Лабораторная работа N 1

Разработка простой программы на Java в среде Eclipse

1. Введение

Создание языка Java - это один из самых значительных шагов вперед в области разработки сред программирования. Язык HTML (Hypertext Markup Language - язык разметки гипертекста) был необходим для статического размещения страниц во WWW (World Wide Web). Язык Java потребовался для качественного скачка в создании интерактивных продуктов для сети Internet.

Три ключевых элемента объединились в технологии языка Java и сделали ее в корне отличной от всего, существующего на сегодняшний день.

1. Java предоставляет для широкого использования свои **апплеты** (applets) - небольшие, надежные, динамичные, не зависящие от платформы активные сетевые приложения, встраиваемые в страницы Web. Апплеты Java могут настраиваться и распространяться потребителям с такой же легкостью, как любые документы HTML.

2. Java высвобождает мощь объектно-ориентированной разработки приложений, сочетая простой и знакомый синтаксис с надежной и удобной в работе средой разработки. Это позволяет широкому кругу программистов быстро создавать новые программы и новые апплеты.

3. Java предоставляет программисту богатый набор классов объектов для ясного абстрагирования многих системных функций, используемых при работе с окнами, сетью и для ввода-вывода. Ключевая черта этих классов заключается в том, что они обеспечивают создание независимых от используемой платформы абстракций для широкого спектра системных интерфейсов.

Целью лабораторной работы является написание простого класса на языке Java и получение практических навыков при работе с средой разработки IDE Eclipse, изучение механизма рефакторинга в среде Eclipse.

2. Общие сведения

2.1 Платформа Java.

Язык Java обладает достоинствами с: простоту и мощь, безопасность, объектную ориентированность, надежность, интерактивность, архитектурную независимость, возможность интерпретации, высокую производительность и легкость в изучении. Даже если вы никогда не напишете ни одной строки на языке Java, знать о его возможностях весьма полезно, поскольку именно перечисленные выше свойства языка придают динамику страницам Всемирной паутины.

Простота и мощь.После освоения основных понятий объектно-ориентированного программирования научится программировать на Java довольно легко. В наши дни существует много систем программирования, гордящихся тем, что в них одной и той же цели можно достичь десятком различных способов. В языке Java изобилие решений отсутствует - для решения задачи у вас будет совсем немного вариантов. Стремление к простоте зачастую приводило к созданию неэффективных и невыразительных языков типа командных интерпретаторов. Java к числу таких языков не относится - для Вас вся мощность ООП и библиотек классов.

Безопасность.Один из ключевых принципов разработки языка Java заключался в обеспечении защиты от несанкционированного доступа. Программы на Java не могут вызывать глобальные функции и получать доступ к произвольным системным ресурсам, что обеспечивает в Java уровень безопасности, недоступный для других языков.

Объектная ориентированность. При разработке языка Java отсутствовала тяжелая наследственность, для реализации объектов был избран удобный прагматичный подход. Объектная модель в Java проста и легко расширяется, в то же время, ради повышения производительности, числа и другие простые типы данных Java не являются объектами.

Надежность. Java ограничивает вас в нескольких ключевых областях и таким образом способствует обнаружению ошибок на ранних стадиях разработки программы. В то же время в ней отсутствуют многие источники ошибок, свойственных другим языкам программирования (строгая типизация, например). Большинство используемых сегодня программ "отказывают" в одной из двух ситуаций: при выделении памяти, либо при возникновении исключительных ситуаций. В традиционных средах программирования распределение памяти является заботой программиста Java снимает эти проблемы, используя сборщик мусора для освобождения незанятой памяти и встроенные объектно-ориентированные средства для обработки исключительных ситуаций.

Интерактивность. Java создавалась как средство для создания интерактивных сетевых программ. В Java реализовано несколько интересных решений, позволяющих писать код, который выполняет одновременно массу различных функций и не забывает при этом следить за тем, что и когда должно произойти. Подпроцессы Java дают возможность реализации в программе конкретного поведения, не встраивая глобальной циклической обработки событий.

Независимость от архитектуры ЭВМ. Создатели Java наложили на язык и на среду времени выполнения несколько жестких требований, которые на деле, а не на словах позволяют, однажды написав, всегда запускать программу в любом месте и в любое время (где существует виртуальная Java-машина - броузеры на всех платформах, OS/2, Netware).

Интерпретация плюс высокая производительность. Программы Java транслируются в некое промежуточное представление, называемое байт-кодом (bytecode). Байт-код, в свою очередь, может интерпретироваться в любой системе, в которой есть среда времени выполнения Java. Большинство ранних систем, в которых пытались обеспечить независимость от платформы, обладало огромным недостатком - потерей производительности (Basic, Perl). Несмотря на то, что в Java используется интерпретатор, байт-код легко переводится непосредственно в "родные" машинные коды (Just In Time compilers) "на лету". При этом достигается очень высокая производительность (Symantec JIT встроен в Netscape Navigator).

Простота изучения. Язык Java, хотя и более сложный чем языки командных интерпретаторов, все же неизмеримо проще для изучения, чем другие языки программирования, например C++.

Богатая объектная среда. В Среду Java встроен набор ключевых классов, содержащих основные абстракции реального мира, с которым придется иметь дело вашим программам. Основой популярности Java являются встроенные классы-абстракции, сделавшие его языком, действительно независимым от платформы. Библиотеки, подобные MFC/COM, OWL, VCL, NeXTStep, Motif и OpenDoc прекрасно работают на своих платформах, однако сегодня главной платформой становится Internet.

2.1.1 Первая программа. “Hello World”

Программы, созданные на языке программирования Java, подразделяются по своему назначению на две группы.

К первой группе относятся приложения Java, предназначенные для локальной работы под управлением интерпретатора (виртуальной машины) Java.

Вторую группу программ называют апплетами (aplets). Апплеты представляют собой небольшие специальные программы, находящиеся на удаленном компьютере в сети, с которым пользователи соединяются с помощью браузера. Апплеты загружаются в браузер пользователя и интерпретируются виртуальной машиной Java, встроенной практически во все современные браузеры.

Приложения, относящиеся к первой группе, представляют собой обычные локальные приложения. Поскольку они выполняются интерпретатором и не содержат машинного кода то их производительность заметно ниже, чем у обычных компилируемых программ (C++, Delphi).

Апплеты Java можно встраивать в документы HTML и помещать на Web-сервер. Использование в интернет-страницах Java-апплетов придает динамический и интерактивный характер поведению последних. Апплеты берут на себя сложную локальную обработку данных, полученных от Web-сервера или от локального пользователя. Для более быстрого выполнения апплетов в браузере применяется особый способ компиляции — Just-In-Time compilation (JIT, «на-лету)», что позволяет увеличить скорость выполнения апплета в несколько раз.

Расмотрим простейшую Java-программу (файл Hello.java):

```
public class Hello {

  public static void main(String[] args) {
      System.out.println("Hello World!");
  }

}
```

Откомпилируем программу и запустим ее на выполнение. На экран будет выведено **Hello World!**

Проанализируем данную программу.

- Весь программный код в Java заключен внутри классов. Не может быть никакого программного текста вне класса (или интерфейса).
- Каждый файл с именем **Name.java** должен содержать класс с именем **Name** (причем, учитывается регистр). Каждый `public` -класс с именем **Name** должен быть в своем файле **Name.java** .
- Внутри указанного файла могут быть и другие классы, но их имена должны отличаться от **Name** и они не должны быть `public` .
- Внутри класса может быть конструкция

```
public static void main(String[] args) {
  . . .
}
```

- Это метод класса. Здесь public, static, void — это описатели, main — имя метода.
- Указанный метод main является специальным случаем. При запуске Java-программы мы указываем имя класса. Java-машина ищет этот класс среди всех доступных ей файлов *.class, и в этом классе запускает на выполнение метод `main`.
- Описание метода `main` должно быть в точности таким, как приведено в примере (можно разве что изменить имя `args` на какое-то другое).
- В скобках после имени метода указываются параметры метода. Для `main` -метода параметры должны быть такими как указано. В данном случае `String[]` указывает, что в качестве параметров методу будет передан массив строк. При вызове программы на Java можно задать параметры вызова. Java-машина обработает их и сформирует массив строк, который будет передан в `main` -метод в качестве параметра. Так, если вызвать программу командой

```
j Hello one two 3 4
```

то внутри программы args будет массивом из 4-х элементов

```
args[0] = "one"
args[1] = "two"
args[2] = "3"
args[3] = "4"
```

- Другое дело, что в данном случае параметры вызова никак не используются нашей программой — она их просто игнорирует

2.1.2 Вторая программа. Числа Фиббоначи

Следующий пример выводит числа Фибоначчи, бесконечную последовательность, первые члены которой таковы:

1 1 2 3 5 8 13 21 34

Ряд чисел Фибоначчи начинается с 1 и 1, а каждый последующий его элемент представляет собой сумму двух предыдущих. Программа для печати чисел Фибоначчи несложна, но она демонстрирует объявление переменных, работу простейшего цикла и выполнение арифметических операций

```
class Fibonacci { /** Вывод чисел Фибоначчи < 50 */
    public static void main(String[] args) {
            int lo = 1;
            int hi = 1;
            System.out.println(lo);
            while (hi < 50) {
                    System.out.println(hi);
                    hi = lo + hi; // Изменение значения hi
                    lo = hi – lo; /* Новое значение lo равно старому hi,
                            то есть сумме за вычетом старого lo */
            }
    }
}
```

В этом примере объявляется класс *Fibonacci*, который, как и Hello World, содержит метод *main*. В первых строках метода *main* объявляются и инициализируются две переменные, *hi* и *lo*. Перед именем переменной должен быть указан ее тип. Переменные hi и lo относятся к типу *int* — то есть являются 32-разрядными целыми числами со знаком. В языке Java имеется несколько встроенных, “примитивных” типов данных для работы с целыми, вещественными, логическими и символьными значениями. Java может непосредственно оперировать со значениями, относящимися к примитивным типам, — в отличие от объектов, определяемых программистом. Типы, принимаемые “по умолчанию”, в Java отсутствуют; тип каждой переменной должен быть указан в программе. В Java имеются следующие примитивные типы данных:

Тип данных	Описание
Boolean	одно из двух значений: true или false
char	16-разрядный символ в кодировке Unicode 1.1
byte	8-разрядное целое (со знаком)
short	16-разрядное целое (со знаком)
int	32-разрядное целое (со знаком)
long	64-разрядное целое (со знаком)
float	32-разрядное с плавающей точкой (IEEE 754-1985)
double	64-разрядное с плавающей точкой (IEEE 754-1985)

2.1.3 Третья программа. Простой класс.

Рассмотрим простой класс Movie:

```
public class Movie {

	private String title;
	private String year;

	public Movie(String title, String year) {
		super();
		this.title = title;
		this.year = year;
	}
	public Movie(String year) {
		this.year = year;
	}
	public Movie(String filmName, int filmYear) {
		super();
		this.title = filmName;
		this.year = Integer.toString(filmYear);
	}
	public String getTitle() {
		return title;
	}

	public void setTitle(String title) {
		this.title = title;
	}
	public String getYear() {
		return year;
	}
	public void setYear(String filmYear) {
		this.year = filmYear;
	}
	public static void main(String[] args) {
	//
		Movie movie = new Movie("add",1999);
		System.out.println(movie.getTitle());
		System.out.println(movie.getYear());

	}
	@Override
	protected void finalize() throws Throwable {
		// TODO Auto-generated method stub
		super.finalize();
	}
	@Override
	public String toString() {
		// TODO Auto-generated method stub
		return title+year;
	}
}
```

В классе имеются две приватные переменные *title*- заголовок фильма и *year*- год выпуска, для которых имеются методы -акцессоры В данном классе созданы 3 конструктора, перегружены методы *toString* и *finalize*. В методе *main* создается объект класса *Movie* и происходит вывод данных на экран. Значительная часть кода данного класса создана автоматически, при помощи кодогенератора и рефакторинга среды Eclipse.

2.2 Основы работы с IDE Eclipse

Eclipse — один из лучших инструментов Java, созданных за последние годы. SDK Eclipse представляет собой интегрированную среду разработки (IDE, Integrated Development Environment) с открытым исходным кодом.

В начале своего существования Eclipse появилась как коммерческий продукт, но в ноябре 2001 г. его исходные коды были опубликованы. Создателем системы является компания Object Technology International (OTI), которая впоследствии была приобретена корпорацией IBM. Начиная с 2001 г. Eclipse была загружена более 50 миллионов раз и в настоящее время используется десятками тысяч программистов по всему миру. Поддержкой и разработкой Eclipse в настоящее время занимается организация Eclipse Foundation и сообщество Eclipse, информацию о которых можно найти на официальном сайте в сети Интернет http://www.eclipse.org.

Основные инструментальные средства Eclipse Java включают в себя: инкрементальный компилятор, редактор Java с подсветкой синтаксиса, контекстным автозавершением и поддержкой шаблонов, отладчик, поддержку автоматического рефакторинга и навигации по исходному коду. Кроме этого в Eclipse доступны множество бесплатных и коммерческих дополнений (плагинов), таких, как инструментальные средства создания схем UML,разработка баз данных и др.

Собственно сама по себе Eclipse — это только платформа, которая предоставляет возможность разрабатывать дополнения, называемые плагинами, которые естественным образом встраиваются в платформу. В Eclipse доступны дополнения для следующих языков: C/C++, Html, Cobol, Perl, Php, Ruby и др. Все плагины Eclipse используют общие концепции, интерфейсы и технические решения.Вы можете также разработать собственное дополнение для расширения возможностей Eclipse.

2.2.1 Создание нового проекта.

Чтобы создать Java-проект, запустите Eclipse и выберите **File->New->Project...** В левом списке появившегося мастера выберите **Java Project**. Затем нажимите кнопку **Next**..

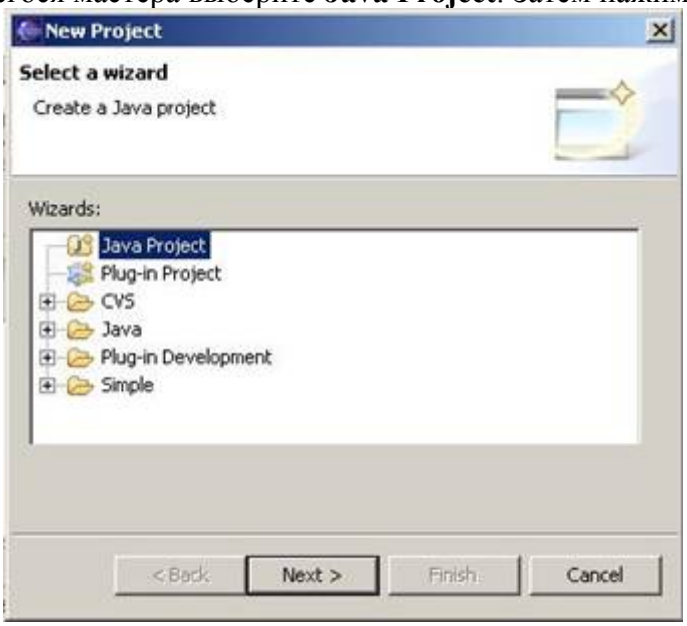


Рисунок 3. Первая страница мастера Нового Проекта

Введите *название проекта* и снова нажмите **Next**. Можно указать место для хранения исходных и скомпилированных файлов, а также создать отдельные каталоги для таких файлов.

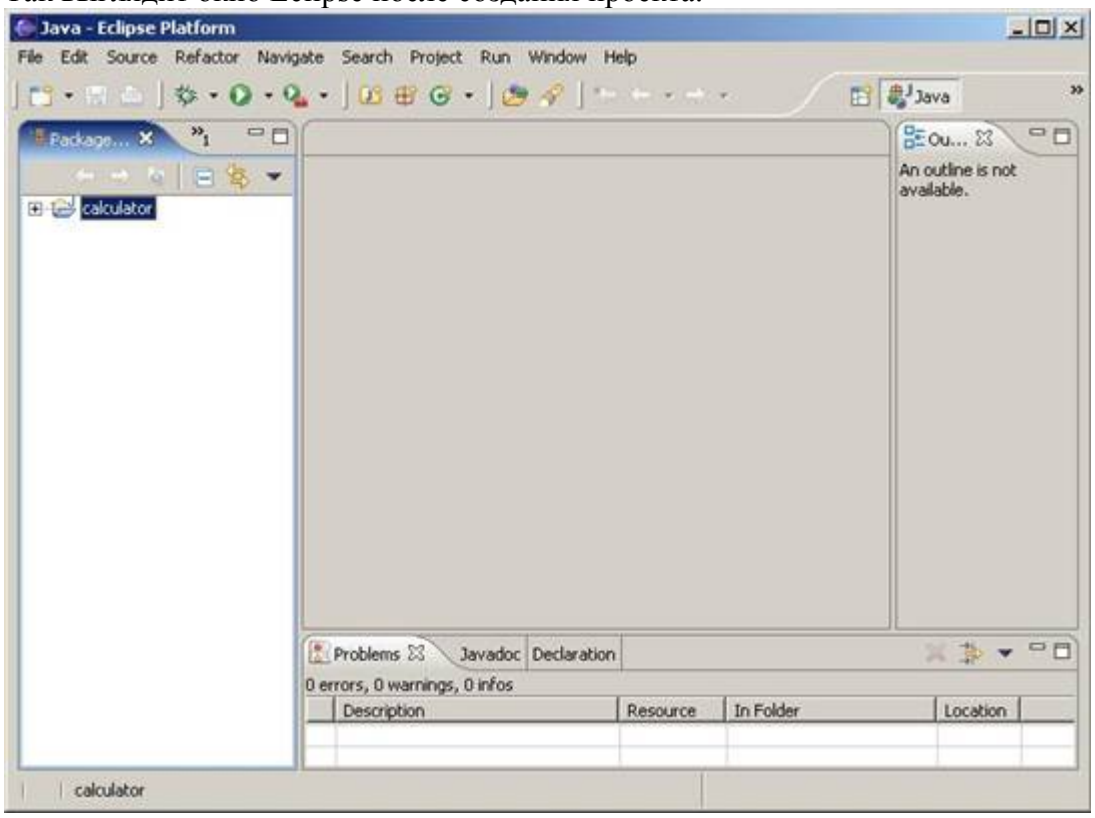
Рисунок 4. Вторая страница мастера Нового Проекта

Последний шаг мастера позволяет указать место для хранения исходных и скомпилированных файлов, а также задать любые подпроекты и библиотеки, которые могут понадобиться для работы и компоновки текущего проекта.

Рисунок 5. Третья страница мастера Нового Проекта

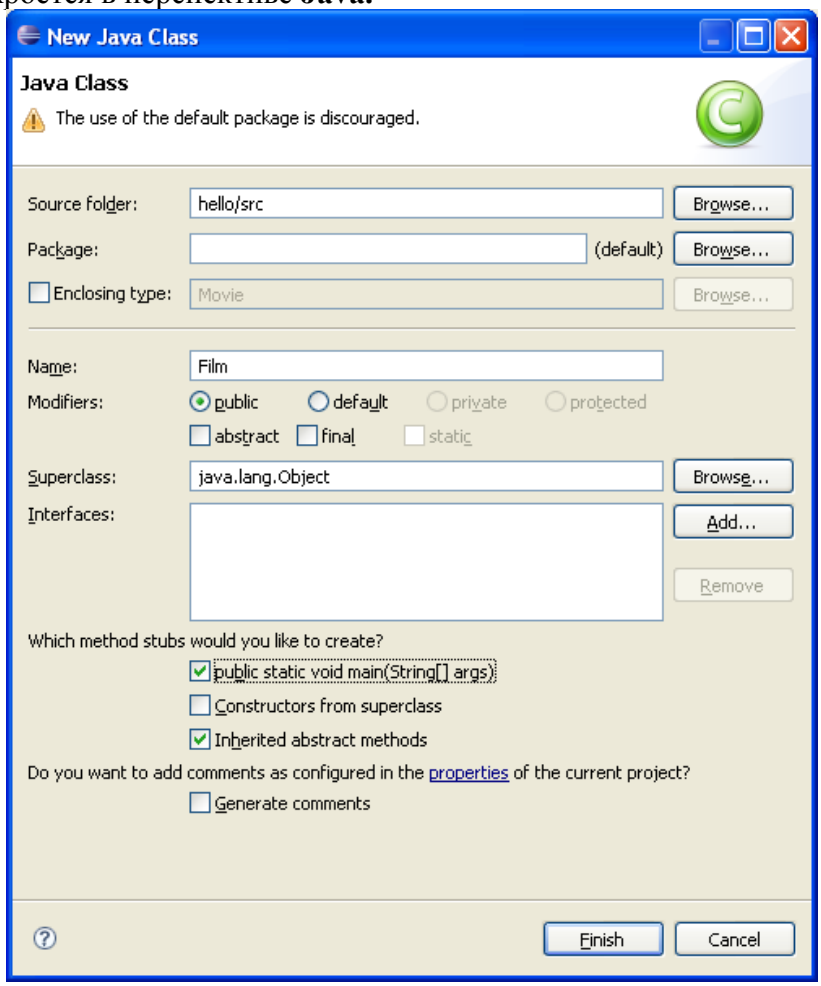
Щелкните на кнопке **Finish.** В появившемся окошке предлагающем переключить перспективу жмем на YES и Eclipse создаст новый проект

Так выглядит окно Eclipse после создания проекта.



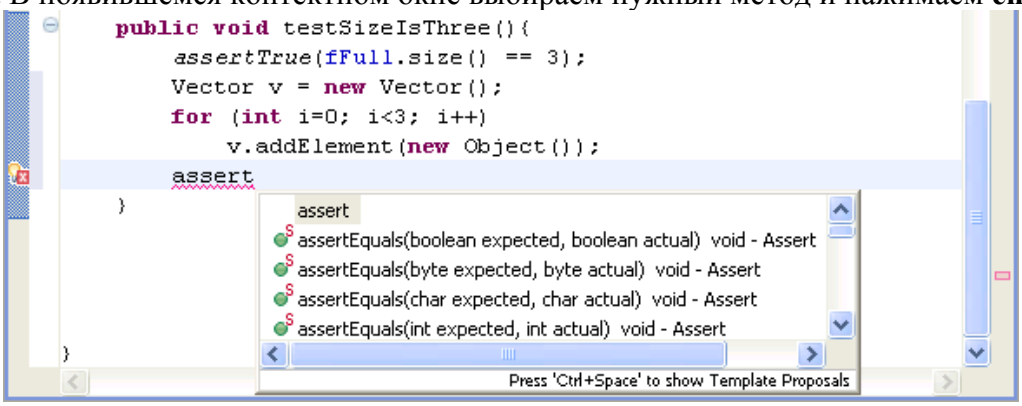
2.2.2. Создание нового класса

Для создание нового класса выберите **File -> New -> Class.** В появившемся окне нужно указать **Имя класса (Name), Папку (Source folder)** в которой он будет находится, а также **класс-предок**(**Superclass**), от которого данный класс наследуется**.** (По умолчанию это будет класс **Object**). Если требуется,чтобы в класс был вставлен метод **main** нужно отметить соответствующую галочку. После нажатия на кнопку **Finish** будет создан новый класс который откроется в перспективе **Java.**



2.2.3 Использование контентой помощи (Content assist)

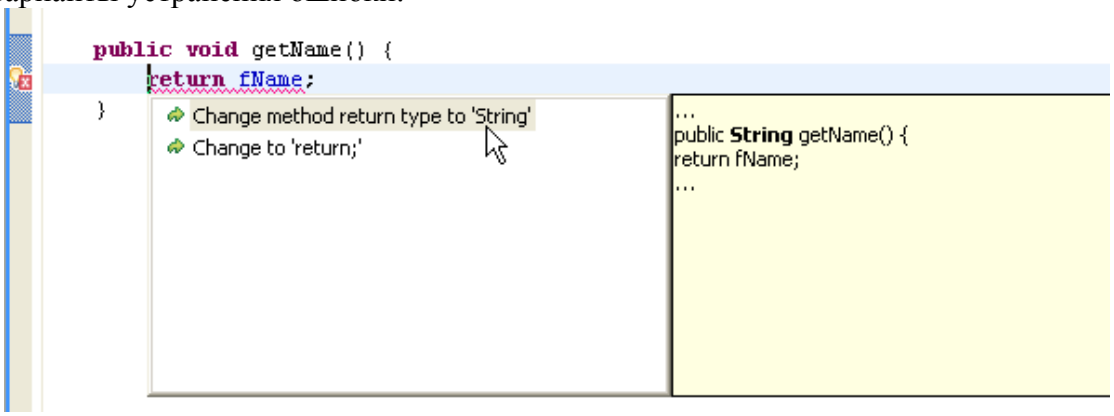
Предположим у нас есть фрагмент кода и мы хотим вставить в него метод assertTrue, но не помним точно его название и параметры. Набираем в редакторе *assist* и нажимаем **Ctrl-space.** В появившемся контектном окне выбираем нужный метод и нажимаем **enter**



При помощи ассистента можно ввести также и параметры метода.

2.2.4 Устранение ошибок «на лету» (Quick fix)

Редактор Eclipse позволяет быстро устранять ошибки копиляции, которые обозначаются лампочкой с крестиком слева от проблемы. Для устранение проблемы выберите строку, где она появилась и нажмите **Ctrl+1 (Edit > Quick Fix).** Среда покажет вам варианты устранения ошибки.



Выберите нужный вам вариант устранения ошибки и нажмите **Enter**.

2.3 Рефакторинг

Рефакторинг - изменение структуры программы без изменения ее функциональности. Рефакторинг - мощное средство, но использовать его следует внимательно, поскольку он может явится причиной ошибок.

Причины по которым код программы может быть подвергнут рефакторингу:

1. Героическая.

Причина: Старый код требует поддержки, а команды его разработчиков уже не существует. Должна быть создана новая версия с новыми возможностями, но код уже непостижим.

Решение: Новая команда разработчиков перелопачивает код и создает новую версию. Это - рефакторинг в героическом масштабе.

2. Появление новых требований.

Причина:В проекте появляются новые требования, которые не учтены в исходном плане или же по причине использования итеративного подхода.

Решение: Изменение иерархии классов, введение интерфейсов или абстрактных классов, расщепление классов, переупорядочивание классов и т.д.

3. Простой способ автоматической генерации кода.

Причина: необходимость освободить разработчика от рутинной работы (переименование данных, выделение методов и т.д.)

Решение: Создание инструментов автоматического рефакторинга, позволяющих экономить время и освобождать от рутины.

Мы будем рассматривать рефакторинг преимущественно относительно третьей причины.

Но рефакторинг сам по себе опасен, поскольку может служить источником ошибок. Как быть? Есть **два способа уменьшения риска испортить код**.

1. Использовать *сквозной набор модульных тестов* (код должен пройти тесты перед и после рефакторинга).

2. Использовать *автоматизированный инструмент рефакторинга* (таковой есть в среде Eclipse).

Таким образом, комбинация сквозного тестирования и автоматического рефакторинга позволяет быстро и безопасно изменять структуру кода с тем, чтобы добавить новую функциональность или улучшить его сопровождаемость.

2.3.1 Типы рефакторинга в Eclipse

Инструменты **рефакторинга** (Alt+Shift+T) в Eclipse можно сгруппировать по трем основным категориям:

1. Переименование полей, переменных, классов и интерфейсов и перемещение пакетов и классов.

2. Переопределение отношений классов, включая превращение анонимных классов во вложенные классы, превращение вложенных классов в классы верхнего уровня создание интерфейсов из определенных классов и перемещение методов или полей из класса в подкласс или суперкласс.

3. Превращение локальных переменных в поля класса, превращение выбранного из метода кода в отдельный метод и генерация методов-акцессоров для полей.

2.3.2. Переименование.

Чтобы переименовать элемент Java, щелкните на нем в представлении **Package Explorer** или выберите его в исходном файле Java, а затем **Refactor -> Rename** (**Alt+Shift+R**). В диалоговом окне выберите новое имя и выберите, должен ли Eclipse также изменить ссылки на имя (Update references…).

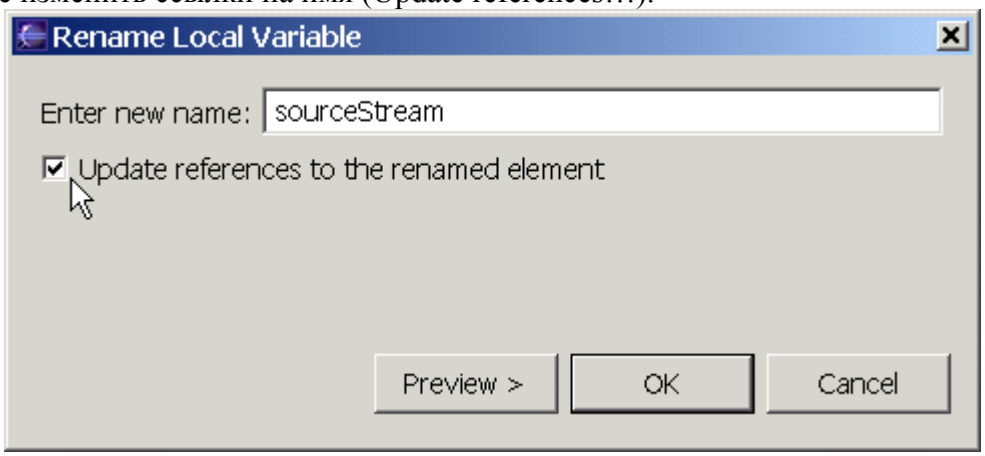


Рисунок 1. Переименование локальной переменной

Кнопка **Preview** используется чтобы увидеть изменения, которые Eclipse предлагает сделать. Если вы доверяете способности Eclipse выполнить изменения корректно, вы можете вместо этого нажать **OK**.

2.3.3. Переопределение отношений классов

Переопределение отношений классов включает в себя: продвижение анонимных и вложенных классов, перемещение членов в иерархии классов, выделение интерфейсов, использование супертипа и т.д. Для заинтересованных лиц см. David Gallardo Рефакторинг для всех [X].

2.3.4. Выделение и встраивание

Выделение метода. (Extract Method)

Рассмотрим, метод *main()* в классе *StartApp*. Он вычисляет опции командной строки и если находит опцию, начинающуюся с *-D*, сохраняет ее пару имя-значение в объекте *Properties***.**

Выделение метода предполагает изъятие части кода и помещение его в отдельный метод. Есть 2 случая когда это необходимо:

1. Метод слишком длинный и выполняет слишком много логически различных операций.

2. Есть логически обособленная секция метода, которая может быть повторно использована в других методах.

Выделим секцию кода, которая включает в себя объявление *StringTokenizer* и следующий за ним оператор *if*. Чтобы сделать это, выделите этот код, а затем выберите из меню **Refactor > Extract Method**(**Alt+Shift+M**). Вы получите приглашение на ввод имени метода; введите *addProperty*, и вот результат:

Листинг X.

До рефакторинга	После рефакторинга
import java.util.Properties; **import java.util.StringTokenizer;** **public class StartApp** **{** ** public static void main(String[] args)** ** {** ** Properties props = new Properties();** ** for (int i= 0; i < args.length; i++)** ** {** ** if(args[i].startsWith("-D"))** ** {** ** String s = args[i].substring(2);** ** StringTokenizer st =** ** new StringTokenizer (s, "=");** ** if(st.countTokens() == 2)** ** {** ** props.setProperty(st.nextToken(),** ** st.nextToken());** ** }** ** }** ** }** ** //continue...** ** }** **}**	**import java.util.Properties;** **import java.util.StringTokenizer;** **public class Extract** **{** ** public static void main(String[] args)** ** {** ** Properties props = new Properties();** ** for (int i = 0; i < args.length; i++)** ** {** ** if (args[i].startsWith("-D"))** ** {** ** String s = args[i].substring(2);** ** addProp(props, s);** ** }** ** }** ** }** ** private static void addProp(Properties props,** ** String s)** ** {** ** StringTokenizer st =** ** new StringTokenizer(s, "=");** ** if (st.countTokens() == 2)** ** {** ** props.setProperty(st.nextToken(),** ** st.nextToken());** ** }** ** }** **}**

Выделение переменной.

Операция рефакторинга **Extract Local Variable** берет выражение, которое используется непосредственно, и сначала присваивает его значение локальной переменной. Эта переменная затем используется там, где использовалось выражение.

В методе *addProp()* выделим первый вызов *st.nextToken()*. Выбрать **Refactor > Extract Local Variable (Alt+Shift+L)**, введем имя переменной - *key*. Вот результат.

Листинг X.

До рефакторинга	После рефакторинга
private static void addProp(Properties props, ** String s)** ** {** ** StringTokenizer st =** ** new StringTokenizer(s, "=");** ** if (st.countTokens() == 2)** ** {** ** props.setProperty(st.nextToken(),** ** st.nextToken());** ** }** ** }**	**private static void addProp(Properties props,** ** String s)** ** {** ** StringTokenizer st =** ** new StringTokenizer(s, "=");** ** if(st.countTokens() == 2)** ** {** ** String key = st.nextToken();** ** String value = st.nextToken();** ** props.setProperty(key, value);** ** }** ** }**

Введение переменных таким способом обеспечивает ряд преимуществ.

1. Введение осмысленных имен для выражений делает код более ясным.
2. Облегчается отладка кода, потому что легче инспектировать переменную, чем результат выражения.

Выделение константы

Операция Extract Constant аналогична Extract Local Variable. Нужно выбрать статическое, константное выражение, которое рефакторинг преобразует в константу *static final*. Это полезно для удаления из кода жестко закодированных чисел и строк.

Листинг X.

До рефакторинга

```
public class Extract
{
  public static void main(String[] args)
  {
    Properties props = new Properties();
    for (int i = 0; i < args.length; i++)
    {
      if (args[i].startsWith("-D"))
      {
        String s = args[i].substring(2);
        addProp(props, s);
      }
    }
  }
//…
```

После рефакторинга

```
public class Extract
{
  private static final String DEFINE = "-D";
  public static void main(String[] args)
  {
    Properties props = new Properties();
    for (int i = 0; i < args.length; i++)
    {
      if (args[i].startsWith(DEFINE))
      {
        String s = args[i].substring(2);
        addProp(props, s);
      }
    }
  }
  // ...
```

Обратный рефакторинг (операция Inline…)

Для каждой операции рефакторинга **Extract...** есть соответствующая операция рефакторинга **Refactor -> Inline...**, которая выполняет обратную операцию.

Листинг X.

После рефакторинга

```
if (args[i].startsWith(DEFINE))
      {
        String s = args[i].substring(2);
        addProp(props, s);
      }
```

Обратный рефакторинг

```
if(args[i].startsWith(DEFINE))
      {
        addProp(props,args[i].substring(2));
      }
```

Изменение сигнатуры метода:

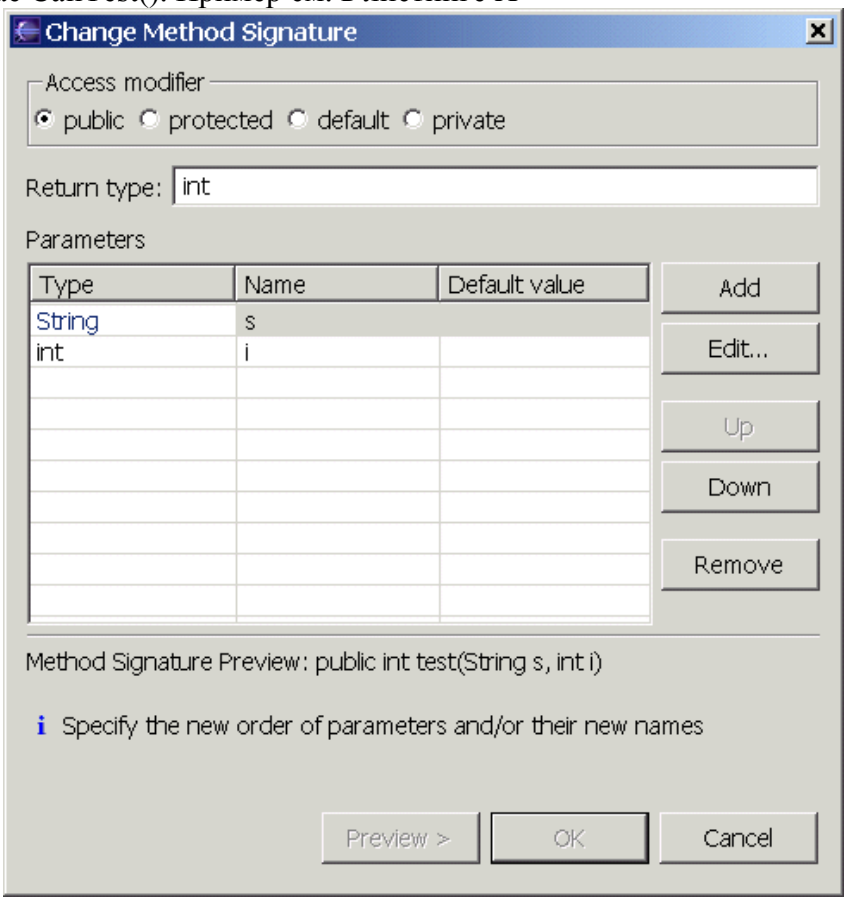
Данная операция **Change Method Signature (Alt+Shift+C)**, является наиболее трудной в использовании, поскольку изменяет параметры, видимость и возвращаемый тип метода. Если изменения вызывают проблемы в коде - операции рефакторинга укажут на это. Вы имеете выбор либо принять рефакторинг везде и откорректировать проблемы после этого, либо отменить рефакторинг. Если рефакторинг порождает проблемы в других методах, они игнорируются, и придется исправлять их вручную.

Изменим сигнатуру метода *test()* в класса *MethodSigExample.* Для чего выделим *test()* и выберем **Refactor > Change Method Signature**. Появится диалог, в котором можно изменят:

1. Видимость метода. Если изменить видимость на protected или private то метод callTest() из второго класса не увидить метод test(). Причем Eclipse не отметит эту ошибку при выполнении рефакторинга – нужно будет править вручную.

2. Возвращаемый тип. Изменение int на float, не отмечается как ошибка, поскольку int в операторе return метода test() автоматически превращается в float. Однако это вызовет проблемы в методе callTest(), поскольку float не может быть преобразовано в int.

3. Изменение параметров. При изменении типа первого параметра из String в int возникнут проблемы и в test() и в CallTest(). Если удаляется параметр i, то нужно начать с удаления ссылок не него в методе test(). Тогда удаление параметра пройдет более гладко. Наконец, при добавлении параметра используется опция Default Value. Она обеспечивает значение добавляемого параметра по умолчанию для вызывающих методов, в данном случае CallTest(). Пример см. в листинге X



Листинг X.

До рефакторинга **После рефакторинга**

Объявление класса	**Объявление класса**
public class MethodSigExample **{** **public int test(String s, int i)** **{** **int x = i + s.length();** **return x;** **}** **}**	**public class MethodSigExample** **{** **public int test(String s, int i, String w)** **{** **int x = i + s.length();** **return x;** **}** **}**
Использование объекта	**Использование объекта**

public void callTest() **{** **MethodSigExample eg =** **new MethodSigExample();** **int r = eg.test("hello", 10);** **}**	**public void callTest()** **{** **MethodSigExample eg =** **new MethodSigExample();** **int r = eg.test("hello", 10, "world");** **}**

2.3.5 Кодогенерация

Наряду с рефакторингом среда Eclipse поддерживает различные механизмы **кодогенерации (Alt+Shift+S)**, в число которых входит генерация методов-акцессоров, конструкторов, перегрузка методов и т.д.

Генерация методов-акцессоров и инкапсуляция полей.

Внутренная структура объектов должна быть скрыта. Чтобы получить доступ к таким полям нужны открытые методы-акцессоры. Эти методы могут быть сгенерированы автоматически двумя разными способами.

1. **Source > Generate Getter and Setter**. **(Alt+Shift+S)** – генерация методов-акцессоров для каждого поля, которое еще их не имеет. Это не рефакторинг, поскольку при этом ссылки на поля не обновляются использованием новых методов,и вам нужно сделать это самому. Лучше использовать при создании класса или при добавлении новых полей.

2. Выбрать поле и Refactor > Encapsulate Field. Этот способ генерирует методы-акцесоры только для одного поля за раз, но, в отличие от **Source > Generate Getter and Setter,** он также меняет ссылки на поле при вызове новых методов.

Выделим имя поля и выбрав **Refactor > Encapsulate Field**. В диалоге введем имена для методов-акцессоров: по умолчанию *getMake()* и *setMake()*.

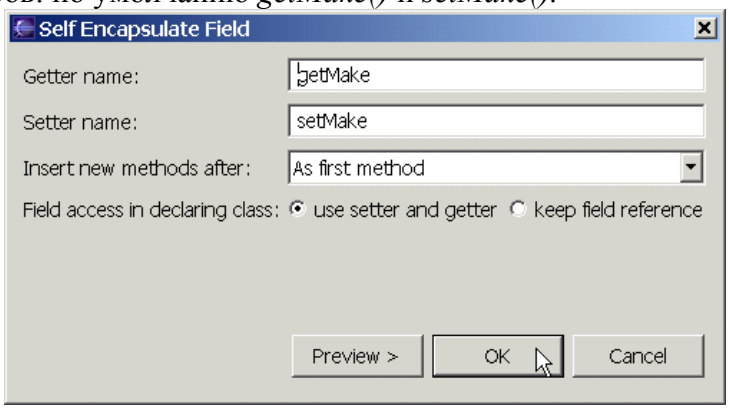


Рисунок X

Листинг X.

До рефакторинга	**После рефакторинга**
Объявление класса	**Объявление класса**

public class Automobile extends Vehicle
{
  public String make;
  public String model;
}
	public class Automobile extends Vehicle
	{
	  private String make;
	  public String model;
	  public void setMake(String make)
	  {
	  this.make = make;
	  }
	  public String getMake()
	  {
	  return make;
	  }
	}

Использование объекта	**Использование объекта**

public class AutomobileTest
{
  public void race()
  {
  Automobilecar1 = new Automobile();
  car1.make= "Austin Healy";
  car1.model= "Sprite";
  // ...
  }
}
	public class AutomobileTest
	{
	  public void race()
	  {
	  Automobilecar1 = new Automobile();
	  car1.setMake("Austin Healy");
	  car1.model= "Sprite";
	  // ...
	  }
	}

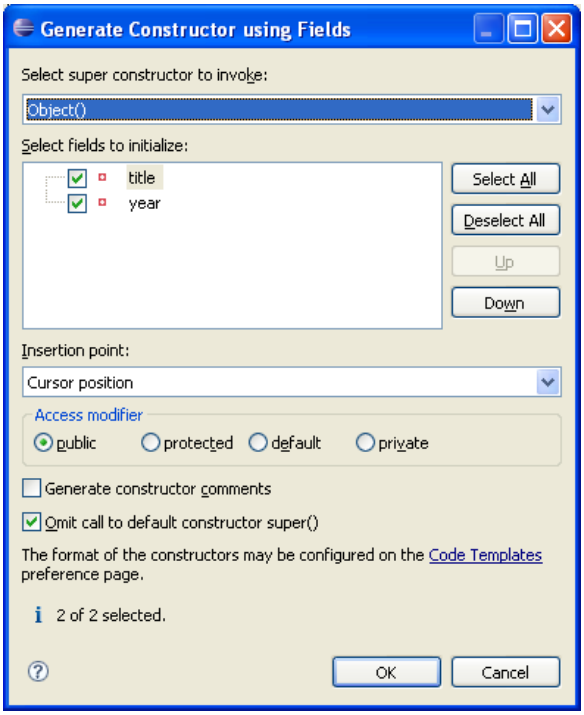
Генерация конструктора.

Данная операция не является в прямом смысле рефакторингом, а скорее кодогенерацией, поскольку добавляет новую функциональность. Выберите **Source - > Generate Constructor using fields.** В появившемся диалоге выберите поля по которым будет сгененирован конструктор, укажите его видимость, а также запрет вызова конструктора базового класса **(Omit call…)** если это необходимо.

Листинг X.

До рефакторинга	После рефакторинга

public class Movie
{
  private String title;
  private String year;
  …
}
	public class Movie
	{
	  private String title;
	  private String year;
	  public Movie(String title, String year)
	  {
	  super();
	  this.title = title;
	  this.year = year;
	  }
	….
	}



Перегрузка метода

Наряду с добавлением новых методов, можно перегрузить уже существующие. Выберем **Source - >Override/Implement method.** В появившемся окне галочкой отметим метод *finalize()* и нажмем OK. В результате получим перегруженный деструктор.

До рефакторинга	После рефакторинга
public class Movie **{** **private String title;** **private String year;** **…** **}**	**public class Movie** **{** **private String title;** **private String year;** **@Override** **protected void finalize() throws Throwable** **{** **// TODO Auto-generated method stub** **super.finalize();** **}** **….** **}**

Итак, рефакторинг, наряду с кодогенерацией позволяют существенно повысить производительность программиста при написании кода, а инструменты Eclipse делают это легким, экономя время при выполнении типовых операций изменения кода и избавляя программиста от рутины. Затратив немного времени сейчас на знакомство с этими инструментами, можно существенно съэкономить его в дальнейшем.

3. Дополнительная литература

Java

1. **Мультимедийный Обучающий Курс TeachPro Java Для Начинающих**
2. Картузов А.В. **Программирование на языке JAVA**
3.Герберт Шилдт, Джеймс Холмс **Искусство программирования на JAVA**.
4. Патрик Нотон, Герберт Шилдт **Полный справочник по Java**.
5. Вязовик Н.А. **Программирование на JAVA. Курс лекций на intuit.ru**

Eclipse

1. Давид Галлардо. **Рефакторинг для всех.**
2. IBM Corp. **Руководство Eclipse JDT. Начало работы**
3. IBM Corp. **Руководство Eclipse Workbench. Начало работы**

4. Порядок выполнения работы

В соответствии с **вариантом** выполните следующее основное задание:

1. Создайте новый проект в среде Eclipse.
2. Создайте новый **класс**, предусмотрев в нем точку входа (main).
2. Создать **приватные поля** указанные в задании
3. Создать методы акцессоры для приватных полей getters & setters
4. Сгенерировать 3 конструктора на основе полей (Без параметров, с 1 параметором, с 2 параметрами)
5. Создать метод**,** осуществляющий **вычисление функции**, в соответствии с вариантом, который возвращает результат вычислений.
5. Создать объекта класса (не создавая переменной) и вывести результат функции вычисления.
7. Выделить переменную класса (из объекта созданного в п. 5).
6. Изменить тип первого параметра конструктора с String на int и обеспечить преобразование данных внутри метода.
8. Выделить метод класса.
9. Выделить константу.
10. Вывести поля класса.
11. Переопределить метод toString()
12. Переименовать приватные переменные. Getters & setters также должны быть переименованы.
13. Переименовать класс.
14. Подготовьте отчет о выполнении лабораторной работы:

Для успешной сдачи лабораторной работы необходимо:

– представить преподавателю отлаженный код программы для указанного варианта задания;
– продемострировать уверенную работу с механизмом рефакторинга (кодогенерации) **в** среде IDE Eclipse;
– подготовить отчет по работе.

5. Порядок оформления отчета

Отчет о выполнении лабораторной работы должен содержать:

1) титульный лист;
2) задание;
3) текст программы;
4) результаты работы программы.

6. Варианты заданий

№ вар-та	*Класс*	*Приватные поля класса*	*Вычисляемая функция*	*Примечание*
1	Треугольник	Высота(h) Основание(a)	Вычислить площадь	
2	Прямоугольник	Стороны(a, b)	Вычислить площадь	
3	Конус	Высота(h) Радиус основания(к)	Вычислить объем	
4	Цилиндр	Высота(h) Радиус основания(к)	Вычислить объем	
5	Температура	по Цельсию(c) по Фаренгейту (f)	Преобразования шкалы Цельсия в шкалу Фаренгейта и наоборот	
6	Степень числа	База(x) основание(n)	Вычислить x^n	
7	Квадратное уравнение	Коэффициенты(a,b,c)	Вычислить корни квадратного уравнения	
8	Калькулятор	числа(a,b)	Реализовать операции +, -, *, /.	
9	Комплексные числа	Вещ. часть(RE) Мнимая (IM)	Реализовать операции над комплекными числами +, -, *.	
10	НОД	Целые числа(a,b)	Вычислить НОД двух целых чисел по алгоритму Евклида	НОД - наибольший общий делитель
11	Арифметическая прогрессия	Первый член(a) Коэффициент приращения (k)	Вычислить сумму первых n членов прогрессии	
12	Геометрическая прогрессия	Первый член(a) Коэффициент приращения (q)	Вычислить сумму первых n членов прогрессии	
13	Прямоугольный треугольник	катеты(a,b)	Вычислить гипотенузу	По теореме Пифагора
14	Трапеция	Основания(a, b) Высота(h)	Вычислить площадь	
15	Товар	Цена (p) Количество(c) НДС в %(n)	Вычислить стоимость товара с учетом НДС	НДС может изменятся