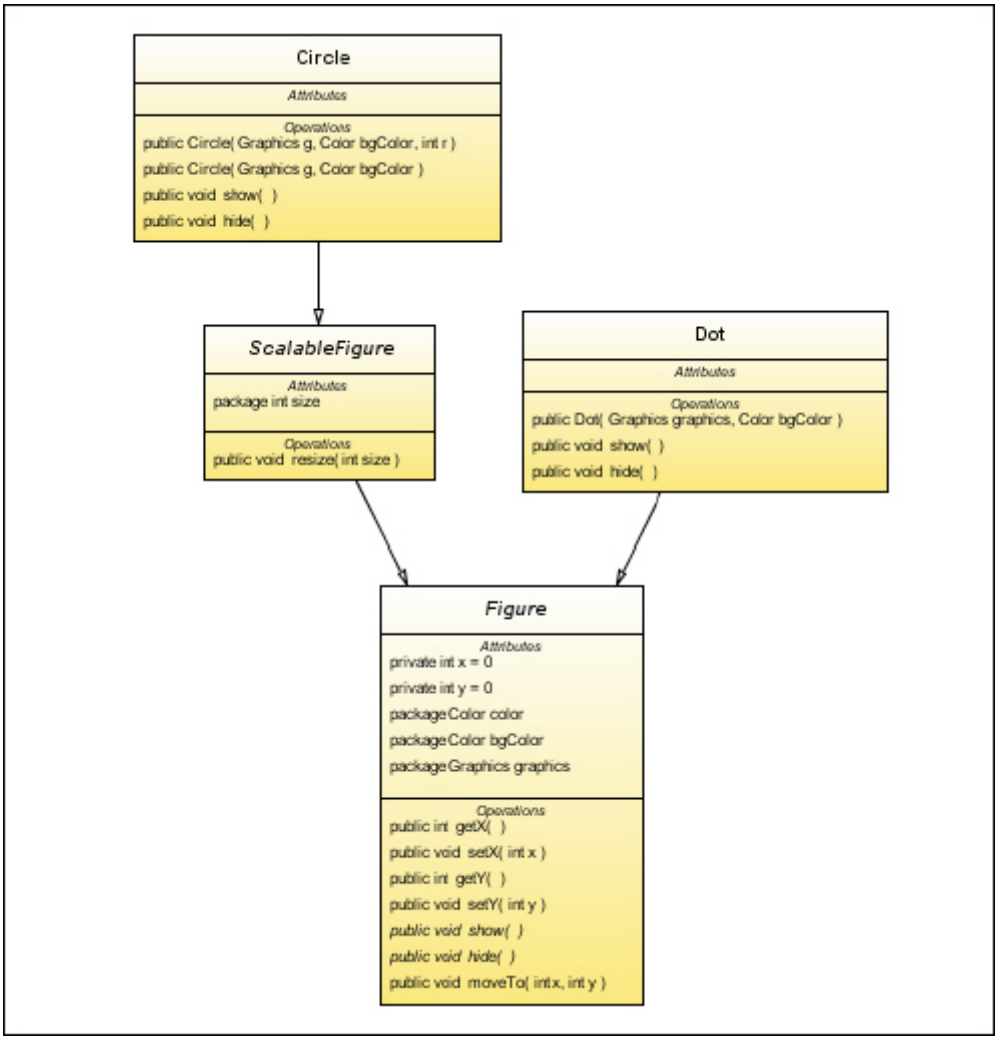


Рис. 6.14. Диаграмма для классов Circle,Dot,Figure, ScalableFigure

Если для класса `Circle` во всплывающем *меню* выбрать *пункт* **"Generate Dependency Diagram"** ("Сгенерировать диаграмму зависимостей"), получим следующую диаграмму :

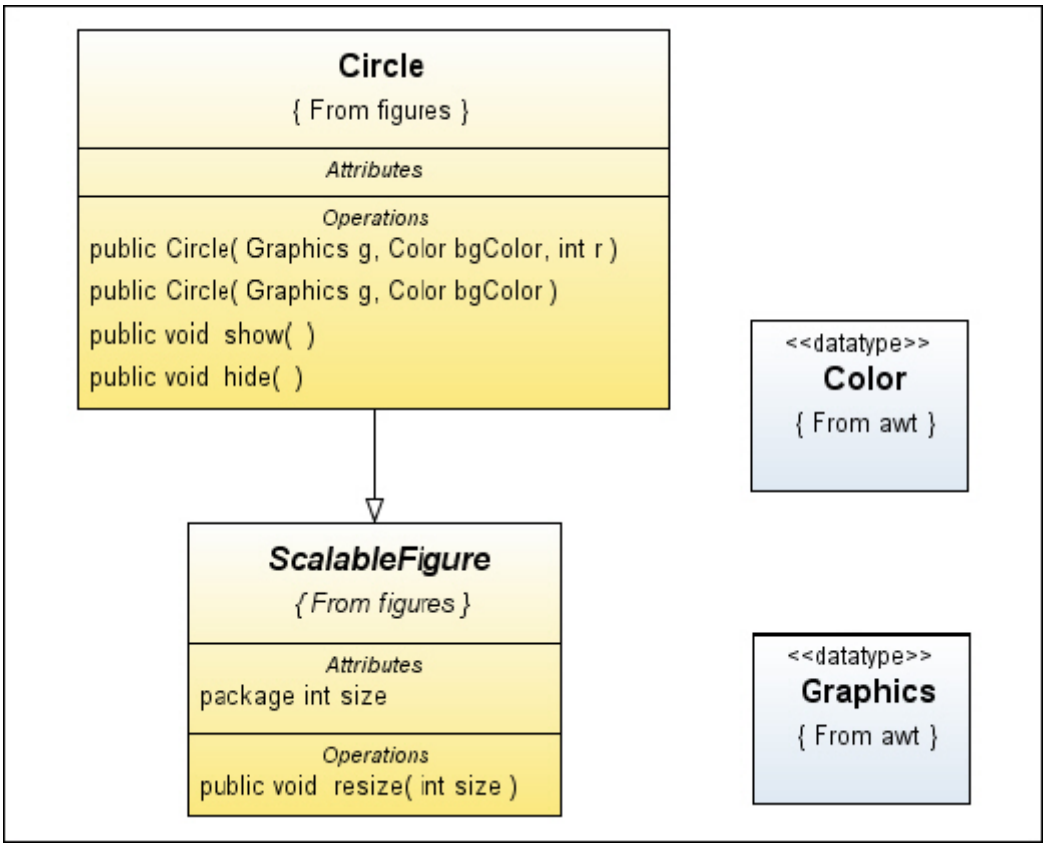


Рис. 6.15. Диаграмма зависимостей для класса Circle

Пункт всплывающего *меню* **Navigate to Source** позволяет вместо диаграмм показывать редактор исходного кода.

На диаграммах можно добавлять в классы или удалять из них поля и методы, проводить переименования, менять модификаторы. Причем изменения, сделанные на любой из диаграмм, автоматически отражаются как на других диаграммах *UML*-проекта, так и в исходном коде проекта *Java* (это проектирование – *Forward* Enineering). И наоборот – изменения, сделанные в исходном коде *Java*, автоматически применяются к диаграммам *UML* (это *обратное проектирование* – *Reverse* Enineering).

В настоящее время работа с *UML*-проектами в NetBeans *Enterprise* Pack не до конца отлажена, иногда наблюдаются "баги" (мелкие ошибки). Но можно надеяться, что в ближайшее время недостатки будут исправлены.

Краткие итоги

Наследование опирается на инкапсуляцию. Оно позволяет строить на основе первоначального класса новые, добавляя в классы новые поля данных и методы. Первоначальный класс называется *родителем* (*ancestor*), новые классы – его *потомками* (*descendants*). От потомков можно наследовать, получая очередных потомков. Набор классов, связанных *отношением наследования*, называется *иерархией классов*. Класс, стоящий во главе иерархии, от которого унаследованы все остальные (прямо или опосредованно), называется *базовым классом иерархии*.

Иерархия нужна для того, чтобы писать *полиморфный код*. Основные преимущества объектного программирования обеспечиваются наличием полиморфного кода.

Поля отражают состояние объекта, а методы – задают его поведение.

Чем ближе к основанию *иерархии* лежит класс, тем более общим и универсальным (general) он является. Чем дальше от базового класса иерархии стоит класс, тем более специализированным (specialized) он является.

Каждый объект класса-потомка при *любых значениях* его *полей данных* должен рассматриваться как экземпляр класса-прародителя на уровне абстракций поведения, но с некоторыми изменениями в реализации этого поведения.

Проектирование классов осуществляется с помощью *UML*-диаграмм.

В Java подпрограммы задаются только как методы в каком-либо классе и называются *функциями*. Объявление в классе функции состоит из задания *заголовка* и *тела* функции (ее *реализации*).

Параметры, указанные в заголовке функции при ее декларации, называются *формальными*. А те параметры, которые подставляются во время вызова функции, называются *фактическими*. *Формальные параметры* нужны для того, чтобы указать последовательность действий с *фактическими параметрами* после того, как те будут переданы в подпрограмму во время вызова. Это ни что иное, как особый вид локальных переменных, которые используются для обмена данными с внешним миром.

Параметры в Java передаются в функцию всегда *по значению*. Передача *по ссылке* отсутствует. В частности, ссылки передаются в функции по значению.

В Java имеются уровни видимости private ("частный", "закрытый"), пакетный, protected ("защищенный") и public (Общедоступный). Пакетный – уровень видимости по умолчанию для полей и методов, для задания остальных уровней используются модификаторы private , protected и public .

Ссылка this обеспечивает ссылку на объект из метода объекта. Чаще всего она используется при перекрытии области видимости имени поля объекта *формальным параметром* функции.

В классе-наследнике методы можно *переопределять*. При этом у них должен сохраняться *контракт* – в который входит весь заголовок метода за исключением имен формальных параметров.

Можно задавать *перегруженные* (overloaded) варианты методов, отличающиеся *сигнатурой*. В сигнатуру входит только часть заголовка метода – имя функции, а также число, порядок и тип ее параметров.

Переменной некоторого *объектного типа* можно присваивать выражение, имеющее тот же тип или тип класса-наследника. В качестве *фактического параметра* функции вместо *формального параметра* некоторого *объектного типа* можно подставлять выражение, имеющее тот же тип или тип класса-наследника. Именно это правило обеспечивает возможность использования полиморфного кода.

Для приведения типа используется имя типа, заключенное в круглые скобки – как и для преобразования типа. Но при преобразовании типа могут меняться содержимое и размер ячейки, к которой применяется данный оператор, а при приведении типа ячейка и ее содержимое остаются теми же, просто начинают считать, что у ячейки другой тип.

Проверка на то, что объект является экземпляром заданного класса, осуществляется оператором `instanceof`: `if(figure instanceof Circle).. .`

Возможна программная проверка точного соответствия объекта нужному типу с помощью ссылки на класс: `if(figure.getClass()==Circle.class).. .` При этом для экземпляра класса-наследника `Circle` сравнение даст `false`, в отличие от экземпляра `Circle`.

Оператор `isInstance` позволяет проверять, является ли тип объекта совместимым с классом, на который задана ссылка `c`: `if(c.isInstance(figure)).. .` При этом если класс, экземпляром которого является объект `figure`, является наследником класса с или совпадает с ним, сравнение даст `true`.

Рефакторинг – изменение структуры существующего проекта без изменения его функциональности. Три наиболее часто встречающихся примера рефакторинга:

Переименование элементов программы – классов, переменных, методов.

Перемещение элементов программы с одного места на другое.

Инкапсуляция полей данных.

С помощью средств *Reverse Engineering* можно создавать *UML*-диаграммы классов и зависимостей классов. Причем после создания *UML*-проекта, сопровождающего Java-проект, изменения, сделанные в исходном коде Java, автоматически применяются к диаграммам *UML*, и наоборот.

Типичные ошибки:

Очень часто ошибочно считают более общим объект, из которого можно получить другой при каких-либо *конкретных значениях полей данных*. Например, окружность считают частным значением эллипса, а точку – частным значением окружности. Но в объектном программировании отношения общности и специализации объектов определяются по сложности их устройства (наличию дополнительных полей данных и методов), а также их поведению при *произвольных* значениях их полей данных. Чем сложнее объект, тем он более специализирован (менее общий).

Очень часто возникают ошибки при попытке вернуть из функции измененные значения ссылочных переменных, переданных через список параметров. Для такого возврата надо либо использовать глобальную переменную, либо передавать ссылку на объект, полем которого является изменяемая переменная.

Используют неправильное приведение типа. Например, если переменной `Object object` присвоена ссылка на объект типа `Dot`, а пытаются сделать приведение `(Circle) object`. Такая ошибка на этапе компиляции не может быть распознана, и возникает исключительная ситуация неправильного приведения типа (`invalid typecast`) во время выполнения программы.

Задания

В классе `MathUtil` написать подпрограмму вычисления факториала `public static double factorial(int n)`

Модификатор `static` помечает подпрограмму как метод класса. То есть позволяет вызывать метод через имя класса без создания объекта.

Напомним, что факториал натурального числа n – это произведение всех натуральных чисел от 1 до n :

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$$

Кроме того, 0! считается равным 1. Обозначение факториала в виде n! математическое, в Java символ "!" зарезервирован для других целей. Также написать подпрограммы вычисления факториала с другими типами возвращаемых значений: `public static long factorial_long(int n)` и `public static int factorial_int(int n)`

Сравнить работу подпрограмм при `n=0,1,5,10,20,50,100`. Объяснить результаты.

Разработать в пакете приложения библиотеку классов для иерархии фигур. Реализация должна быть с заглушками при реализации методов `show` и `hide`. Вместо показа на экране эти методы должны выводить в консольное окно вывода имя класса фигуры и слово show или hide, а также координаты `x` и `y` фигуры.

Разработать приложение, в котором используются эти классы – создается фигура нужного типа при нажатии на кнопку **"показывается"**. Создать документационные комментарии для полей и методов разработанных классов. Вызвать генерацию документации для проекта, просмотреть в ней созданные комментарии.

Проверить в исходном коде проекта справку, возникающую при вызовах `figure.` и `dot.`, где эти переменные заданы как `Figure figure` и `Dot dot`.

Создать пакет figures . С помощью средств рефакторинга переместить классы фигур в пакет figures.

С помощью средств *Reverse Engineering* создать *UML*-диаграммы классов и зависимостей классов для разработанного пакета figures.

Внимание! Если Вы увидите ошибку на нашем сайте, выделите её и нажмите Ctrl+Enter.