ПРИЛОЖЕНИЕ 2

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

"НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ"

«УТВЕРЖДАЮ»

Проректор по учебной работе

_____САБЛИНА С.Г.

«__» _____ 20__ г

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС
КУРС 4-Й, VII СЕМЕСТР

«Информационные технологии и языки программирования»

Кафедра информационной биологии

Новосибирск
2012 г.

Учебный курс ориентирован на студентов IV курса факультета естественных наук, специальность «биология». В состав пособия включены: программа курса лекций, структура курса, приведены примеры контрольных вопросов для тестирования по материалам лекций, даны примеры задач и вопросов на контрольных работах и зачёте, а также методические указания и примеры решения типовых задач.

Составитель

Лашин С.А.

Учебно-методический комплекс подготовлен в рамках реализации Программы развития НИУ-НГУ

Оглавление

Аннотация рабочей программы

Дисциплина «Информационные технологии и языки программирования» является базовой дисциплиной изучения программирования по направлению подготовки «Информационная биология», профиль «**Биология**». Дисциплина реализуется на Факультете естественных наук Национального исследовательского университета Новосибирский государственный университет кафедрой информационной биологии ФЕН НИУ НГУ.

Содержание дисциплины охватывает круг вопросов, связанных с объектно-ориентированным программированием и проектированием, а также с применением программирования в задачах биоинформатики.

Дисциплина нацелена на формирование общекультурных компетенций ОК-3, ОК-6, ОК-9, ОК-12, ОК-13 ОК-14, ОК-15, ОК-16, ОК-18; профессиональных компетенций ПК-1, ПК-4, ПК-6, ПК-7, ПК-16, ПК-17, ПК-19 выпускника.

Преподавание дисциплины предусматривает следующие формы организации учебного процесса: лекции, практические занятия, контрольная работа, домашние задания, консультации, самостоятельная работа студента.

Программой дисциплины предусмотрены следующие виды контроля: текущий контроль успеваемости в форме контрольной работы, промежуточный контроль в форме зачета. Формы рубежного контроля определяются решениями Ученого совета, действующими в течение текущего учебного года. Результатом прохождения дисциплины является итоговая оценка по пятибалльной шкале (дифференцированный зачёт).

Программой дисциплины предусмотрены следующие виды контроля:

Текущий контроль. Формой текущего контроля при прохождении дисциплины «Информационные технологии и языки программирования» является контроль посещаемости занятий, сдача домашних заданий и написание контрольных работ.

Для того, чтобы быть допущенным зачёту, студент должен выполнить следующее:

- в ходе обучения посетить не менее 70 % занятий;
- сдать все домашние задания в виде работающих компьютерных программ;
- написать на положительные оценки две контрольные работы.

В случае отсутствия на контрольной работе по уважительной причине (наличие медицинской справки) контрольную работу можно переписать в течение недели от окончания срока действия справки.

В зависимости от результатов работы в течение семестра, студент имеет право на получение оценки без экзамена (оценки-«автомата»). Для этого он должен:

- в ходе прохождения дисциплины посетить не менее 70 % занятий;
- написать две контрольных работы на оценку не ниже «хорошо»;
- написать и сдать все компьютерные программы, заданные в течение семестра в срок до последнего занятия;

либо:

- по согласованию с лектором выбрать и самостоятельно решить задачу из категории «сложных» (написать работающую компьютерную программу);
- выступить на одном из практических занятий с кратким докладом о проделанной работе: «защитить» свою программу.

Оценка-«автомат» выводится как средневзвешенная из полученных студентом баллов по результатам работы в семестре.

Итоговый контроль**.** Итоговую оценку за семестр студент может получить на дифференцированном зачёте в конце семестра, где студент имеет возможность либо повысить оценку, полученную им «автоматом», либо получить любую положительную (или неудовлетворительную) оценку в случае отсутствия у него «оценки-автомата» по результатам работы в семестре.

Общая трудоемкость дисциплины составляет 1,5 зачётных единицы, 54 академических часа. Программой дисциплины предусмотрены 16 часов лекционных и 16 часов практических занятий. Остальное время – самостоятельная работа студентов и контроль в форме контрольной и зачета.

1. Цели освоения дисциплины

Дисциплина «Информационные технологии и языки программирования» ставит своей целью усвоение студентами понятий, связанных с разработкой программного обеспечения в области биологии, и развивает базовые навыки в программировании на примере языка, поддерживающего объектно-ориентированную парадигму, языка Java.

В первой части, данный курс знакомит студентов с историей развития программирования, с современными парадигмами программирования, в частности, с понятиями объектно-ориентированного программирования и проектирования, а также с современными средствами разработки программного обеспечения. Объектно-ориентированный подход изучается на примере языка Java.

Во второй части курса студентами рассматриваются и реализуются типовые задачи биоинформатики, связанные с анализом строковых последовательностей (последовательностей ДНК, РНК и белков).

В заключительной части курса студенты выполняют самостоятельное задание, связанное с написание программ анализа биологических данных, содержащихся в базах данных (например, GenBank).

Основной целью освоения дисциплины является получение студентами навыков построения и анализа методов и алгоритмов, а также навыков разработки и использования программных средств для анализа биологических данных, таких как геномные и протеомные последовательности, данные о воздействии различных веществ (включая лекарственные) на протекание биохимических процессов в организме и т.д. Особое внимание уделено изучению форматов представления данных генетики, молекулярной биологии и биомедицины и изучению базовых алгоритмов их обработки.

2. Место дисциплины в структуре ООП

Курс «информационные технологии и языки программирования» является вводным курсом по специальности «информационная биология, биоинформатика». Чтение этого курса происходит в самом начале специализации студентов по указанной по специальности.

Дисциплина «информационные технологии и языки программирования» опирается на следующие дисциплины:

- Основы компьютерной грамотности;
- Математический анализ;
- Математическая статистика и теория вероятностей.

Результаты освоения дисциплины «информационные технологии и языки программирования» используются в следующих дисциплинах данной ООП:

- Системная компьютерная биология II: математическое моделирование

3. Компетенции обучающегося, формируемые в результате освоения дисциплины «Информационные технологии и языки программирования»:

Общекультурные компетенции:

- приобретение новых знаний и формирование суждений по научным, социальным и другим проблемам, с использованием полученного базового образования, а также современных образовательных и информационных технологий **(ОК-3)**;
- использование в познавательной и профессиональной деятельности базовых знаний в области математики и естественных наук, применение методов математического анализа и моделирования, теоретического и экспериментального исследования **(ОК-6)**;
- критический анализ, переоценка своего профессионального и

социального опыта, при необходимости готовность изменить профиль своей профессиональной деятельности **(ОК-9)**;

- использование основных технических средства в профессиональной деятельности: работа на компьютере и в компьютерных сетях, использование универсальные пакеты прикладных компьютерных программ, создание базы данных на основе ресурсов Интернет, способность работать с информацией в глобальных компьютерных сетях **(ОК-12)**;
- способность использовать базовые знания и навыки управления информацией для решения исследовательских профессиональных задач, соблюдение основных требований информационной безопасности, в том числе защиты государственной тайны **(ОК-13)**;
- проявление творческих качеств **(ОК-14)**;
- адекватная постановка цели, проявление целеустремленности в их достижении **(ОК-15)**;
- ответственность за качество выполняемой им работы **(ОК-16)**;
- умение работать самостоятельно и в команде **(ОК-18)**;

Профессиональные компетенции:

- базовые представления о разнообразии биологических объектов, понимание значения биоразнообразия для устойчивости биосферы **(ПК-1)**;
- знание принципов клеточной организации биологических объектов, биофизических и биохимических основ, мембранных процессов и молекулярных механизмов жизнедеятельности **(ПК-4)**;
- базовые представления об основных закономерностях и современных достижениях генетики, о геномике, протеомике **(ПК-6)**;
- понимание роли эволюционной идеи в биологическом мировоззрении; имеет современные представления об основах эволюционной теории, о микро- и макроэволюции **(ПК-7)**;

- применение на практике приемов составления научно-технических отчетов, обзоров, аналитических карт и пояснительных записок **(ПК-16)**;
- понимание, способность излагать и критически анализировать получаемую информацию, способность представлять результаты лабораторных и полевых биологических исследований **(ПК-17)**;
- использование современных методов обработки, анализа и синтеза лабораторной и полевой биологической информации, знание принципов составления научно-технических проектов и отчетов **(ПК-19)**;

В результате освоения дисциплины обучающийся должен:

- иметь представление о современных методах программирования и некоторых методах проектирования программных продуктов,
- знать синтаксис и семантику языка Java,
- уметь проектировать и реализовывать программы на языке Java,
- знать основные форматы представления биологических данных (геномные, протеомные и другие данные),
- иметь представления о программных библиотеках (API), применяемых в области биоинформатики (BioJava, BioPerl),
- иметь представления о принципах построения программных конвейеров (pipe-line).

4. Структура и содержание дисциплины

Общая трудоемкость дисциплины составляет 54 академических часа.

№ п/п	Раздел дисциплины	Семестр	Неделя семестра	Виды учебной работы, включая самостоятельную работу студентов и трудоемкость (в часах)					Формы текущего контроля успеваемости *(по неделям семестра)* Форма промежуточной аттестации *(по семестрам)*
				Лекция	**Практ. занятие**	**Самост. работа**	**Контр. работа**	**Зачет**	
1.1	История программирования. Программирование: основные этапы, подходы	7	1	1					
1.2	Введение в Java. Компиляция и выполнение программ	7	1	1	2	2			
1.3	Синтаксис языка Java. Переменные, основные операторы	7	2	1	1	1			
1.4	Объектно-ориентированное программирование в Java. Классы и объекты. Наследование.	7	2	1	1	1			
1.5	Строки, массивы и работа с файлами в Java	7	3	2	2	2			
1.6	Исключения в Java	7	4	1	1	1			
1.7	Простейший синтаксический анализ в Java. Класс Tokenizer	7	4	1	1	1			

1.8	Контейнеры в Java. Организация программ. Пакеты	7	5	2	2	2			
1.9		7	5				2		**контрольная**
2.1	Форматы представления биологических данных в БД	7	6	2	2	2			
3.1	Методы обработки биологических текстовых последовательностей	7	7	2	2	2			
3.2	Разработка программ анализа биологической информации, хранящейся в БД	7	8	2	2	2			
			9				2		**контрольная**
		7	10					2	**Зачет**
				16	**16**	**16**	**4**	**2**	

Рабочий план

	Неделя	Темы занятий
Введение в язык Java	**СЕНТЯБРЬ** 1-я неделя	**Тема 1.** Предмет и история программирования. Обзор технологий и задач программирования. **Тема 2.** Переменные, типы данных и управляющие конструкции в языке Java. **Практика.** Установка JDK. Установка и обзор средства разработки Eclipse. Компиляция и выполнение программ в среде Eclipse.
	2-я неделя	**Тема 3.** Основы объектно-ориентированного программирования. Классы и объекты. Наследование. **Практика.** Разработка простейших классов. Наследование и иерархии классов.

	3-я неделя	**Тема 4.** Строки и массивы в языку Java. **Практика.** Операции со строками (поиск, выделение подстроки, конкатенация), регулярные выражения. Операции с массивами (поиск, сортировка). Массивы примитивных типов и массивы объектов.
	4-я неделя	**Тема 5.** Исключительные ситуации в Java. Работа с файлами в Java. Потоки ввода-вывода в Java. **Тема 6.** Синтаксический анализ в Java. Класс Tokenizer. Основные контейнеры Java. **Практика.** Генерация и перехват исключений в Java. Создание собственных исключений. Чтение из файла и запись в файл. Перенаправление потоков. Реализация простейших синтаксических анализаторов с помощью класса StringTokenizer. Использование контейнеров List<E>, Map<K,V>.
Форматы биол. данных	**ОКТЯБРЬ** 1-я неделя	**Тема 7.** Запуск внешних программ. Организация программных пакетов в Java. **Контрольная работа № 1:** Три задачи на работу со строками, массивами и контейнерами, написание простейшего синтаксического анализатора.
	2-я неделя	**Тема 8.** Типы биологических данных. Принципы организации биологических данных. Компьютерная обработка биологических данных. Популярные форматы хранения биологических данных. **Практика.** Разработка программ-парсеров для популярных форматов хранения биологических данных (Genbank, FASTA и др.).

	3-я неделя	**Тема 9.** Программные библиотеки обработки биологических данных. BioJava. **Практика.** *Использование классов библиотеки BioJava при написании приложения анализа биологических данных*.
Разработка биоинф. приложений	4-я неделя	**Тема 10.** Другие биологические данные, программные библиотеки для работы с ними. Принципы разработки комплексных приложения для анализа биологических данных. **Практика.** Разработка программ анализа биологических данных, решение биоинформатических задач.
	НОЯБРЬ 1-я неделя	**Контрольная работа № 2:** Три задания на разработку программ для анализа данных, записанных в известных форматах представления.
	2-я неделя	**Дифференцированный зачёт.**

Программа курса

I. Введение в язык Java

Тема 1. Предмет и история программирования. Обзор технологий и задач программирования.

История программирования. Основные этапы развития программирования. Обзор современных подходов: процедурное, модульное, структурное и объектно-ориентированное программирование.

Компилируемые и интерпретируемые языки. Интерпретация компилирующего типа. Байт-код, виртуальная машина. Java - язык и платформа. Средства разработки на языке Java. Программа Eclipse. Простейшая программа на языке Java.

Тема 2. Переменные, типы данных и управляющие конструкции в языке Java.

Переменные, типы данных и управляющие конструкции в языке Java. Основные числовые и булевы типы данных. Примитивные и сложные типы. Строки. Особенности целочисленной арифметики. Символьные последовательности и специальные символы. Арифметические и логические операторы. Простые и составные инструкции. Условный выбор. Циклы for, while, do-while и управление циклами: операторы break и continue. Оператор case.

Тема 3. Основы объектно-ориентированного программирования.

Основы объектно-ориентированного программирования. Классы и объекты. Программные модели: ориентированные на процесс, и ориентированные на данные. Абстрация и принципы объектно-

ориентированного программирования: инкапсуляция, наследование, полиморфизм. Состояние, поведение и уникальность объектов. Сообщения и методы. Сигнатура методов. Перегрузка методов. Конструкторы и создание объектов, оператор new. Автоматическая уборка мусора в Java. Статические методы. Области действия и время жизни переменных и объектов.

Тема 4. Строки и массивы в языку Java.

Массивы в языке Java. Одномерные, двумерные и многомерные массивы. Массивы примитивных типов и массивы объектов.

Строки в языке Java. Конкатенация, сравнение строк. Поиск подстроки и другие операции со строками в Java. Метод toString().

Тема 5. Ввод-вывод, исключения в Java

Исключительные ситуации в программировании. Механизмы обработки исключений в языке Java. Операторы try, catch, finally. Операторы throw и throws. Концепция потоков ввода-вывода в Java. Иерархии классов потоков ввода и вывода: байтовые и символьные потоки. Стандартные потоки. Работа с файлами в Java.

Тема 6. Синтаксический анализ в Java

Синтаксический анализ: основные понятия. Разбор аргументов командной строки. Преобразование строк в числа: методы Integer.parseInt, Float.parseFloat и т.п. Простейшие парсеры. Биоинформатика как синтаксический анализ генетических текстов. Обзор биоинформатических программ. Геномные браузеры. Разбиение строки на элементы. Класс StringTokenizer.

Абстрактные классы и интерфейсы. Понятие контейнера в программировании. Классы коллекций в языке Java. Списки и ассоциативные массивы.

Тема 7. Организация программных пакетов в Java.

Запуск внешних программ в Java. Понятие многопоточности в программировании. Многопоточность в языке Java. Способы использования сторонних программ в своём коде. Порождение процесса в Java, класс ProcessBuilder.

Организация собственных программных пакетов в Java. Импорт сторонних пакетов. Пример запуска программы UNAFold для расчёта вторичной структуры и энергии Гиббса молекулы РНК.

II. Форматы представления биологических данных

Тема 8. Типы биологических данных

Типы биологических данных. Принципы организации биологических данных. Компьютерная обработка биологических данных. Популярные форматы хранения биологических данных. Формат FASTA. Разметка геномов, форматы Genbank, EMBL и другие.

Тема 9. Программные библиотеки обработки биологических данных

Программные библиотеки обработки биологических данных. Пакет BioJava. Установка и работа с BioJava. Основные классы BioJava.

III. Принципы разработки комплексных приложения для анализа биологических данных

Тема 10. Принципы разработки комплексных приложения для анализа биологических данных.

Другие биологические данные, программные библиотеки для работы с ними. Формат UniProt/SwissProt. Формат PDB. Чтение различных форматов с помощью BioJava. Формат SBML для описания математических моделей биологических систем. Программный интерфейс (API) Systems Biology Workbench.

Конвейерная обработка данных с помощью пакета Taverna.

Принципы разработки комплексных приложения для анализа биологических данных.

5. Виды учебной работы и образовательные технологии, используемые при их реализации

Виды/формы образовательных технологий.

Преподавание курса ведется в виде чередования лекций и практических занятий. Основная форма образовательной активности на практических занятиях – написание программ на языке Java. Другие формы работы на практических занятиях – индивидуальные консультации студентов преподавателем, приём и оценка домашних заданий, совместные консультации студентов по решению трудных задач («мозговой штурм»).

Решаемые студентами задачи, с одной стороны, имеют своей целью закрепление текущего лекционного материала, а, с другой стороны, образуют «серии» задач. Каждая серия задач представляет собой одну большую, комплексную задачу, решение которой требует вовлечения нескольких тем, рассматриваемых на лекциях. Соответственно, в каждой частной задаче серии студенты отрабатывают определённый навык: алгоритм, шаблон программирования или технологию программирования. В конце серии все полученные знания и технологии суммируются в одной программе, что позволяет студентам, с одной стороны, лучше усвоить конкретные

принципы, идеи и технологии программирования, а, с другой стороны, получить системный взгляд как на программирование, так и на решаемые предметные (биологические) проблемы.

В начале каждого практического занятия студентам даётся список заданий, которые необходимо решить (написать программы) до следующего занятия: какая-то часть заданий решается в терминальном классе непосредственно на практическом занятии, оставшаяся часть решается самостоятельно в виде домашнего задания. В течение практического занятия студенты сдают преподавателю задачи, заданные на дом (если таковые имелись) и задачи, заданные на текущем занятии. Кроме того, возможна сдача домашних заданий по электронной почте или другим способом через сеть Интернет.

Помимо выдачи и приёма заданий, преподаватель консультирует студентов в ходе решения задач. При этом если преподавателем фиксируется непонимание одних и тех же вопросов (рассмотренных на лекциях) у двух и более студентов, преподаватель может разобрать эти вопросы более детально «у доски». Иногда для этих целей могут привлекаться сильные студенты, хорошо усвоившие данный материал. Если преподавателем фиксируется непонимание отдельных тем у 60% и более студентов, то преподаватель может внести изменения в план следующих лекций и практических занятий с целью более тщательной проработки трудных вопросов. Возможность такой обратной связи с аудиторией обусловлена тем, что практические и лекционные занятия ведутся одним преподавателем. Это обеспечивает большую гибкость в формировании планов лекционных и практических занятий.

Начиная со второй трети курса, часть практических занятий посвящается «защите» студентами своих программных проектов. При этом поощряется активное участие в дискуссии всех студентов. Такая форма обучения подготавливает студентов к защите дипломов, а также к будущей работе, как

в науке, так и в области коммерческой биоинформатики. Таким образом, на семинарских занятиях реализуется интерактивная форма обучения.

У студентов имеются дополнительные возможности для повышения квалификации в области программирования и автоматического получения зачёта:

1. По согласованию с преподавателем, студент может выбрать одну задачу из списка «сложных» задач и решить её в течение семестра. В конце семестра студент сдаёт программу преподавателю, показывая функциональность программы и отвечая на вопросы, касающиеся работы тех или иных частей программы, использованных технологиях и т.д. При положительной сдаче программы, студент демонстрирует работу остальным студентам в форме защиты проекта.
2. По согласованию с преподавателем и научным руководителем, студент самостоятельно, или совместно с научным руководителем, формулирует задачу разработки программной системы в рамках своей дипломной работы (специфика кафедры информационной биологии ФЕН состоит в том, что большинство дипломных работы студентов кафедры так или иначе связаны с компьютерной обработкой биологической информации). В конце семестра студент сдаёт программу преподавателю и своему научному руководителю.

В случае возникновения у студента трудностей с усвоением лекционного материала или решением задач предусмотрены также индивидуальные занятия во внеучебное время.

Отметим, что преподаватель курса «информационные технологии и языки программирования» является действующим специалистом в области биоинформатики и математического и компьютерного моделирования биологических процессов. Значительная часть задач, предлагаемых в курсе, основана на реальных научных задачах, в разное время решавшихся в ИЦиГ СО РАН и на кафедре информационной биологии. К тому же преподаватель

тесно взаимодействует с преподавателями других курсов кафедры информационной биологии и, что позволяет формировать список решаемых задач с учётом потребностей других курсов. Это способствует формированию у студентов системно-биологического мышления.

6. Оценочные средства для текущего контроля успеваемости, промежуточной аттестации по итогам освоения дисциплины.

Формой текущего контроля при прохождении дисциплины «Информационные технологии и языки программирования» является контроль посещаемости занятий, сдача домашних заданий и написание контрольных работ.

Для того, чтобы быть допущенным зачёту, студент должен выполнить следующее:

- в ходе обучения посетить не менее 70 % занятий;
- сдать все домашние задания в виде работающих компьютерных программ;
- написать на положительные оценки две контрольные работы.

	Тема
Контрольная работа № 1	Три задачи на работу со строками, массивами и контейнерами, написание простейшего синтаксического анализатора.
Контрольная работа № 2	Три задания на разработку программ для анализа данных, записанных в известных форматах представления.

В случае отсутствия на контрольной работе по уважительной причине (наличие медицинской справки) контрольную работу можно переписать в течение недели от окончания срока действия справки. Контрольные работы

оцениваются по принципу:

- Одна решёная задача – «удовлетворительно»;
- Две решёных задачи – «хорошо»;
- Три решёных задача – «отлично»;

В зависимости от результатов работы в течение семестра, студент имеет право на получение оценки без экзамена (оценки-«автомата»). Для этого он должен:

- в ходе прохождения дисциплины посетить не менее 70 % занятий;
- написать две контрольных работы на оценку не ниже «хорошо»;
- написать и сдать все компьютерные программы, заданные в течение семестра в срок до последнего занятия;

либо:

- по согласованию с лектором выбрать и самостоятельно решить задачу из категории «сложных» (написать работающую компьютерную программу);
- выступить на одном из практических занятий с кратким докладом о проделанной работе: «защитить» свою программу.

Оценка-«автомат» выводится как средневзвешенная из полученных студентом баллов по результатам работы в семестре. Дополнительные баллы студент может получить, выступая на практических занятиях с защитой разработанных им программ, принимая активное участие в дискуссиях на защитах программ других студентов, выступая на практических занятиях с объяснением той или иной лекционной темы другим студентам.

Итоговый контроль. Итоговую оценку за семестр студент может получить на дифференцированном зачёте в конце семестра, где студент имеет возможность либо повысить оценку, полученную им «автоматом», либо получить любую положительную (или неудовлетворительную) оценку в случае отсутствия у него «оценки-автомата» по результатам работы в семестре.

7. Учебно-методическое обеспечение самостоятельной работы студентов.

Учебно-методическое обеспечение дисциплины: при подготовке к лекциям и семинарам студенты могут использовать рекомендованные преподавателем литературные источники и Интернет-ресурсы, а также любую доступную справочную литературу, программное обеспечение и базы данных.

Рекомендованные источники:

1. *Вирт Н.* Алгоритмы и структуры данных. // СПб.: Невский диалект, 2005.

2. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений. 3-е изд. // М.: Вильямс, 2008 г.

3. *Эккель Б.* Философия Java. Библиотека программиста. 4-е изд. // СПб.: Питер, 2009.

4. *Дурбин Р., Эдди Ш., Крог А., Митчисон Г.* Анализ биологических последовательностей // РХД, 2006 г., 480 стр.

5. *Леск А.* Введение в биоинформатику // М.: Бином. Лаб. знаний, 2009., 318 стр.

Предмет и история программирования. Обзор технологий и задач программирования.

Программирование – в узком смысле это означает процесс написания программного кода, или *разработки* компьютерных программ с использованием какого-либо *языка программирования*. Однако в настоящее время процесс разработки программы подразумевает большее число стадий, и написание программного кода является только одной, хотя и важной стадией. Всего же выделяют шесть стадий разработки программного обеспечения: анализ, проектирование, реализация (написание программного

кода), отладка и тестирование, внедрение и, наконец, сопровождение (рис. 1.1.1) [Буч, 1998].

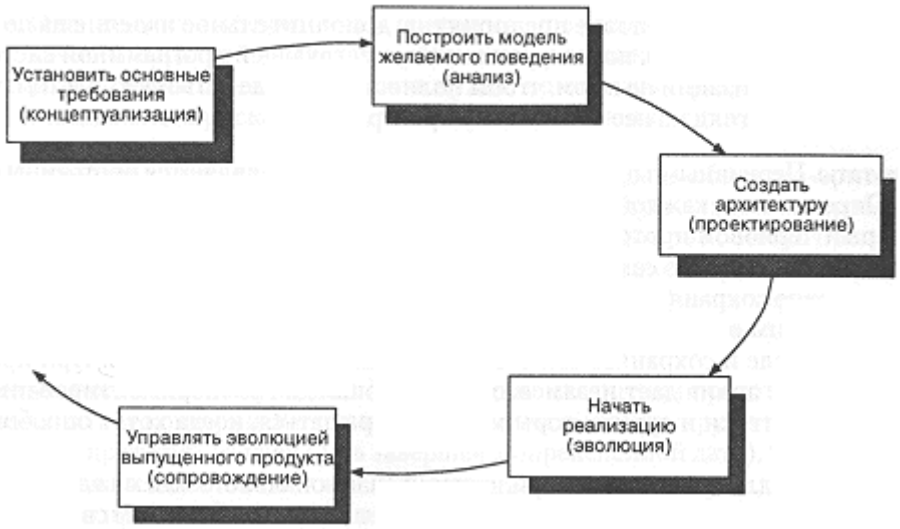

Рис. 1. Стадии разработки программного обеспечения (по [Буч, 1998]).

Стадия анализа (analysis) включает в себя сбор требований к программному обеспечению, их систематизацию и документирование. Для выявления противоречий и неполноты в этих требованиях анализирующий должен иметь как минимум базовые представления о предметной области и существующих методах и подходах к решению сходных задач.

Стадия проектирования (design) или создания проекта программного обеспечения подразумевает спецификацию свойств программы на основе требований, сформулированных на стадии анализа. В ходе проектирования обычно формируется архитектура программного обеспечения (software archictecture): структур программного комплекса, включающая программные модули и компоненты, а также отношения между ними.

Реализация программы (implementation) включает написание исходного текста программы (в том числе, отдельных программных модулей и компонент) и его *компиляцию* – преобразование в исполнимый код с помощью специальной программы-*компилятора*.

Тестирование (testing) и отладка (debugging) программы подразумевает проверку соответствия программы заданным спецификациям, а также

обнаружение, локализацию и устранение ошибок в программных модулях. Для поиска ошибок применяются специальные программы – отладчики (дебаггеры, от англ. debugger).

Внедрение или сдача в эксплуатацию программы включает в себя обеспечение пользователей программой, настройку программы под конкретные условия использования, а также обучение пользователей работе с программой.

И, наконец, стадия сопровождения подразумевает улучшение, оптимизацию и устранение ошибок программного обеспечения после внедрения, что зачастую достигается посредством обратной связи с пользователями.

Приведённый выше порядок стадий разработки программного обеспечения не являются строго зафиксированным, так, например сопровождение зачастую требует реализации новых функций, затем повторное тестирование и отладку и т.д.

Объектно-ориентированное программирование и язык Java

Java – один из наиболее популярных объектно-ориентированных языков программирования настоящего времени. Программы, написанные на этом языке, работают на всех современных компьютерных платформах благодаря тому, что исходные тексты компилируются в специальный байт-код, который может работать на любой виртуальной Java-машине (Java Virtual Machine – JVM). Более того, подобные виртуальные машины также существуют для множества других электронных устройств (мобильные телефоны, навигаторы, и др.).

Объектно-ориентированная парадигма программирования основана на понятиях классов и объектов. Алан Кэй выделяет пять основных характеристик, присущих объектно-ориентированному программированию (ООП):

1. Всё есть объект. Представляйте объекты как усовершенствованные переменные; они хранят данные, но вы можете “сделать запрос” к такому объекту, попросив его самого выполнить операцию. Теоретически любой умозрительный компонент в вашей задаче (собак, дома, услугу и т.п.) может быть представлен в виде объекта в вашей программе.

2. Программа - это группа объектов, указывающих друг другу, что делать, путём посылки сообщений. Чтобы сделать запрос к объекту, вы “посылаете ему сообщение”. Более общо сообщение можно представить как вызов функции, принадлежащей определённому объекту.

3. Каждый объект имеет свою собственную «память», состоящую из других объектов. Другими словами, вы создаете объект нового вида, встраивая в него уже существующие объекты. Благодаря этому, вы можете построить сколь угодно сложные объекты, скрывая общую сложность системы за простотой отдельных объектов.

4. Каждый объект имеет тип. Другими словами, каждый объект является экземпляром класса, где “класс” - это синоним слова “тип”. Важнейшее отличие классов друг от друга заключается в ответе на вопрос: “Какие сообщения вы можете посылать объекту этого класса”.

5. Все объекты определенного типа могут принимать одинаковые сообщения. Это действительно важное утверждение, как мы увидим позднее. Так как объект типа «круг» также является объектом типа «фигура», «круг» гарантированно примет сообщения для «фигуры». Это означает, что вы можете писать программный код для фигур и быть уверенным в том, что он подойдёт для всего, что подходит под понятие фигуры. Данное свойство взаимозаменяемости является одним из наиболее полезных свойств ООП.

Брюс Эккель [2004] формулирует основные свойства объекта в более сжатой форме: «объект может иметь в своём распоряжении внутренние данные (которые и есть состояние объекта), методы (которые определяют поведение объекта), и каждый объект можно уникальным образом отличить

от любого другого объекта – говоря более конкретно, каждый объект обладает уникальным адресом в памяти». В случае распределённых программ разные объекты могут существовать на разных компьютерах и в разных адресных пространствах, и тогда для их идентификации используются не адреса памяти, а дополнительных характеристики.

В силу того, что объекты, идентичные во всём кроме внутреннего состояния в ходе работы программы, могут быть сгруппированы в «классы объектов», в языке Java (а также в ряде других объектно-ориентированных языков программирования) выделяется ключевое слово class, декларирующее описание класса, или по-другому абстрактного типа данных. Таким образом, создание новых типов данных и манипулирование переменными этих типов (т.е. объектами или экземплярами класса) является основой ООП. Основным способом взаимодействия с объектами является посылка объекту сообщения или запроса, что осуществляется с помощью вызова метода (функции конкретного объекта, которому посылается сообщение).

Список методов класса и его декларируемых свойств определяют его интерфейс, который специфицирует, какие запросы вы вправе делать объектам этого класса. При этом пользователь объекта, вообще говоря, может не знать, каким образом выполняется тот или иной метод – с точки зрения пользователя класс представляет собой чёрный ящик. Данный принцип сокрытия внутренней реализации классов носит название инкапсуляции и также является одним из основных принципов ООП. Принцип инкапсуляции призван минимизировать число связей между классами и, соответственно, упростить независимую реализацию и модификацию классов.

Следующим важным принципом ООП является наследование – возможность определять один класс на основе другого с сохранением всех свойств и методов класса-предка (base class, parent class – базовый или родительский класс, а в языке Java также принят термин super class – суперкласс) и, при необходимости, добавлять дополнительные свойства и

методы. Набор классов, связанных отношением наследования, называют иерархией классов. Используя наследование от родительского класса, можно создавать новые классы, которые не только будут содержать в себе все данные-члены родителя, но, что особенно важно, будут обладать интерфейсом класса-родителя. Благодаря этому объекты производного класса (derived class, класс-наследник) также являются объектами родительского класса, а производный класс – частным случаем родительского. Несмотря на то, что производные классы обладают интерфейсом класса-родителя, конкретные реализации отдельных методов в производных классах могут отличаться от их реализации в родительском классе (а могут и не отличаться). При обращении к объектам производного класса как к объектам базового класса (что позволяет сделать принцип наследования), вызываемые методы всё равно будут принадлежать объектам производного класса. Этот принцип называется полиморфизмом, и является третьим наиболее важным принципом ООП.

Переменные, массивы, классы и объекты в Java

Под словом «переменная» в науке и технике подразумевается некий атрибут физической или абстрактной системы (модели), который может изменять своё значение (например: температура воздуха, давление, число молекул АТФ в клетке и т.д.). В программировании под переменной подразумевается именованная область памяти, имя или адрес которой можно использовать для осуществления доступа к данным, находящимся в переменной. Таким образом, переменные – это «места для хранения данных».

В языке Java выделяют особую группу типов данных, которые наиболее часто применяются в программировании – так называемые *примитивные типы*. Размеры всех примитивных типов жёстко фиксированы и не меняются при переходе с одной машинной архитектуры на другую, в отличие от многих других языков (например, C или C++). Это выгодно отличает Java и

является одной из причин очень хорошей переносимости Java-программ на разные платформы.

Заметим, что в отличие от переменных примитивных типов данных, инициализация более сложных, составных объектов требует явного создания с помощью ключевого `new`. Более сложные типы данных, определяемые в классах, называются ссылочными типами и по умолчанию инициализируются специальным значением `null`, аналогом нулевого указателя в языках C и C++. При попытке обращения к объекту со значением `null`, возникает ошибка времени исполнения (run-time error), так называемое *исключение нулевого указателя* (Null Pointer Exception).

Отметим, что объекту нельзя послать сообщения, если для него не определены методы (как в случае класса, определённого выше в тексте). Вообще говоря, при определении класса в него включается две разновидности элементов: поля данных (fields), то есть переменные класса, а также методы (methods), то есть функции класса. Поле представляет собой либо объект любого (заранее определённого) типа, либо переменную примитивного типа. Каждый объект имеет свой собственный адрес в памяти, при этом совместное использование обычных полей разными объектами невозможно. Рассмотрим пример класса, определяющего комплексные числа – такой класс должен содержать в себе поля, соответствующие вещественной и мнимой части комплексного числа:

```
class ComplexNumber {
        double real;  //  вещественная  часть  комплексного
числа
        double image; // мнимая часть комплексного числа
}
```

Данный класс пока только хранит данные и не имеет методов. Тем не менее, можно создать объект этого класса:

```
ComplexNumber complex = new ComplexNumber();
```

Полям данного класса можно присваивать значения, при этом мы должны обращаться к полям как к членам объекта. Для этого используется имя объекта, оператор точка (`.`), и, наконец, имя поля данных объекта. Например:

```
complex.real = -10.5;
complex.image = 7.2;
```

Управляющие конструкции в языке Java

Все управляющие конструкции так или иначе связаны с условиями, для которых вычисляется истинность или ложность, и, исходя из результата выбирается способ выполнения (ветвь программы). Условным выражением может быть как результат вычисления оператора (например, результат проверки на неравенство `A!=B`), так и результат выполнения метода, тип возвращаемого значения которого `boolean`.

Конструкция `if-else` («если-иначе») является наиболее распространённым способом передачи управления в программе. Ключевое слово `else`, вообще говоря, не является обязательным. Таким образом, возможно две формы конструкции `if`:

```
if( логическое выражение){
 // Команды...
}
и
if( логическое выражение){
 // Команды...
}
else{
 // Другие команды...
}
```

Логическое выражение, определяющее условие должно выдавать результат типа `boolean`. Рассмотрим применение условных конструкций на примере программы решения квадратного уравнения:

```
    public class QuadraticEquationSolve {
     public static void main(String[] args) {
          // Коэффициенты уравнения ax^2 + bx + c = 0
          double a = 1.0;
          double b = 3.0;
          double c = 2.0;
                    if(a==0){
               if((b==0) && (c==0)){
                    System.out.println("Бесконечное множество
корней");
               }
               else if(b==0){ // Т.е. ур-е вида 0*x=c, c!=0
                    System.out.println("Корней нет");
               }
               else{
                    double x = -c/b;
                    System.out.println("Один корень: x="+x);
               }
          }
          else{ // т.е. a!=0
               double discr = b*b - 4*a*c;
               if(discr>0){
                    double x1 = (-b + Math.sqrt(discr))/(2*a);
                    double x2 = (-b - Math.sqrt(discr))/(2*a);
                    System.out.println("Два корня: x1="+x1+"
x2="+x2);
```

```
                }
                else if(discr==0){
                    double x = -b/(2*a);
                    System.out.println("Один корень: x="+x);
                }
                else{
                    System.out.println("Действительных корней
нет");
                }
            }
        }
    }
```

Данная программа проверяет уравнение на вырожденность и отрицательность дискриминанта, для каждого варианта реализуется свой способ нахождения корней, или выдаётся информация о том, что корней нет. Как видно из данного примера, условные конструкции могут быть вложенными.

Следующим важнейшим видом управляющих конструкций являются *циклы*, предназначенные для многократного повтора набора инструкций. Последовательность инструкций, предназначенная для многократного исполнения, называется *телом цикла*. Единичное исполнение тела цикла называется *итерацией* (реже, шагом цикла). В языке Java существует три циклических конструкции: `while`, `do-while` и `for`.

Цикл `while` строится следующим образом:

```
while(логическое-выражение){
 // Инструкции…
}
```

`Логическое-выражение` вычисляется перед началом цикла, а затем каждый раз перед началом следующей итерации. Следующая программа вычисляет квадратный корень из двух с точностью до 10^{-6}:

```
public class SquareRoot {
 public static void main(String[] args) {
      double eps = 1e-6; // Точность вычисления: 10^-6
      double a = 2.0;    // Из чего извлекается корень
      double xprev = a;
      double xnext = 0.5*(a + 1); // Первый член
// последовательности
      double delta = xnext-xprev; // Расстояние между
двумя
// соседними членами последовательности
      while(Math.abs(delta)>eps){// Пока разница больше
delta
           xprev = xnext;
           xnext = 0.5*(xprev+a/xprev);
           delta = xnext-xprev;
      }
      System.out.println("Корень из "+a+" = "+xnext);
 }
}
```

Цикл `do-while` строится следующим образом:

```
do {
 // Инструкции…
} while(логическое-выражение)
```

Единственное отличие от цикла `while` заключается в том, что цикл `do-while` выполняется как минимум один раз, даже если `логическое-выражение` изначально ложно.

Наиболее часто используемой конструкцией цикла является конструкция `for`. Цикл `for` проводит инициализацию перед первой итерацией, затем

выполняется проверка условия цикла, и в конце каждой итерации осуществляется некое приращение: обычно это изменение управляющей(-щих) переменной(-ных). Цикл `for` строится следующим образом:

```
for(инициализация; логическое-выражение; приращение) {
    // Инструкции…
}
```

Любое из трёх выражений цикла (`инициализация`, `логическое-выражение`, `приращение`) можно опустить. Перед каждой итерацией цикла проверяется условие – значение `логического-выражения`, если оно становится ложно, то выполнение переходит к инструкции, следующей за конструкцией `for` – происходит «выход из цикла». Цикл for обычно используется для инициализации массивов и других контейнеров, а также для выполнения счётных задач:

```
int[] array = new int[10];
for(int i = 0; i<array.length; i++){
    array[i] = (i+1)*(i+1); // Инициализация массива квадратами
//первых 10 положительных целых чисел
}
```

Обратите внимание на использование `array.length` в условии цикла: при инициализации массивов, а также при работе с ними с помощью циклов рекомендуется использовать именно `length` – это предотвратит выход за пределы массива при работе цикла.

Следующая группа ключевых слов обеспечивает безусловный переход, то есть передачу управления без проверки каких либо условий. Эти ключевые слова: `return`, `break` и `continue`. Ключевое слово `return` используется для двух целей: во-первых, оно указывает, какое значение возвращается методом (если тип возвращаемого значение не `void`), а во-вторых, оно используется для незамедлительного выхода из метода. Если

метод возвращает значение, отличное от `void`, то необходимо проследить, чтобы каждая логическая ветвь метода возвращала конкретное значение.

Ключевое слово `break` принудительно завершает выполнение цикла, при этом оставшиеся операторы даже текущей итерации не выполняются. Команда `continue` заканчивает выполнение текущей итерации и переходит сразу к началу нового шага.

Наконец, ключевое слово `switch` используется для выбора из нескольких вариантов, в зависимости от значения целочисленного выражения. Логика выполнения команды выглядит следующим образом: Вычисляется значение выражения `expr`, если значение `expr` совпадает с одним из значений в метках `case`, то происходит переход к метке, в противном случае переход к метке `default`. Инструкции выполняются до конца ветвления, либо до команды `break`:

```
switch (expr)
{
case знач1:
 набор_инструкций1 //break;
case знач2:
 набор_инструкций2 //break;
…
default:
 набор_инструкцийD
};
```

Таким образом, используя многообразие управляющих конструкций Java, можно создавать сложные конструкции, руководствуясь логикой решаемой задачи.

Строки в языке Java

Работа со строками является одним из основных инструментов при разработке приложений в области биоинформатики. Выгодной стороной языка Java является наличие уже в базовой версии языка достаточно мощных средств работы со строками (в сравнении, например, с C/C++). Встроенный класс String в Java обеспечивает все основные операции со строками: конкатенацию, поиск подстроки т.д.

Конструирование строк в Java возможно несколькими способами:

- В обычном стиле (указывает на особый статус класса `String` – возможность создавать объекты без использования оператора `new`):

```
String str = "Start";
```

- В объектно-ориентиованном стиле (с использованием оператора `new`):

```
String str = new String("Start");
```

- В объектно-ориентированном стиле из массива символов `char[]`:

```
char chars[] = {'a','u','g'};
String str2 = new String(chars);
```

- С помощью различных выражений:

```
String str3 = str+str2;
// str3 = "stop aug"
String str4 = str+"codon "+str2;
// str4 = "stop codon aug"
```

Длину строки можно узнать с помощью метода `length()`. Что интересно, мы можем применить этот метод к необъявленной строке (строковой константе в кавычках):

```
System.out.println(str4.length());
```

```
System.out.println("Stop codon".length());
```

Возможно извлечение одного символа из строки:

```
char ch;
ch = str.charAt(2);
```

извлечение нескольких символов из строки в массив:

```
int start = 5;
int end = 10;
char buf[] = new char[end-start];
str4.getChars(start, end, buf, 0);
```

или преобразование всей строки в массив символов:

```
char[] chars = str4.toCharArray();
```

Возможно регистрозависимое и регистронезависимое сравнение строк:

```
String str1 = "STOP";
String str2 = "stop";
System.out.println(str1.equals(str2)); // false
System.out.println(str2.equalsIgnoreCase(str1)); // true
```

Можно также определить, является ли подстрока началом (префиксом) или концом (суффиксом) другой строки:

```
String str1 = "STOP";
String str2 = "stop";
System.out.println(str1.startsWith("STO")); // true
System.out.println(str2.endsWith("top")); // true
```

Лексикографическое сравнение (аналогичная сортировка слов в словаре) можно провести для двух строк с учётом или без учёта регистра:

```
String str1 = "start";
String str2 = "STOP";
System.out.println(str1.compareTo(str2)); // 32
```

```
System.out.println(str1.compareToIgnoreCase(str2)); // -14
```

Можно определить индекс (позицию) подстроки в строке – с начала строки, начиная с некоторой фиксированной позиции или с конца строки:

```
String str1 = "stararararart";
int fromIndex = 5;
System.out.println(str1.indexOf("ar")); // 2
System.out.println(str1.indexOf("ar", fromIndex)); // 6
System.out.println(str1.lastIndexOf("ar")); // 8
```

Можно, наоборот, выделить подстроку по индексам:

```
int start = 5;
int end = 8;
String str1 = "stararararart";
System.out.println(str1.substring(start, end)); // "rar"
```

Следующий пример показывает, как заменить в строке одну подстроку на другую:

```
public class SubstituteString {

    public static void main(String[] args) {
        String org = "Это стоп-кодон, стоп";
        String search = "стоп";
        String subst = "старт";
        String result = "";
        int i;
        do { // заменить все совпавшие подстроки
            System.out.println(org);
            i = org.indexOf(search); // индекс первого
вхождения
```

```
                if (i != -1) { // поиск удался
                    result = org.substring(0, i);
                    result = result + subst;
                    result = result + org.substring(i +
search.length());
                    org = result;
                }
            } while (i != -1);
        }
}
```

Возможна замена всех вхождений одного символа в строке другим символом:

```
String str1 = "Здравствуйте, товарищи!";
String str2 = str1.replace('в', 'ф');
System.out.println(str2); // Здрафстфуйте, тофарищи!
```

Кроме того, можно заменить одну подстроку другой:

```
String str1 = "Здравствуйте, товарищи!";
String str2 = str1.replace("вствуйте", "ссте");
System.out.println(str2); // Здрассте, товарищи!
```

Метод `contains` проверяет наличие подстроки в строке:

```
String str1 = "ttgaacatg";
if(str1.contains("atg")){
    System.out.println("Содержит старт-кодон");
}
else{
    System.out.println("Не содержит старт-кодон");
}
```

Замена всех вхождений регулярного выражения строкой производится следующим образом:

```
String str1 = "What;do;you;mean?";
String str2 = str1.replaceAll(";", " ");
System.out.println(str2); // “What do you mean?”
```

Возможна также замена только первого вхождения:

```
String str1 = "What;do;you;mean?";
String str2 = str1.replaceFirst(";", " ");
System.out.println(str2); // What do;you;mean?
```

Разделение строки на подстроки (токены) по регулярному выражению, напоминающее разделение с помощью класса `StringTokenizer`, показано ниже:

```
String str1 =
"Instruction1;;;Instruction2;;;Instruction3;;;Instruction4";
String[] arr = str1.split(";;;");
System.out.println(arr.length);
for(int i = 0; i< arr.length; i++)
    System.out.println(arr[i]);
```

Перегруженная версия метода `String.split(String s, int num)` разделяет строку на массив подстрок по ограничителю (регулярное выражение), длина массива не больше числа, определяемого вторым аргументом:

```
String str1 =
"Instruction1;;;Instruction2;;;Instruction3;;;Instruction4";
String[] arr = str1.split(";;;",2);
System.out.println(arr.length);
for(int i = 0; i< arr.length; i++)
```

```
System.out.println(arr[i]);
```

Проверить, удовлетворяет ли строка некоему регулярному выражению, можно с помощью метода `String.matches`:

```
String str1 = "gcgcgcgcgcgcgcgc";
String str2 = "gcgcgcgcgcgcgcga";
if(str1.matches("(gc)*")){
    System.out.println("str1: GC-повтор");
}else{
    System.out.println("str1: не GC-повтор");
}
if(str2.matches("(gc)*")){
    System.out.println("str2: GC-повтор");
}else{
    System.out.println("str2: не GC-повтор");
}
```

Перевести все символы строки в верхний (или нижний) регистр можно с помощью методов `String.toUpperCase` и `String.toLowerCase`.

```
String str1 = "ЭтА СТроКа нАПИсАнА КАк ПопалО";
String strU = str1.toUpperCase();
String strL = str1.toLowerCase();
System.out.println(strU);
System.out.println(strL);
```

Массивы в языке Java

Для хранения наборов однотипных данных в Java, также как и в большинстве других языков программирования, используются *массивы*.

Массивы состоят из *элементов*, доступ к которым организуется по имени массива и *индексу* (порядковому номеру) элемента в массиве, заключённому в квадратные скобки (`[ ]`). Объявление массива, его инициализация и доступ к его элементам показаны ниже:

```
int array1[]; // Объявление массива целых чисел
int[] array2; // Такая запись тоже допустима
array1 = new int[10];// Инициализация массива из 10 элементов
array1[0] = 1; // Индексация: доступ к первому эл-ту массива
array2 = {10,8,7}; // Статическая инициализация ячеек массива
```

Массивы могут объявляться не только для примитивных типов данных, но и для объектов определённого класса. Декларация и инициализация таких массивов абсолютно такая же, как и для массивов примитивных типов:

```
ComplexNumber[] complexes; // массив комплексных чисел
complexes = new ComplexNumber[10]; // создаём из 10 элементов
```

Особое внимание стоит обратить на то, что при создании массива объектов его элементы инициализируются нулевыми (`null`) ссылками (в отличие от массивов примитивных типов, которые инициализируются значениями по умолчанию, например, для числовых типов, нулями). Поэтому каждый отдельный объект-элемент массива необходимо инициализировать с помощью оператора `new`. Поскольку массивы в языке Java являются ссылочными типами данных, то они обладают свойствами объектов. В частности, мы можем узнать размер (количество элементов) массива, с помощью поля `length`, нумерация элементов при этом идёт от 0 до length-1:

```
int array[] = {13, 10, 24, -17, 6, 85}; // инициализация
```

```
    int len = array.length; // len теперь равно 6
    int a = array[0]; // a = 13
    int b = array[6]; // ошибка времени исполнения
(исключение)
```

Для объявления двумерных массивов, а также массивов более высокой размерности, используется соответствующее количество пар квадратных скобок: `type[][]`, `type[][][]` и т.д. Такие многомерные массивы рассматриваются как массивы, элементами которых являются массивы меньшей на единицу размерности. Соответственно, многомерные массивы могут быть несимметричными, рассмотрим это на примере двумерного массива:

```
    int array[][] = new int[3][3]; // массив 3x3
    array[0] = new int[10]; // первая строка состоит из
10 эл-тов
    array[1] = new int[2]; // вторая строка состоит из 2
эл-тов
    array[2] = null; // третья строка не имеет эл-тов
```

Ввод-вывод в Java

Задачи ввода-вывода данных встречаются в программистской практике, пожалуй, наиболее часто. Для организации ввода и вывода информации в языке Java используется программная абстракция *потока*. Существуют *потоки ввода (input stream)* информации, и *потоки вывода (output stream)* информации. В качестве потоков ввода могут выступать файлы, устройства (клавиатура, мышь), интернет-соединения (сокеты), потоки вывода других программ. Аналогично, в качестве потоков вывода могут выступать и файлы, и устройства (монитор, звуковая карта, принтер), и интернет-соединения, и потоки ввода других программ.

Замечательным свойством данной абстракции является то, что как только нужный поток (например, входной) настроен, вашей программе

становится безразлично, из какого именно потока она получает данные – с клавиатуры, из файла или из другого источника. Способ обработки этих данных будет неизменным, поскольку этот способ специфицирован в иерархии классов потоков языка Java. Базовыми классами для обслуживания ввода и вывода информации в языке Java являются классы *потоков* (stream), которые подразделяются на *байтовые* (`InputStream/OutputStream`) и *символьные* (`Reader/Writer`). Исходно в языке Java присутствовали только байтовые потоки, но затем были добавлены символьные, ориентированные на работу с символами и строками (подробная информация о потоках может быть найдена, например, в книге Б.Эккеля [2009]).

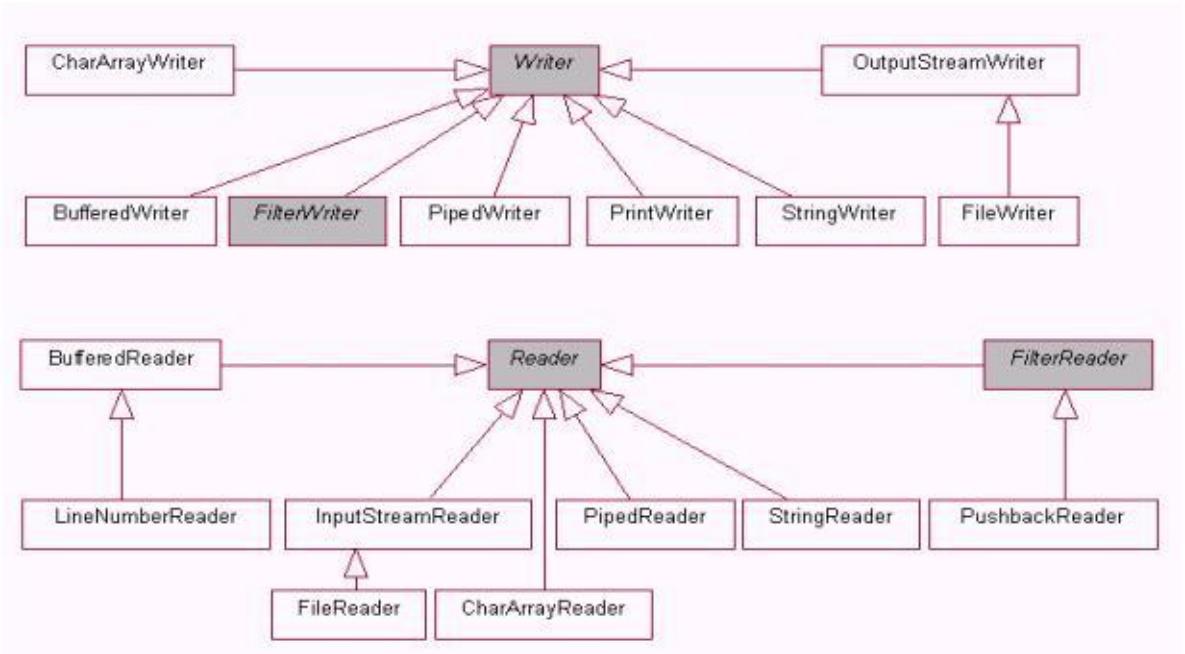

Рис. 2. Иерархия классов символьных потоков вывода (Writer) и ввода (Reader).

В первом примере показана программа, которая считывает символы из стандартного потока ввода `System.in`, который обычно ассоциирован с клавиатурой, и затем выводит эти символы в стандартный поток вывода `System.out`, который обычно ассоциирован с экраном:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class IOExample01 {
     public static void main(String[] args) {
          BufferedReader br = new BufferedReader(new
             InputStreamReader(System.in));
          char c='a';
          do {
              try {
                    c = (char) br.read();
               } catch (IOException e) {
                    // "Отлов" ошибки ввода-вывода
                    e.printStackTrace();
               }
              System.out.println(c);
          }while(c!='q');
     }
}
```

В данном примере продемонстрирован важный аспект работы с потоками – возможность *перенаправления* одного потока в другой:

```
BufferedReader br = new BufferedReader(new
             InputStreamReader(System.in));
```

В данном фрагменте создаётся объект буферизованного символьного потока ввода `BufferedReader`, который при инициализации связывается с объектом символьного потока ввода `InputStreamReader`, который, в свою очередь, при инициализации был связан со стандартным потоком ввода `System.in`. Посимвольное чтение из потока осуществляется с помощью метода `read`, определённого для всех классов входных потоков. Чтение

продолжается до тех пор, пока из потока не будет прочитан символ ‘q’ (т.е. пока не будет нажата соответствующая клавиша).

В следующем примере показана программа читающая из входного потока не *посимвольно* (как в предыдущем случае), а *построчно*:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class IOExample02 {
     public static void main(String[] args) {
          BufferedReader br = new BufferedReader(new
              InputStreamReader(System.in));
              String str="";
              do {
                  try {
                      str = br.readLine();
                  } catch (IOException e) {
                      e.printStackTrace();
                  }
                System.out.println(str);
             }while(!str.equals("stop"));
          }
}
```

Построчное чтение осуществляется с помощью метода `readLine`. Оно продолжается до тех пор, пока не будет введена строка “stop”. Благодаря возможности перенаправления потоков, программа, осуществляющая чтение строк из файла, будет выглядеть очень похоже, за исключением несколько отличающейся инициализации `BufferedReader`:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
```

```
public class IOExample03 {
     public static void main(String[] args) {
          BufferedReader reader;
          try {
          reader = new BufferedReader(new
FileReader("file.txt"));
          String line = "";
          while ((line = reader.readLine()) != null) {
              System.out.println("Line: "+line+
              " : has "+line.length()+" length");
          }
          }
          catch (IOException e) {
              e.printStackTrace();
          }
     }
}
```

Данная программа считывает строки из файла и выводит в стандартный поток вывода информацию о длине каждой строки. На этот раз буферизованный поток `reader` инициализируется с помощью объекта класса `FileReader`, который, в свою очередь, инициализируется строкой с именем файла (“`file.txt`”). Считывание происходит до тех пор, пока не будет достигнут конец файла. В этом случае `reader.readLine()` вернёт нулевой объект `null` и цикл завершится.

Запись в файл осуществляется способом, аналогичным обычному выводу в стандартный поток System.out (на экран):

```
import java.io.IOException;
import java.io.PrintStream;
public class IOExample04 {
```

```
    public static void main(String[] args) {
        PrintStream out;
        try {
            out = new PrintStream("out.txt");
            for (int i = 1; i <= 100; i++)
                out.println(i * i);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Данная программа считает квадраты всех чисел от одного до ста и записывает их в файл `out.txt`. Для более сложных операций с файлами в Java имеется дополнительные классы, основным из которых является класс `File`. Данный класс позволяет работать как с файлами, так и с каталогами. В следующем примере показано, как получить список всех файлов в текущей директории. Стоит обратить внимание на то, что когда создаётся объект класса `File`, это не ведёт автоматически к созданию соответствующего файла. Созданный объект может быть ассоциирован как с уже существующим файлом, так и с вновь создаваемым.

```
import java.io.File;
public class IOExample05 {
    public static void main(String args[]) {
        File directory;
        directory = new File(".");
        String list[] = directory.list();
        for (int i = 0; i < list.length; i++)
            System.out.println(list[i]);    }
}
```

Исключения в Java

Для обработки аварийных состояний, возникающих в программе во время выполнения (например, при делении на нуль, выходе за пределы массива и т.д.) в языке Java используется механизм так называемых *исключений*. Исключение в языке Java – это объект, описывающий ошибочную (исключительную) ситуацию, произошедшую в некоторой части кода. Логика работы виртуальной машины при этом следующая. Когда возникает исключение, создаётся объект и «вбрасывается» в метод, вызвавший ошибку. Метод может обработать исключение, либо метод может передать исключение в вызвавший его метод. В некоторой точке исключение должно быть «захвачено» и обработано.

Оператор `throws` в сигнатуре метода указывает на то, что метод может сгенерировать исключение. Если внутри метода генерируется исключение, этот оператор обязан присутствовать в сигнатуре. Оператор `throw` непосредственно генерирует исключение (внутри метода):

```
public ComplexNumber divide(ComplexNumber b) throws Exception{
        if((b.real==0) && (b.image == 0))
                throw new Exception();

        double denom = b.real*b.real+b.image*b.image;
        double re = (this.real*b.real+this.image*b.image)/denom;
        double im = (this.image*b.real-this.real*b.image)/denom;
        return new ComplexNumber(re,im);
}
```

Рис. 3. Генерация исключения в Java.

Оператор `try` определяет блок кода, который может вызвать ошибку исполнения и исключение. Оператор `catch` определяет тип перехватываемого исключения и описывает обработчик исключения. Оператор `finally` определяет блок кода, выполняющийся после блока

try/catch, независимо от того, было исключение, или нет (необязательный блок):

```
ComplexNumber a = new ComplexNumber(3.0,4.0);
ComplexNumber b = new ComplexNumber(1.0,2.0);
ComplexNumber c = new ComplexNumber();

try {
    ComplexNumber d = a.divide(b);
    System.out.println("d = "+d);
    ComplexNumber e = a.divide(c);
    System.out.println("e = "+e);
} catch (Exception e1) {
    // Здесь принимаются меры для борьбы с ошибкой
    //(повторный ввод,подстановка значения по умолчанию и т.д.)
    e1.printStackTrace(); // Печать стека вызовов
}
```

Рис. 4. Обработка исключения в Java.

Синтаксический анализ в Java

Любые задачи анализа текстов, в том числе и биоинформатические, начинаются с процесса считывания текстовой информации и её преобразования в некие структуры данных. Процесс анализа входной последовательности символов с целью разбора грамматической структуры называется *синтаксическим анализом* (syntax analysis, парсинг, parsing), а программа или часть программы, выполняющая синтаксический анализ, соответственно, *синтаксическим анализатором* (парсером, parser).

Язык Java предоставляет различные классы для разработки синтаксических анализаторов. Одним из наиболее простых в использовании классов для синтаксического анализа в Java является класс StringTokenizer. Термин *tokenizing* (англ. разметка, пометка) в данном случае означает процесс разбиения строки символов на последовательность

значащих элементов, слов-токенов (*tokens*), раздёленных определёнными *разделяющими символами (delimiters)*. Рассмотрим в качестве примера следующую строку:

```
double A = 10; int B = 20; long C = 30; float D = 50;
```

Если в качестве разделяющего символа мы возьмём символ `;` (точка с запятой), то разбив данную строку, мы получим следующее множество слов (токенов):

```
double A = 10
int B = 20
long C = 30
float D = 50
```

Если в качестве разделяющего символа будет выбран символ = (знак равенства), то мы получим другое множество токенов:

```
double A
10; int B
20; long C
30; float D
50;
```

Класс `StringTokenizer` реализует разбиение строки на элементы (токены) по разделяющим символам (разделители) (рис. 5):

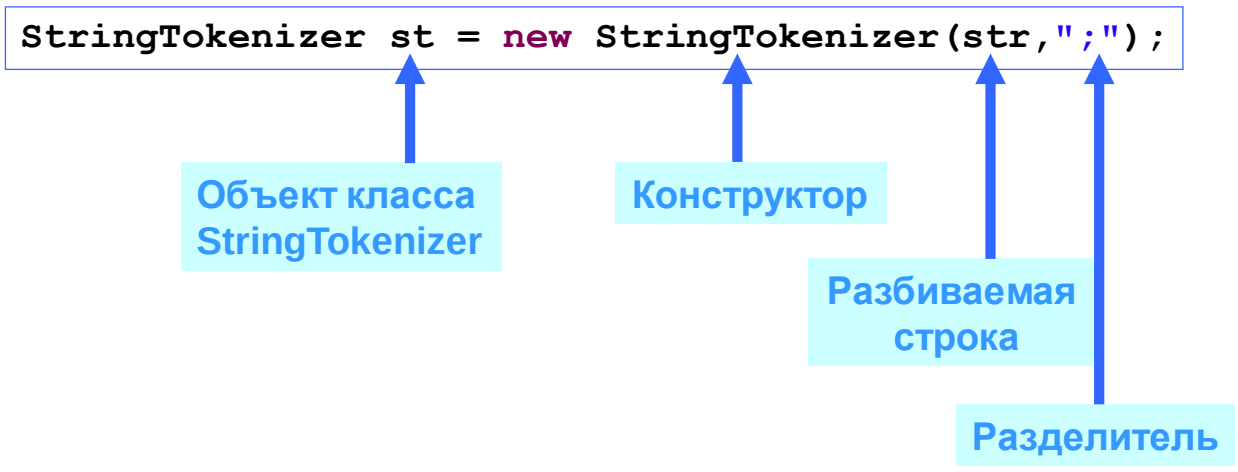

Рис. 5. Создание объекта класса `StringTokenizer` и основные сущности.

При создании объекта `StringTokenizer` в качестве первого аргумента передаётся разбиваемая строка (`str` на рис. 5), а в качестве второго аргумента – символ(-ы)-разделитель(-и). Основные методы `StringTokenizer`:

- `int countTokens()` – возвращает количество элементов (токенов) в строке. В результате выполнения следующего фрагмента кода, будет выведено число 4:

```
String str = "A = 10; B = 20; C = 30; D = 50;";
StringTokenizer st1 = new StringTokenizer(str,";");
System.out.println(“tokens "+st1.countTokens());
```

- `String nextToken()` – возвращает следующий элемент (токен) и переходит на следующую позицию. В результате выполнения следующего фрагмента кода, будет выведена строка «A = 10»:

```
String str = "A = 10; B = 20; C = 30; D = 50;";
StringTokenizer st1 = new StringTokenizer(str,";");
System.out.println("st1.nextToken(): " + st1.nextToken());
```

- `String nextToken(String delim)` – возвращает следующий элемент (токен), отделённый символом из группы `delim`, переходит на следующую позицию. В результате выполнения следующего фрагмента кода, будет выведена строка «A»:

```
String str = "A = 10; B = 20; C = 30; D = 50;";
StringTokenizer st1 = new StringTokenizer(str,";");
System.out.println("st1.nextToken(\“=\“): " +
st1.nextToken(“=”));
```

- `boolean hasMoreTokens()` – возвращает `true`, если в строке ещё есть непрочитанные токены, иначе возвращает `false`. Часто используется при организации циклов. Например, в результате выполнения следующего фрагмента кода, последовательно будут выведены все токены строки `str`:

```
String str = "A = 10; B = 20; C = 30; D = 50;";
StringTokenizer st1 = new StringTokenizer(str,";");
while (st1.hasMoreTokens()){
    System.out.println(st1.nextToken());
}
```

Принцип работы этого фрагмента программы проиллюстрирован ниже на рисунке 6:

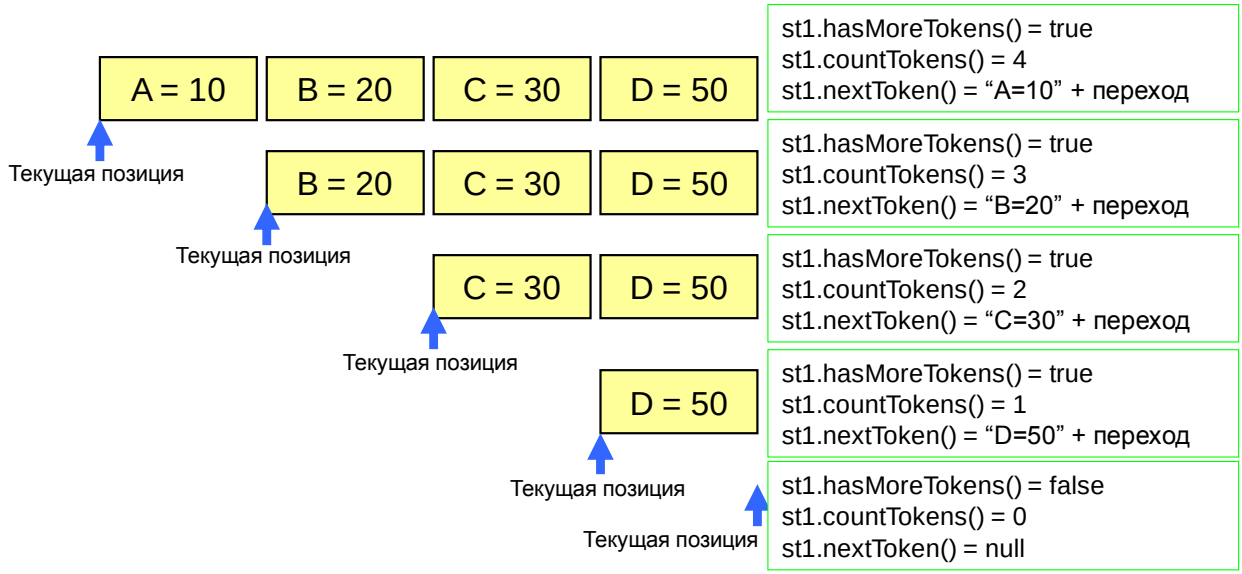

Рис. 6. Принцип работы класса StringTokenizer и его основных методов при работе цикла `while(st1.hasMoreTokens())`. Верхняя строка – первая итерация цикла, последующие строки – последующие итерации.

Рассмотрим пример простой программы, считывающей из файла список генов, с указанием названия гена, номера хромосомы, координат гена в хромосоме и других данных, как показано в следующей строке:

```
name = dnaA; chromosome = 1; start = 100; stop = 300;
plus = true; seq = "augccagcattg";
```

Для хранения всей этой информации нам необходим дополнительный класс, который мы назовём `SimpleGene`:

```
public class SimpleGene {
	private String name;	// Название гена
	private int chromosome;	// Номер хромосомы
	private int start;	// Позиция начала гена
	private int stop;	// Позиция конца гена
	private boolean isOnPlusChain; // Находится на плюс цепи

	private String sequence; // Нуклеотидная последовательность гена

	public SimpleGene(){}; // Пустой конструктор

	public SimpleGene(String _name, int _chrom, int _start, int _stop, boolean
_plus, String _seq){
		this.name = _name;
		this.chromosome = _chrom;
		this.start = _start;
		this.stop = _stop;
		this.isOnPlusChain = _plus;
		this.sequence = _seq;
	}

	@Override
	public String toString() {
		return "SimpleGene [name=" + name + ", chromosome=" + chromosome + ",
start=" + start + ", stop=" + stop + ", isOnPlusChain=" + isOnPlusChain + ",
sequence=" + sequence + "]";
	}
	// Другие методы
}
```

Класс `SimpleGene` представляет собой, по сути, «обёртку» для данных, предоставляя только конструктор, методы редактирования/доступа к полям класса, а также сервисный метод `toString()`. Для использования в

реальных приложениях этот класс, конечно, необходимо будет дополнить дополнительными «функционально нагруженными» методами, осуществляющими различные проверки и т.д. Тем не менее, объекты класса `SimpleGene` могут хранить простейшую информацию о генах и, что является для нас важным в данном примере, эти объекты могут храниться в объектах-контейнерах Java, таких, например, как `List<>`. Следующая программа показывает считывание из файла информации о генах, сохранение этой информации в объектах `SimpleGene` и сохранение уже этих объектов в контейнере `ArrayList<>`:

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;

public class TokenizerExample {

public static void main(String[] args) {
	String fileName = "F:/file.txt";

	FileReader fr;
	try {
		fr = new FileReader(fileName);
		BufferedReader reader = new BufferedReader(fr);
		String parseLine = "";

		// Создаём список, куда будем помещать "гены"
		List<SimpleGene> genes = new ArrayList<SimpleGene>();

		// Читаем строки, пока не кончился файл
		while ((parseLine = reader.readLine()) != null) {
			// Новая строка - новый ген
			SimpleGene gene = new SimpleGene();

			StringTokenizer st = new StringTokenizer(parseLine, ";");
```

```
                    // Обработка отдельных токенов st (обработка команд)
                    while (st.hasMoreTokens()) {
                          StringTokenizer st2 = new
StringTokenizer(st.nextToken(),"=");
                          // То, что слева от "="
                          String command = st2.nextToken();
                          // То, что справа от "="
                          String value = st2.nextToken();

                          if (command.trim().equalsIgnoreCase("name")) {
                                gene.setName(value.trim());
                          }
                          if (command.trim().equalsIgnoreCase("chromosome")){
                                gene.setChromosome(Integer.
                                            parseInt(value.trim()));
                          }
                          if (command.trim().equalsIgnoreCase("start")) {
                                gene.setStart(Integer.
                                            parseInt(value.trim()));
                          }
                          if (command.trim().equalsIgnoreCase("stop")) {
                                gene.setStop(Integer.
                                            parseInt(value.trim()));
                          }
                          if (command.trim().equalsIgnoreCase("plus")) {
                                gene.setOnPlusChain(Boolean.
                                      parseBoolean(value.trim()));
                          }
                          if (command.trim().equalsIgnoreCase("seq")) {
                                gene.setSequence(value.trim());
                          }
                    } // Конец обработки токенов st

                    // "Ген готов", помещаем его в список
                    genes.add(gene);

                    // Выводим информацию о гене
                    System.out.println(gene);

              } // Конец файла
```

```
            reader.close(); // Закрываем поток/файл

            // Выводим статистику:
            System.out.println("Обработано\t" + genes.size() + "\tгена(ов)");

        } catch (FileNotFoundException e) {
            // Что-то сделать, если не найден файл
            e.printStackTrace();
        } catch (IOException e) {
            // Что-то сделать, если ошибка ввода
            e.printStackTrace();
        }
        }
}
```

Обратим внимание, что в данной программе используется два объекта `StringTokenizer`: `st` и `st2`. С помощью `st` мы осуществляем «крупноблочную» разбивку считанной строки – в результате этой разбивки, строка:

```
name = dnaA; chromosome = 1; start = 100; stop = 300; plus = true; seq = "augccagcattg";
```

будет разбита на следующие токены:

```
name = dnaA
chromosome = 1
start = 100
stop = 300
plus = true
seq = "augccagcattg"
```

Каждый полученный токен, в свою очередь, может быть разобран на токены более низкого уровня, с использованием разделителя `=` (знак равенства), как это и сделано в случае с `st2`. Также стоит обратить внимание на метод `trim()`, применяемый, в частности, для токенов `value`. Этот метод «обрезает» непечатные символы (пробелы, символы табуляции и т.д.,

что в английской терминологии носит название *whitespaces*) по обоим краям строки, что является необходимым для корректного разбора строки методами вида `Integer.parseInt`, `Double.parseDouble` и т.д.

Как мы уже сказали, при синтаксическом анализе исходный текст преобразуется в некую структуру данных. Поэтому, ещё до стадии написания синтаксического анализатора программист должен продумать основные структуры данных (классы), которые будут описывать предметную область решаемой им задачи. Например, при написании синтаксического анализатора аннотированных геномов, записанных в формате GenBank, представляется логичным описать классы, содержащие информацию об аннотированных генах: название гена, его координаты в хромосоме, его нуклеотидную последовательность и т.д.

Организация программных пакетов в Java

Использование сторонних разработок в собственной работе – неотъемлемая часть процесса разработки программного обеспечения. Наиболее часто используемым механизмом включения сторонних разработок в собственный проект является механизм *библиотек*, или, в терминах Java – *пакетов.* Механизм пакетов позволяет организовывать Java классы в *пространства имён* (аналогично модулям в других языках программирования). Основные пакеты стандартной версии платформы Java (Java Standard Edition, Java SE) с кратким описанием функциональности пакета приведены в таблице 1:

Таблица 1. Основные пакеты Java SE.

Имя пакета	Краткое описание функциональности
java.lang	Базовая функциональность языка и основные типы данных (классы)
java.util	Классы коллекций и вспомогательных структур данных

java.io	Операции ввода-вывода, в том числе файлового
java.math	Математические операции высокой точности
java.nio	Расширенная (новая – New I/O) функциональность ввода-вывода
java.net	Операции и сетью, сокеты, протоколы и др.
java.security	Безопасность: генерация ключей, шифрование/дешифрование
java.sql	Работа с базами данных
java.awt	Иерархия пакетов для построения графического интерфейса (базовые компоненты)
javax.swing	Иерархия пакетов для построения платформо-независимого графического интерфейса (расширенные компоненты)

Для использования класса из пакетов, необходимо подключить этот класс с помощью команды `import`:

```
// Импорт класса поддержки больших чисел
import java.math.BigInteger;
```

Для подключения всех классов пакета используется символ звёздочки (*):

```
// Импорт всех классов пакета java.math
import java.math.*;
```

Для создания своего собственного пакета необходимо использовать ключевое слово `package имя.пакета` в начале файла, в котором содержится класс. Файл исходного текста в языке Java, называется *модулем*

компиляции (compilation unit). Каждый модуль компиляции должен иметь расширение `.java`, и внутри него может быть расположен публичный (`public`) класс, имя которого должно совпадать с именем файла (учитывая регистры, но без расширения .java). В каждом модуле компиляции может быть только один публичный класс, в противном случае, компиляция будет невозможна. Остальные классы в этом модуле компиляции, если они есть, скрыты от мира за пределами этого пакета, т.к. они не публичные, и представляют классы «поддержки» для главного публичного класса. При компиляции java-файла получается выходной файл с точно таким же именем, но с расширением `.class`, то есть из нескольких java-файлов получается несколько class-файлов. Собственно говоря, работающая Java-программа и есть набор class-файлов, которые могут быть собраны в пакет и запакованы в JAR-файл (с помощью специального архиватора jar для языка Java). А интерпретатор Java способен находить, загружать и интерпретировать эти файлы. Вообще говоря, Java-библиотека тоже представляет собой набор class-файлов. Каждый файл содержит один публичный класс, и по сути составляет один компонент. Именно для того, чтобы все компоненты из различных java- и class-файлов и хранились вместе, используется ключевое слово `package`, например:

```
package testpackage;
public class TestClass{
 // Тело класса…
}
```

Теперь, если кто-то захочет использовать класс `TestClass` или любой другой публичный класс из пакета `testpackage`, ему нужно будет использовать ключевое слово `import` чтобы сделать доступными имена из этого пакета.

Необходимо особо отметить, что для именования пакетов используется соглашение, которое почти наверняка исключает появление дублирующих имён при выкладывании пакета в общий доступ и последующем его

использовании совместно с пакетами других производителей. Идея заключается в использовании в наименовании пакета уникального доменного имени, характерного для разработчика(-ков) данного пакета. Например, если некий пакет `BioPackage` разрабатывает в Лаборатории теоретической генетики Института цитологии и генетики СО РАН, которая имеет интернет-адрес: bionet.nsc.ru/theorylab, то подходящим полным названием для такого пакета будет:

```
package ru.nsc.bionet.theorylab.BioPackage;
```

Таким образом, практически наверняка исключается конфликт имён с пакетами `BioPackage` других производителей (если они, конечно, назовут так свои пакеты).

Компьютерное представление биологических данных

Биология, пожалуй, является наукой с наиболее широким диапазоном представленных данных. К биологическим данным относятся и биометрические данные (такие как рост, вес и другие измеряемые параметры), данные молекулярной биологии (данные о генетических и белковых последовательностях, сайтах связывания различных молекул, пространственной организации молекул), данные биохимии (различные биохимические константы, такие как константа Михаэлиса), данные о структурной организации генетических процессах (генные сети) и тому подобное. Генерируемые в настоящее время массивы экспериментальных биологических данных невозможно проанализировать без использования современных вычислительных средств, что и определяет растущий интерес к информационной биологии, а также стремительный прогресс в этой области. В настоящее время существует множество форматов представления биологических данных самых разных уровней организации

FASTA формат используется для представления нуклеотидных или аминокислотных последовательностей в обычном текстовом виде (plain text). Каждый файл в этом формате может содержать одну или несколько

последовательностей. Последовательность в FASTA формате представляется в строках, длина каждой их которых не должна превышать 120 символов (обычно не превышают 80 символов). Первым символом в FASTA-файле должен быть либо символ «больше» (>), либо символ «точка с запятой» (;) с последующим далее кратким комментарием. После чего следуют непосредственно строки, содержащие последовательность. Каждой новой последовательности предшествует строка с комментарием, также начинающаяся с символов > (рис. 7).

```
>gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTNLV
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDFLG
LLILILLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVLALFLSIVIL
GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAFLPIAGX
IENY
```

Рис. 7. Пример файла в FASTA-формате: последовательность белка цитохром-б индийского слона (*Elephas maximus*).

Строка, начинающаяся с символа >, так называемая заголовочная строка, содержит название последовательности и/или её уникальный идентификатор в одной из общедоступных баз данных (см. таблицу 2):

Таблица 2. Формат уникальных идентификаторов и соответствующих баз данных биологических последовательностей.

База данных	Идентификатор
GenBank	gi|*gi-номер*|gb|*образец*|*локус*
EMBL, Data Library	gi|*gi-номер*|emb|*образец*|*локус*
DDBJ, DNA Database of Japan	gi|*gi-номер*|dbj|*образец*|*локус*
NBRF PIR	pir||*запись*
Protein Research Foundation	prf||*имя*
SWISS-PROT	sp|*образец*|*имя*
Brookhaven Protein Data Bank (1)	pdb|*запись*|*цепь*
Brookhaven Protein Data Bank	*запись*:*цепь*|PDBID|CHAIN|SEQUENCE

(2)	
Patents	`pat|страна|номер`
GenInfo Backbone Id	`bbs|номер`
Общий идентификатор базы данных	`gnl|база даных|идентификатор`
NCBI Reference Sequence	`ref|образец|локус`
Local Sequence identifier	`lcl|идентификатор`

Следующая таблица показывает принятые расширения для FASTA-файлов (данные расширения не обязательны, но желательны):

Таблица 3. Принятые расширения для FASTA-файлов.

Расширение	Содержание	Примечание
fasta, fa, seq, fsa	общий FASTA	Любой FASTA-файл
fna	нуклеиновые кислоты	Для кодирующих районов конкретных геномов следует использовать расширение ffn, данное расширение полезно для остальных типов нуклеиновых кислот
ffn	нуклеотидные кодирующие районы FASTA	Содержит кодирующие участки геномов
faa	аминокислоты	Содержит аминокислотную последовательность. Для файлов с несколькими последовательностями также используется расширение mpfa
frn	некодирующие РНК	Содержит некодирующие участки генома (например, тРНК, рРНК)

Более совершенный формат был предложен Национального центра биотехнологической информации США (National Center for Biotechnology

Information – NCBI, http://www.ncbi.nlm.nih.gov/) для базы данных, содержащей последовательности ДНК и/или аминокислотные последовательности (в настоящее время одна из крупнейших баз данных такого рода). Файл в формате GenBank содержит информацию о последовательности в обычном текстовом виде (plain text), а также дополнительную информацию, о характере последовательности, о публикациях и авторах описывающих последовательность, разметку последовательности (для полногеномных последовательностей). Пример сервисной части GenBank-файла, описывающей аннотированный организм, а также основные публикации по последовательности, показан ниже (рис. 8):

```
LOCUS       NC_009782          2880168 bp    DNA     circular BCT 03-MAY-2010
DEFINITION  Staphylococcus aureus subsp. aureus Mu3, complete genome.
ACCESSION   NC_009782
VERSION     NC_009782.1  GI:156978331
DBLINK      Project:18509
KEYWORDS    .
SOURCE      Staphylococcus aureus subsp. aureus Mu3
  ORGANISM  Staphylococcus aureus subsp. aureus Mu3
            Bacteria; Firmicutes; Bacillales; Staphylococcus.
REFERENCE   1  (bases 1 to 2880168)
  AUTHORS   Neoh,H.M., Cui,L., Yuzawa,H., Takeuchi,F., Matsuo,M. and
            Hiramatsu,K.
  TITLE     Mutated response regulator graR is responsible for phenotypic
            conversion of Staphylococcus aureus from heterogeneous
            vancomycin-intermediate resistance to vancomycin-intermediate
            resistance
  JOURNAL   Antimicrob. Agents Chemother. 52 (1), 45-53 (2008)
   PUBMED   17954695
```

Рис. 8. Сервисная часть GenBank-файла, содержащая информацию о локусе (LOCUS), организме (SOURCE, ORGANISM) и типе генома (полный геном – complete genome) (DEFINITION), а также о связанных публикациях (REFERENCE), авторах (AUTHORS), названии стать (TITLE), журнале (JOURNAL).

```
source      1..2880168
            /organism="Staphylococcus aureus subsp. aureus Mu3"
            /mol_type="genomic DNA"
            /strain="Mu3"
            /sub_species="aureus"
            /db_xref="taxon:418127"
  gene      517..1878
            /gene="dnaA"
            /locus_tag="SAHV_0001"
            /db_xref="GeneID:5559096"
  CDS       517..1878
            /gene="dnaA"
            /locus_tag="SAHV_0001"
            /note="binds to the dnaA-box as an ATP-bound complex at
            the origin of replication during the initiation of
            chromosomal replication; can also affect transcription of
            multiple genes including itself."
            /codon_start=1
            /transl_table=11
            /product="chromosomal replication initiation protein"
            /protein_id="YP_001440591.1"
            /db_xref="GI:156978332"
            /db_xref="GeneID:5559096"
```

Рис. 9. Информативная часть GenBank-файла, содержащая разметку генома (`gene`), включая его координаты в хромосоме (517..1878), название (dnaA) и другую информацию.

Информационная часть GenBank-файла содержит разметку генома в виде списка генов. Для каждого гена прописывается его положение в хромосоме (локализация на минус-цепи помечается словом `complement`), его имя (`/gene`), ссылки на базы данных (`/db_xref`), а также координаты транслируемой части гена (`CDS` – coding DNA sequence) и соответствующую аминокислотную последовательность (рис. 9). Поскольку не все гены являются белок-кодирующими, то такие нетранслируемые гены (гены транспортных и рибосомных РНКА – tRNA, rRNA) не имеют полей `CDS` и `/translation` (рис. 10).

```
gene        14011..14087
            /locus_tag="SACE_8001"
            /note="tRNA-Ile(GAT)"
            /db_xref="GeneID:4939872"
  tRNA        14011..14087
            /locus_tag="SACE_8001"
            /product="tRNA-Asp"
            /codon_recognized="GAU"
            /db_xref="GeneID:4939872"
```

Рис.10. Для генов транспортной РНК отсутствуют поля `CDS` и `/translation`.

В конце GenBank-файла, после тега `ORIGIN` приводится полная нуклеотидная последовательность аннотированного участка генома, каждая строка в данной части файла содержит строку последовательности длиной 60 нуклеотидов (6 блоков по 10 нуклеотидов); также каждая строка показывает позицию в последовательности, соответствующую данной строке (рис. 11):

```
ORIGIN
        1 gaattccggc ctgctgccgg gccgcccgac ccgccgggcc acacggcaga gccgcctgaa
       61 gcccagcgct gaggctgcac ttttccgagg gcttgacatc agggtctatg tttaagtctt
      121 agctcttgct tacaaagacc acggcaattc cttctctgaa gccctcgcag ccccacagcg
      181 ccctcgcagc cccagcctgc
```

Рис. 11. Нуклеотидная последовательность в GenBank-файле.

Типичными расширениями для GenBank файлов являются `gb` и `gbk`.

Программные библиотеки обработки биологических данных

BioJava (www.biojava.org) представляет собой проект с открытым исходным кодом (open-source) и предоставляет платформу для обработки биологических данных, включающую в себя объекты для работы с биологическими последовательностями, разборщики файлов, поддержку распределённой системы аннотации (Distributed Annotation System – DAS), доступ к базам данных BioSQL и Ensembl, а также средства для визуального и статистического анализа последовательностей и другие средства.

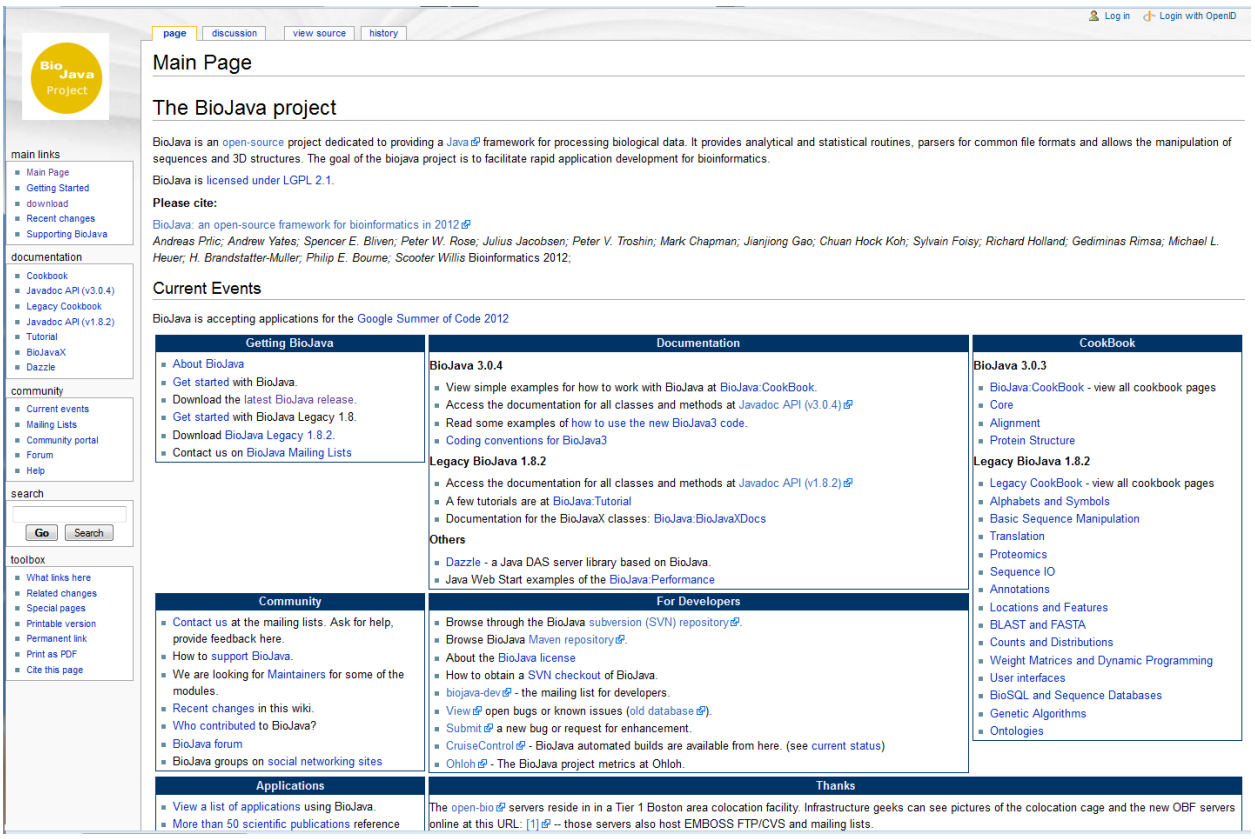

Рис. 12. Сайт проекта BioJava (www.biojava.org).

Пакеты BioJava разделяются на 10 основных категорий, как показано в таблице ниже:

Таблица 4. Группы пакетов BioJava и их функциональность.

Группа пакетов	Функциональность пакета
Core biological processes (Базовые биологические процессы)	Основные классы для работы с последовательностями (выравнивание, сравнение, использование различные алфавитов, алгоритмы вероятностного анализа биологических последовательностей и др.)
Sequence databases and formats (поддержка баз данных и форматов последовательностей)	Драйверы для работы с различными базами данных
User interface components (компоненты графического	Графические классы для работы с объектами данных biojava (визуализация

интерфейса)	последовательностей и разметок)
Dynamic programming packages (пакеты динамического программирования)	Классы для работы с марковскими моделями и алгоритмами динамического программирования
Chromatogram and trace file support (поддержка хроматограмм и файлов трассировки	Классы для работы с файлами хроматограмм, генерируемых секвенирующим оборудованием; средства для визуализации хроматограмм
Macromolecular structure (макромолекулярные структуры)	Классы для работы с белками и белковыми структурами (PDB), алгоритмы структурного выравнивания
Utilities and developers' packages (утилиты и пакеты для разработки)	Классы для работы с файлами, XML-форматами, онтологиями, регулярными выражениями, математические утилиты и работа с сетью
Molecular biology packages (пакеты молекулярной биологии)	Классы для поддержки экспериментальных методов молекулярной биологии, таких как рестрикционное картирование и ПЦР
Experimental packages (Экспериментальные пакеты)	Экспериментальные классы, расширяющие функциональность за счёт новых технологий
Biojava extension (biojavax) packages (пакеты расширения biojava)	Пакеты расширения для Biojava, а также классы для связи biojava объектов с базами данных biosql и данными NCBI
Genetic Algorithm Framework (генетические алгоритмы)	Классы для работы с генетическими алгоритмами
Experiment Phylogeny packages	Классы для работы с алгоритмами филогении, включая экспорт и импорт формата PHYLIP

(экспериментальные пакеты для филогении)	
Другие пакеты	Классы для работы с различными форматами биологических данных, классы встраивания сторонних графических утилит (например, Jmol) и др.

Классы для хранения и обработки последовательностей в BioJava спроектированы и реализованы для достижения больше эффективности по сравнению с классами других библиотек, рассматривающих последовательности в качестве строковых объектов или массивов символов. Экономия памяти и скорость работы с объектами последовательностей в BioJava обусловлены использованием специальных классов в пакетах `org.biojava.bio.symbol` и `org.biojava.bio.seq`. Это классы символов, реализующие интерфейсы `Symbol` (Символ, см. листинг ниже) и `Alphabet` (Алфавит).

```
public interface Symbol {
    public String getName();
    public Annotation getAnnotation();
    public Alphabet getMatches();
}
```

Следующий фрагмент кода показывает создание алфавита ДНК и последовательный вывод слов `guanine`, `thymine`, `cytosine`, `adenine`:

```
FiniteAlphabet dna = DNATools.getDNA();
Iterator dnaSymbols = dna.iterator();
while (dnaSymbols.hasNext()) {
    Symbol s = (Symbol) dnaSymbols.next();
    System.out.println(s.getName());
}
```

Базовыми классами, описывающими нуклеотидные последовательности в BioJava являются классы `DNASequence` и `RNASequence`. Первый из этих классов имеет метод `getRNASequence`, который «транскрибирует ДНК-последовательность в РНК», т.е., с точки зрения программирования, возвращает объект класса `RNASequence`. Аргументом этого метода является объект класса `Frame`, указывающий способ транскрипции/трансляции последовательности. Всего возможны шесть таких способов: `ONE`, `TWO`, `THREE`, `REVERSED_ONE`, `REVERSED_TWO`, `REVERSED_THREE`, соответствующих различным рамкам считывания и направлению транскрипции. Ниже приведёны: пример программы, использующей классы `DNASequence` и `RNASequence` и результат работы этой программы (рис. 13):

```
import org.biojava3.core.sequence.DNASequence;
import org.biojava3.core.sequence.RNASequence;
import org.biojava3.core.sequence.transcription.Frame;
public class BioJavaTranscriptionExample {

	public static void main(String[] args) {
	    DNASequence dna = new DNASequence("AATGAATCGAATGAATGAATG");
	    RNASequence[] rna = new RNASequence[6];
	    rna[0] = dna.getRNASequence(Frame.ONE);
	    rna[1] = dna.getRNASequence(Frame.TWO);
	    rna[2] = dna.getRNASequence(Frame.THREE);
	    rna[3] = dna.getRNASequence(Frame.REVERSED_ONE);
	    rna[4] = dna.getRNASequence(Frame.REVERSED_TWO);
	    rna[5] = dna.getRNASequence(Frame.REVERSED_THREE);

	    for(RNASequence r : rna){
	        System.out.println(r);
	    }
	}
}
```

```
AAUGAAUCGAAUGAAUGAAUG
AUGAAUCGAAUGAAUGAAUG
UGAAUCGAAUGAAUGAAUG
CAUUCAUUCAUUCGAUUCAUU
AUUCAUUCAUUCGAUUCAUU
UUCAUUCAUUCGAUUCAUU
```

Рис. 13. Результат работы программы `BioJavaTranscriptionExample`.

Аналогично методу `getRNASequence`, в классе `RNASequence` имеется метод `getProteinSequence`, который возвращает объект класса `ProteinSequence`, предназначенный для описания белковых последовательностей (т.е. «транслирует» РНК-последовательность). Пример программы трансляции РНК и результат работы этой программы (рис. 14) приведены ниже:

```
import org.biojava3.core.sequence.DNASequence;
import org.biojava3.core.sequence.ProteinSequence;
import org.biojava3.core.sequence.RNASequence;
import org.biojava3.core.sequence.transcription.Frame;

public class BioJavaTranslationExample {
	public static void main(String[] args) {
	    DNASequence dna = new DNASequence("AATGAATCGAATGAATGAATG");
	    RNASequence rna1 = dna.getRNASequence(Frame.ONE);
	    RNASequence rna2 = dna.getRNASequence(Frame.TWO);
	    ProteinSequence protein1 = rna1.getProteinSequence();
	    ProteinSequence protein2 = rna2.getProteinSequence();

	    System.out.println("РНК1\t"+rna1);
	    System.out.println("РНК2\t"+rna2);
	    System.out.println("Белок1\t"+protein1);
	    System.out.println("Белок2\t"+protein2);
	}
}
```

```
РНК1 AAUGAAUCGAAUGAAUGAAUG
РНК2 AUGAAUCGAAUGAAUGAAUG
Белок1 NESNE*M
Белок2 MNRMNE
```

Рис. 14. Результат работы программы `BioJavaTranslationExample`.

Метод `getProteinSequence` позволяет «транслировать» РНК-последовательности, пользуясь различными таблицами трансляции (список возможных вариантов которых приведён в таблице 5). Для этого в BioJava используется вспомогательный класс `TranscriptionEngine`, использование которого показано в следующем листинге:

```
TranscriptionEngine.Builder b = new TranscriptionEngine.Builder();
b.table(2); // Таблица №2
TranscriptionEngine te2 = b.build();
```

Метод `table` устанавливает трансляционную (кодонную) таблицу в соответствии с её числовым или строковым идентификатором (см. Табл. 5). С другими методами этого класса, предназначенными для тонкой настройки трансляционных правил, можно ознакомиться на сайте BioJava API (http://www.biojava.org/docs/api/overview-summary.html).

Таблица 5. Кодонные таблицы BioJava

Код	Таблица	Код	Таблица
1	UNIVERSAL	11	BACTERIAL
2	VERTEBRATE_MITOCHONDRIA	12	ALTERNATIVE_YEAST_NUCLEAR
3	YEAST_MITOCHONDRIAL	13	ASCIDIAN_MITOCHONDRIAL
4	MOLD_MITOCHONDRIAL	14	FLATWORM_MITOCHOND

			RIAL
5	INVERTEBRATE_MITOCHONDRIAL	15	BLEPHARISMA_MACRONUCLEAR
6	CILIATE_NUCLEAR	16	2CHLOROPHYCEAN_MITOCHONDRIAL
9	ECHINODERM_MITOCHONDRIAL	21	TREMATODE_MITOCHONDRIAL
10	EUPLOTID_NUCLEAR	23	SCENEDESMUS_MITOCHONDRIAL

Пример построения белковых последовательностей с использованием разных кодонных таблиц и результат работы программы (рис. 15), приведёны ниже:

```
import org.biojava3.core.sequence.DNASequence;
import org.biojava3.core.sequence.ProteinSequence;
import org.biojava3.core.sequence.RNASequence;
import org.biojava3.core.sequence.transcription.Frame;
import org.biojava3.core.sequence.transcription.TranscriptionEngine;
public class BioJavaTranslationExample2 {

	public static void main(String[] args) {
	    DNASequence dna = new DNASequence("ACCGCCTGAATCGAATGAATGAATGTTT");
	    RNASequence rna1 = dna.getRNASequence(Frame.ONE);

	    TranscriptionEngine teDefault = TranscriptionEngine.getDefault();
	    TranscriptionEngine.Builder b = new TranscriptionEngine.Builder();
	    b.table(2).initMet(true).trimStop(true);
	    TranscriptionEngine te2 = b.build();

	    ProteinSequence protein1 = rna1.getProteinSequence(teDefault);
	    ProteinSequence protein2 = rna1.getProteinSequence(te2);

	    b.table(10).initMet(true).trimStop(true);
	    TranscriptionEngine te10 = b.build();
	    ProteinSequence protein3 = rna1.getProteinSequence(te10);
```

```
            System.out.println("РНК1\t"+rna1);
            System.out.println("Белок1 (Трансл. таблица 1)\t"+protein1);
            System.out.println("Белок2 (Трансл. таблица 2)\t"+protein2);
            System.out.println("Белок3 (Трансл. таблица 10)\t"+protein3);
        }
}
```

```
РНК1    UCCAGACCGCCUGAAUCGAAUGAAUGAAUGUUU
Белок1  (Трансл. таблица 1)         SRPPESNE*MF
Белок2  (Трансл. таблица 2)         S*PPESNEWMF
Белок3  (Трансл. таблица 10)        SRPPESNECMF
```

Рис. 15. Результат работы программы `BioJavaTranslationExample2`.

Принципы разработки комплексных приложений для анализа биологических данных

Несмотря на большое количество библиотек, пакетов и API, разработанных для решения задач информационной биологии, остаётся ещё достаточно много областей, не покрываемых библиотеками. Причём часто возникают ситуации, когда для решения определённой задачи существуют средства, которые, однако, крайне затруднительно непосредственно использовать в своих разработках. Например, программный комплекс создаётся на языке Java, а библиотеки существуют для другого языка (Fortran, Perl и т.д.). С одной стороны, проблема частично решается с помощью механизма JNI (Java Native Interface), а также с помощью специальных библиотек, облегчающих интеграцию разных языков. С другой стороны, иногда такая интеграция требует существенных затрат: временных, компьютерных и т.д., неоправданных для разработчика. Кроме того, нередки ситуации, когда задача решается с помощью программы, но соответствующей библиотеки, позволяющей использовать такую разработку в собственном проекте, не существует. Радикальным способом решения

описанной проблемы является непосредственный запуск сторонней программы с интересующими нас входными данными, и последующий анализ выходных данных этой программы.

Механизмом, облегчающем запуск сторонне программы в языке Java, является поддержка *многопоточности*. Многопоточность – это свойство операционной системы или приложения, состоящее в том, что *процесс*, порождённый в операционной системе, может состоять из нескольких *потоков* (threads), выполняющихся «параллельно», то есть без предписанного порядка во времени. Такие потоки называют также *потоками команд* (в противоположность потокам данных) или *потоками выполнения* (threads of execution); иногда называют «нитями» («тредами»).

Непосредственно запуск программы в языке Java осуществляется с помощью класса `ProcessBuilder`, который помимо запуска обеспечивает ещё и управление внешними процессами (программами). Для запуска процесса необходимо создать объект класса `ProcessBuilder`, указав имя выполняемой программы и необходимые аргументы. После этого надо вызвать метод `start()`, который запустит программу, вернув при этом ссылку на объект класса `Process`, который и будет отслеживать состояние запущенной программы. Метод `waitFor()` класса `Process` приостанавливает выполнение исходной программы (которая вызывает стороннюю) до конца выполнения сторонней программы. Код выхода сторонней программы можно получить с помощью метода `exitValue()` класса `Process`. Пример ниже демонстрирует запуск программы notepad (Блокнот Windows) в ходе выполнения Java-программы (Принципиальная схема использования сторонних программ показана на рисунке 16):.

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

      ProcessBuilder pb = new ProcessBuilder("notepad");
      try {
            Process proc = pb.start();
          OKFlag = process.waitFor();
```

```
            OKFlag = process.exitValue();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
}
```

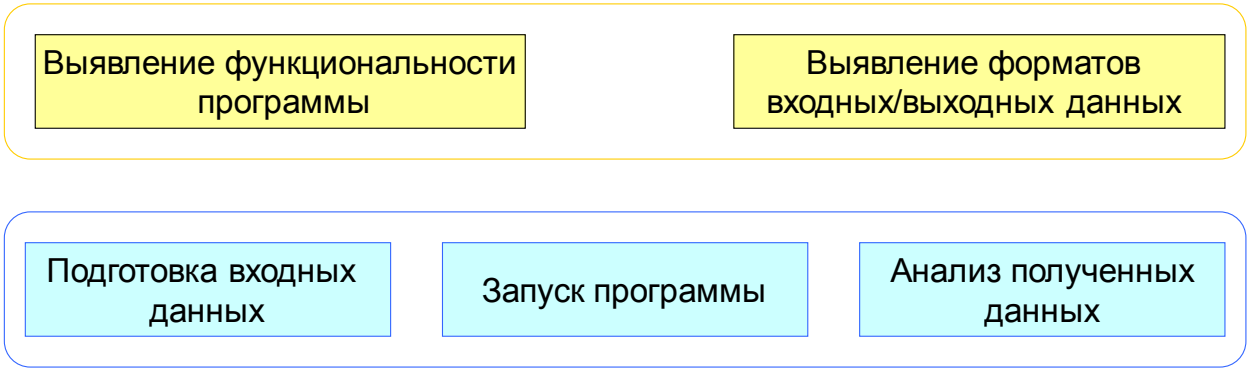

Рис. 16. Схема использования сторонней программы. Вверху – этап анализа, внизу – этап реализации.

8. Примеры контрольных работ и экзаменов

Примеры задач для первой контрольной работы

Вариант 1

1. Считать из файла нуклеотидные последовательности, найти самую короткую последовательность, самую длинную последовательность, посчитать среднюю длину последовательности.
2. В директории находятся файлы с нуклеотидными последовательностями следующего формата:

   ```
   > Организм(вид)
   > Название гена
   Последовательность
   ```

   Считать все такие файлы и сгенерировать на их основе список организмов (видов), где с каждым организмом (видом) ассоциирован набор генов и последовательностей.
3. Имеется массив строк. Найти наибольшую общую подстроку для всех строк.

Вариант 2

1. Считать из файла нуклеотидные последовательности, найти средний GC-состав всех последовательностей, вывести последовательности с самым высоким и самым низким GC-составом.
2. Имеется файл с парами: <название гена, длина гена>. Посчитать среднюю длину генов, дисперсию длины генов.
3. Имеется файл с биологическими (генетическими) данными. Предложить структуру данных (класс) для хранения подобных данных и написать синтаксический анализатор для данного типа файлов.

Примеры задач для второй контрольной работы

Вариант 1

1. Имеется файл в формате Genbank. Посчитать количество кодирующих и некодирующих генов в файле.
2. Имеются два файла геномов организмов в формате Genbank. Вывести гены, встречающиеся в обоих геномах.
3. В директории находятся файлы с нуклеотидными последовательностями РНК. Рассчитать вторичные структуры для всех последовательностей (с использованием сторонних программ), вывести структуру с наибольшей длиной стебля шпильки.

Вариант 2

1. Имеется файл в формате Genbank. Найти самый длинный кодирующий ген, самый длинный ген тРНК и самый длинный ген рРНК.
2. Имеются два файла геномов организмов в формате Genbank. Вывести в файлы с названием организмов гены, уникальные для этих организмов (т.е. присутствующие только в геноме одного организма).
3. В директории находятся файлы с нуклеотидными последовательностями РНК. Рассчитать вторичные структуры для всех последовательностей (с использованием сторонних программ), вывести последовательности, ранжированные по величине энергии Гиббса вторичных структур, с указанием структуры и энергии.

Вопросы для подготовки к зачёту

1. Условные переходы: конструкции if и switch.
2. Циклы for, while, do-while. Управление циклами.
3. Стандартные типа данных Java.
4. Метод main: точка входа в программу. Аргументы командной строки.

5. Понятие объекта и класса. Реализация в Java. Конструкторы.
6. Области видимости и существования имен.
7. Наследование.
8. Строки. Работа со строками.
9. Массивы. Создание и инициализация.
10. Операции ввода-вывода.
11. Работа с файлами. Чтение и запись.
12. Исключения. Генерация и перехват.
13. Использование класса Tokenizer для синтаксического анализа.
14. Контейнеры данных.
15. Пакеты в Java. Jar-архивы.
16. Основные форматы представления биологических данных
17. Формат FASTA.
18. Формат GenBank.
19. Библиотеки обработки биологических данных.
20. Пакет BioJava.

9. Методические указания к решению заданий по разработке компьютерных программ для анализа биологических данных.

Общие рекомендации

1. Анализ задачи. Поиск существующих решений целой задачи.
2. Выявление основных этапов решения задачи – подзадач.
    a. Определение основных сценариев решения задачи в виде упорядоченных списков выделенных подзадач.
    b. Поиск методов решения выделенных подзадач. Данный этап чрезвычайно важен, т.к. в настоящее время в биоинформатике создан большой задел в решении большого количества задач (геномика, транскриптомика, протеомика и т.д.). Спектр решений включает в себя готовые биоинформатические приложения, программные библиотеки, API, описанные алгоритмы и т.д.
    c. Определение конкретного набора готовых компонент, которые будут использоваться в проекте для решения подзадач (библиотек, API, алгоритмов).
    d. Определение недостающих компонент, которые будет необходимо реализовать самостоятельно.
    e. Составление списка форматов данных, используемых всеми компонентами (как готовыми, так и собственными).
3. Реализация компонент из пункта 2.c.
4. Стыковка всех частей в один программный комплекс.
    a. При необходимости, написание драйверов (переводчиков из одних форматов данных в другие).
    b. Реализация конкретных сценариев (пункт 2.a).
5. Тестирование и решение конкретных биологических задач с помощью созданного программного средства.

Пример разработки программы

Рассмотрим применение данного подхода на примере задачи поиска вторичной структуры и энергии Гиббса для короткой последовательности РНК (или ДНК), с использованием популярных и доступных программ.

В первую очередь нам необходимо провести поиск программ, осуществляющих предсказание вторичной структуры. На рисунке 17 показан список таких программ:

Single sequence structure prediction [edit]

Name	Description	Knots	Links	References
CONTRAfold	Secondary structure prediction method based on conditional log-linear models (CLLMs), a flexible class of probabilistic models which generalize upon SCFGs by using discriminative training and feature-rich scoring.	no	sourcecode webserver	[1]
KineFold	Folding kinetics of RNA sequences including pseudoknots.	yes	linuxbinary, webserver	[2][3]
MC-Fold MC-Sym Pipeline	Thermodynamics and Nucleotide cyclic motifs for RNA structure prediction algorithm. 2D and 3D structures.	yes	sourcecode, webserver	[4]
Mfold	MFE RNA structure prediction algorithm.	no	sourcecode, webserver	[5]
Pknots	A dynamic programming algorithm for optimal RNA pseudoknot prediction using the nearest neighbour energy model.	yes	sourcecode	[6]
PknotsRG	A dynamic programming algorithm for the prediction of a restricted class of RNA pseudoknots.	yes	sourcecode, webserver	[7]
RNAfold	MFE RNA structure prediction algorithm. Includes an implementation of the partition function for computing basepair probabilities and circular RNA folding.	no	sourcecode, webserver	[8] [5][9] [10][11][12]
Sfold	Statistical sampling of all possible structures. The sampling is weighted by partition function probabilities.	no	webserver	[13][14][15][16]
UNAFold	The UNAFold software package is an integrated collection of programs that simulate folding, hybridization, and melting pathways for one or two single-stranded nucleic acid sequences.	no	sourcecode	[17]

*Knots: Pseudoknot prediction, <yes|no>.

Рис. 17. Список программ предсказания вторичной структуры последовательностей РНК и ДНК (http://en.wikipedia.org/wiki/List_of_RNA_structure_prediction_software) с сайта Wikipedia.

К примеру, мы выбрали последнюю программу из списка – UNAFold. После скачивания, установки программы, а также изучения её документации, мы видим, что данная программа получает на вход файл с последовательностью РНК (ДНК), не делая различия между тимином и урацилом (т.е. всё равно, в каком алфавите будет записана последовательность, программа её поймёт). Переключение же режимов РНК и ДНК происходит с помощью ключей: "`-n RNA`" или "`-n DNA`", соответственно. В результате выполнения программы генерируется серия файлов (рис. 18):

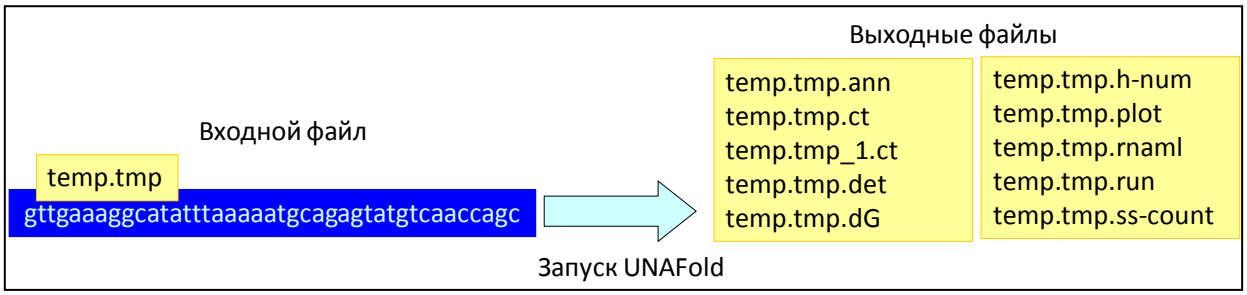

Рис. 18. Входной (temp.tmp) и выходные файлы программы UNAFold.

Число нуклеотидов | Энергия структуры | Файл исходной посл.

41	**dG = -9.1 temp.tmp**						
1	**g**	**0**	**2**	**41**	**1**	**0**	**2**
2	**t**	**1**	**3**	**40**	**2**	**1**	**3**
3	**t**	**2**	**4**	**39**	**3**	**2**	**4**
4	**g**	**3**	**5**	**38**	**4**	**3**	**5**
5	**a**	**4**	**6**	**0**	**5**	**4**	**0**
6	**a**	**5**	**7**	**0**	**6**	**0**	**0**
7	**a**	**6**	**8**	**0**	**7**	**0**	**8**
8	**g**	**7**	**9**	**34**	**8**	**7**	**9**
…							
34	**c**	**33**	**35**	**8**	**34**	**33**	**35**
35	**a**	**34**	**36**	**0**	**35**	**34**	**0**
36	**a**	**35**	**37**	**0**	**36**	**0**	**0**
37	**c**	**36**	**38**	**0**	**37**	**0**	**38**
38	**c**	**37**	**39**	**4**	**38**	**37**	**39**
39	**a**	**38**	**40**	**3**	**39**	**38**	**40**
40	**g**	**39**	**41**	**2**	**40**	**39**	**41**
41	**c**	**40**	**0**	**1**	**41**	**40**	**0**

Рис. 19. Структура ct-файла программы UNAFold. Цветные круги отмечают номера нуклеотидов в последовательности, комплементарные нуклеотиду в текущей строке (на рисунке парные нуклеотиды показаны одинаковыми цветами). Нули в этом же столбце соответствуют неспаренным нуклеотидам.

Беглый анализ файлов, полученных, полученных после выполнения программы, показывает, что наиболее «ценным» для нашей задачи является ct-файл (`temp.tmp.ct` на рис. 18). Структура данного файла достаточно очевидна и показана на рисунке 19.

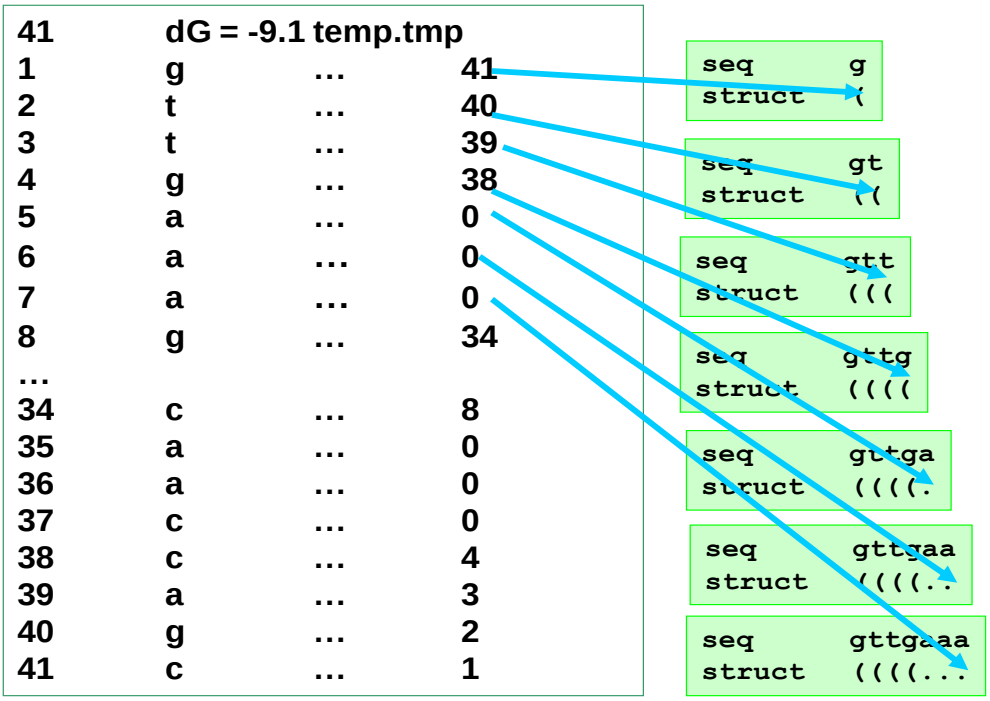

Рис. 20. Схема построения вторичной структуры в скобочном виде по ct-файлу.

Таким образом, дальнейшее построение вторичной структуры, например, в скобочном виде, становится достаточно очевидным – первые несколько шагов этого процесса показаны на рисунке 20.

Теперь мы можем реализовать алгоритм использования программы UNAFold в нашей программе, реализовав его в виде метода некоторого класса. Блок-схема алгоритма приведена на рисунке 21, а схема классов для реализации – на рисунке 22.

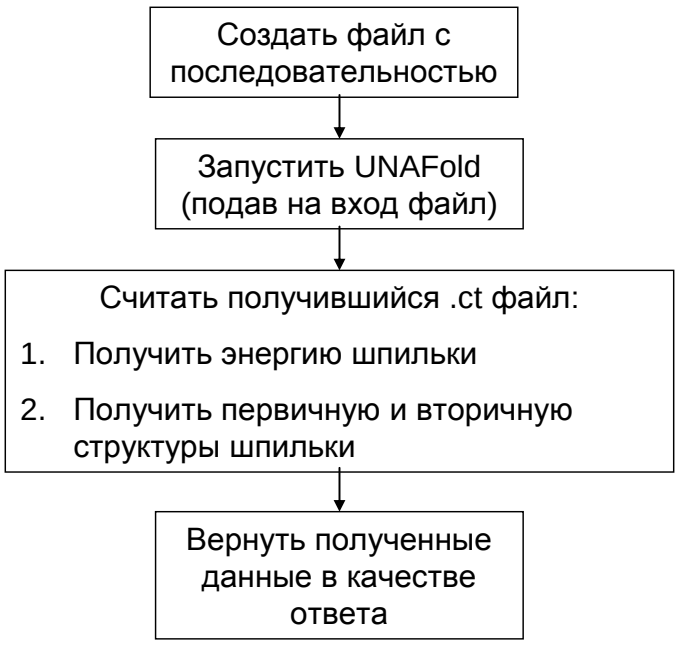

Рис. 21. Блок-схема алгоритма использования программы UNAFold для нахождения вторичной структуры нуклеотидной последовательности и её энергии Гиббса.

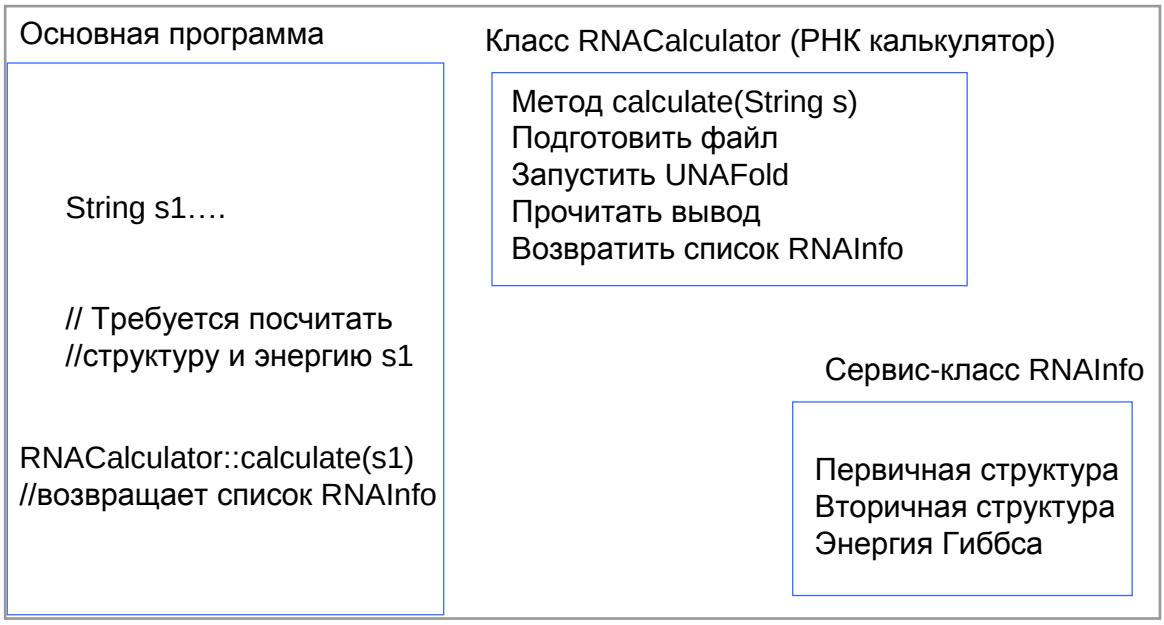

Рис. 22. Схема классов программы, использующей UNAFold для нахождения вторичной структуры нуклеотидной последовательности и её энергии Гиббса.

Как видно из рисунка выше, для решения задачи предлагается создать два класса. Первый класс – `RNACalculator`, с методом `calculate`, который и будет делать всю подготовительную работу для запуска программы UNAFold, а также сам запуск, и последующий анализ результатов. Второй класс – это маленький сервис класс `RNAInfo`, предназначенный для хранения интересующей нас информации (первичная структура, вторичная структура и энергия Гиббса) в одном объекте.

Листинг класса `RNAInfo` приведён ниже:

```
package ru.nsc.bionet.theorylab.simplernacalculator;

public class RNAInfo {
    private String sequence;//последовательность (первичная структура)
    private String structure;//вторичная структура в формате..((..))..
    private double gibbsEnergy;// Энергия Гиббса


    /**
```

```
     * Стандартный конструктор
     * @param seq - нуклеотидная последовательность
     * @param struct - вторичная структура (в формате ...(((...)))...)
     * @param gibbs - энергия Гиббса для данной вторичной структуры
     */
    public RNAInfo(String seq, String struct, double gibbs){
      this.sequence = seq;
      this.structure= struct;
      this.gibbsEnergy = gibbs;
    }
    public String getSequence() {
      return sequence;
    }

    public String getStructure() {
      return structure;
    }

    public double getGibbsEnergy() {
      return gibbsEnergy;
     }
}
```

Листинг класса `RNACalculator`:

```
package ru.nsc.bionet.theorylab.simplernacalculator;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;

public class RNACalculator {
   private static final String temperature = "25"; // "37"
   private static final String unafoldPath = "C:/UNAFold/bin";
   private static final String tempDir = "TempDirectory";
   private static final String tempFileName = "temp.tmp";

   public static List<RNAInfo> calculate(String sequence){
```

```
    int errorCode = runUNAFold(sequence);
    if(errorCode == 0){ // Т.е. нет ошибки
      return parseUNAFoldOutput();
    }
    else{
       System.err.println("UNAFold finished with error code");
       return null;
    }
}

/**
 * Запускает программу <tt>UNAFold</tt> для расчёта вторичной структуры
 * и энергии Гиббса для последовательности <tt>sequence</tt>. Программа
 * запускается в директории <tt>tempDir</tt>, имя входного файла берётся
 * из <tt>tempFileName</tt> (это статические поля данного класса).
 *
 * @param sequence - последовательность, для которой рассчитывается
 * вторичная структура
 * @return - код завершения программы <tt>UNAFold</tt>
*/
public static int runUNAFold(String sequence) {
    int OKFlag = -3000;

    File dir = new File(tempDir);
    dir.mkdirs();

    PrintWriter fileWriter;
    try {
       // Формирование файла входных данных UNAFold
       fileWriter = new PrintWriter(new File(tempDir + File.separator
          + tempFileName));
       fileWriter.print(sequence);
       fileWriter.close();

       // Запуск mfold
       ProcessBuilder pb = new ProcessBuilder(
          unafoldPath, "-n", "RNA",
          "-t", temperature, tempFileName);
       pb.directory(dir);
       pb.redirectErrorStream(true);
       Process process = pb.start();

       OKFlag = process.waitFor();
```

```
            OKFlag = process.exitValue();

            process.destroy();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return OKFlag;
    }

    public static List<RNAInfo> parseUNAFoldOutput() {
        List<RNAInfo> ans = new ArrayList<RNAInfo>();
        try {
            BufferedReader mfoldReader = new BufferedReader(new FileReader(
                tempDir + File.separator + tempFileName + ".ct"));
            String parseLine = "";
            String sequence = "";
            String secondary = "";
            double gibbsEnergy = 0.0;

            while ((parseLine = mfoldReader.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(parseLine, "\t");
                String firstWord = st.nextToken();
                String secondWord = st.nextToken();
                if (secondWord.startsWith("dG")) {
                // Начало нового варианта (вторичной структуры)
                    sequence = "";
                    secondary = "";
                    StringTokenizer st2 = new StringTokenizer(secondWord, "=");
                    st2.nextToken();
                    gibbsEnergy = Double.parseDouble(st2.nextToken());
                    // Смотрим структуру
                    int numIter = Integer.parseInt(firstWord);
                    for (int i = 0; i < numIter; i++) {
                        parseLine = mfoldReader.readLine();
                        st = new StringTokenizer(parseLine, "\t");
                        firstWord = st.nextToken();
                        secondWord = st.nextToken();
                        sequence += secondWord.trim();
                        st.nextToken(); // Пропускаем 2 столбца
```

```
                        st.nextToken();
                        int complementary = Integer.parseInt(st.nextToken());
                        if (complementary == 0) {
                              secondary += ".";
                        } else {
                              String brace = complementary > Integer
                              .parseInt(firstWord) ? "(" : ")";
                              secondary += brace;
                        }
                  } // end for(int i = 0; i < numIter; i++)
                  ans.add(new RNAInfo(sequence,secondary,gibbsEnergy));
            } // end if (secondWord.startsWith("dG"))
         } // end while ((parseLine = mfoldReader.readLine()) != null)
         mfoldReader.close();
      } catch (FileNotFoundException e) {
            e.printStackTrace();
      } catch (IOException e) {
            e.printStackTrace();
      }
      return ans;
   }
}
```

Следующий фрагмент кода показывает использование классов `RNACalculator` и `RNAInfo`:

```
String seq =
"gttgaaaggcatatttaaaaatgcagagtatgtcaaccagc";
List<RNAInfo> list = RNACalculator.calculate(seq);
// list содержит список объектов RNAInfo
```

Хотя запуск сторонних программ не является признаком хорошего стиля, иногда этот способ становится наиболее быстрым с точки зрения времени разработки. Общая схема использования сторонних программ показана на рисунке ниже:

- Анализ задачи
  - Чёткая формулировка задания. Выделение отдельных (атомарных) стадий
  - Поиск существующих решений (программ, методов, алгоритмов и т.д.)
- Анализ найденных программ (методов): изучение возможности применения
  - Анализ входных и выходных данных
  - Оценка временных и аппаратных затрат
- Встраивание сторонней программы (метода) в собственную программу
  - Написание драйверов (генераторов-«переводчиков» данных) из(в) собственного формата в(из) формат сторонней программы
  - Внедрение полученной функциональности в собственную программу

Рис. 23. Общая схема построения программы с использованием сторонних разработок.

10. Учебно-методическое и информационное обеспечение дисциплины

а) основная литература:

1. *Вирт Н.* Алгоритмы и структуры данных. // СПб.: Невский диалект, 2005.
2. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений. 3-е изд. // М.: Вильямс, 2008 г.
3. *Эккель Б.* Философия Java. Библиотека программиста. 4-е изд. // СПб.: Питер, 2009.
4. *Дурбин Р., Эдди Ш., Крог А., Митчисон Г.* Анализ биологических последовательностей // РХД, 2006 г., 480 стр.
5. *Леск А.* Введение в биоинформатику // М.: Бином. Лаб. знаний, 2009., 318 стр.

б) дополнительная литература:

1. П. Ноутон, Г.Шилдт. Java 2. Наиболее полное руководство.
2. Bal H., Hujol J. Java for bioinformatics and biomedical applications // Springer, 2007, 353 p.
3. Bergeron B. Bioinformatics Computing // Prentice Hall PTR, 2002, 439 p.
4. Gibas C., Jambeck P. Developing Bioinformatics Computer Skills // O'Reilly, 2001, 446 p.
5. Holzner S. Eclipse. Java developer's guide // O'Reilly, 2004, 334 p.

в) Интернет-ресурсы:

1. Сайт проекта BioJava – www.biojava.org/ .
2. Сайт проекта BioPerl – www.bioperl.org/ .
3. Список открытого программного обеспечения в области биоинформатики - http://en.wikipedia.org/wiki/List_of_Open_Source_Bioinformatics_software
4. Сборник рецептов программирования на разных языках – www.java2s.com/ .

11. Материально-техническое обеспечение дисциплины

- В качестве технического обеспечения лекционного процесса используется ноутбук, мультимедийный проектор, доска.
- Для демонстрации иллюстрационного материала используется программа Microsoft Power Point 2003.
- Проведение контрольных работ и зачёта обеспечивается печатным раздаточным материалом.
- Терминальный класс с установленным программным обеспечением: компилятор JDK (не ниже 1.6 версии), среда разработки Eclipse.

Программа составлена в соответствии с требованиями ФГОС ВПО с учетом рекомендаций ПООП ВПО по направлению «020400 БИОЛОГИЯ», а также в соответствии с Образовательным стандартом высшего профессионального образования, принятым в Федеральном государственном бюджетном образовательном учреждении высшего профессионального образования Новосибирский государственный университет.

Автор: Лашин Сергей Александрович, к.б.н., старший преподаватель кафедры информационной биологии ФЕН.

Рецензент (ы) _____

_____

Программа одобрена на заседании УМК ФЕН НГУ

(Наименование уполномоченного органа вуза (УМК, НМС, Ученый совет))

от _____ года, протокол № _____