Обробка помилок в Java

Класи винятків

Во время работы программы могут возникать ситуации, которые делают невозможным дальнейшее ее выполнение ввиду одной из причин:

- ошибочное программирование (например, задание недопустимого индекса массива или недопустимого значения переменной);
- ввод пользователем неверных данных (например, задание значения делителя равным нулю);
- ошибки компилятора и интерпретатора Java.

Программист должен проанализировать все возможные причины возникновения ошибок в своей программе и организовать обработку возможных ошибок.

Многие ошибки являются типовыми, поэтому их обработка является в большинстве случаев типовой – сигнализация о характере и месте возникновения ошибки и окончания работы программы. Обработку типовых ошибок может выполнять не программист, а средства языка программирования и/или среда выполнения программы. Программисту следует оставить возможность самому организовать нужную ему обработку типовых ошибок, а также обрабатывать нестандартные ошибки.

В объектно-ориентированных языках программирования при возникновении ошибки создается объект определенного класса, содержащий сведения о том, что, где и когда произошло. Этот объект передается на обработку программе, в которой возникла ошибка.

Если программа на языке Java не обрабатывает ошибку, то объект передается обработчику по умолчанию среды выполнения Java. Обработчик выводит в стандартный вывод **System.err** сообщение об ошибке и прекращает выполнение программы.

Объекты-ошибки в Java создаются классами, образующими свою собственную иерархию. Суперклассом для всех классов ошибок в Java является класс **Throwable**, являющийся подклассом класса **Object**. Только экземпляры объектов этого класса и всех других классов, входящих в иерархию этого класса могут обрабатывать ошибки в среде выполнения Java.

В классе **Throwable** определены следующие основные методы:

public String getMessage() – возвращает строку с информацией об ошибке;

public String getLocalizedMessage() – возвращает строку с информацией об ошибке в локализованном виде;

public String toString() – выводит имя класса ошибки, а затем строку с информацией об ошибке;

public void printStackTrace() – отображает на стандартном выводе **System.err** диагностический вывод обработчика ошибок по умолчанию: имя класса ошибки и стек вызовов методов;

public StackTraceElement[] getStackTrace() – возвращает массив объектов класса **StackTraceElement**.

С помощью методов класса **StackTraceElement** можно получить следующие характеристики элемента стека**:**

public String getClassName() – возвращает имя класса, содержащего точку выполнения для данного элемента стека;

public String getFileName() – возвращает имя файла, содержащего точку выполнения для данного элемента стека;

public String getMethodName() – возвращает имя метода, содержащего точку выполнения для данного элемента стека;

public int getLineNumber() – возвращает номер строки для точки выполнения данного элемента стека;

public boolean isNativeMethod() – возвращает true, если метод, содержащий точку выполнения для данного элемента стека, является внешней функцией C/C++ и false – в противном случае;

public String toString() – возвращает строковое представление данного элемента стека.

Использование метода **getStackTrace()** класса **Throwable** и методов класса **StackTraceElement** позволяет формировать пользовательские сообщения об ошибках.
Класс Throwable имеет два подкласса: **Error** (класс ошибок) и **Exception** (класс исключений):
Класс **Error**, образующий, в свою очередь, собственную иерархию классов, обрабатывает исключительные ситуации, связанные с ошибками, которые возникают при аппаратных сбоях или сбоях в работе операционной системы, а также компонент виртуальной машины Java.

Класс **Exception** образуют те исключения, для которых программист на языке Java должен организовать свою собственную обработку.

Класс **RuntimeException** служит родительским классом для тех исключений, которые могут возникать только во время выполнения программы, а не во время ее компиляции. Особенность класса **RuntimeException**: обработку исключений этого класса и производных от него классов в программе описывать необязательно. Если обработка какого-либо исключения времени выполнения в программе отсутствует, оно, как и другие исключения обрабатывается обработчиком по умолчанию среды выполнения Java.

Приклади винятків Java

Исключение	Причина возникновения
ArithmeticException	Математические ошибки, например, деление на ноль (исключение времени выполнения).
ArrayStoreException ArrayIndexOutOfBoundsException	Программа пытается записать в массив неправильный тип данных(исключение времени выполнения). Программа пыталась обратиться к несуществующему индексу массива (исключение времени выполнения).
NegativeArraySizeException	Попытка создать массив с отрицательным размером (исключение времени выполнения).
StringIndexOutOfBoundsException	Программа пыталась обратиться к несуществующей позиции символа в строке (исключение времени выполнения).
IOException FileNotFoundException	Общие исключения ввода/вывода, например, невозможность чтения из файла Обращение к несуществующему файлу (исключение ввода/вывода).
EOFException	Достигнут конец файла (исключение ввода/вывода).
NullPointerException	Ссылка на несуществующий объект – возникает при попытке использовать вместо объекта значение null (например, при доступе к полю или изменении поля объекта null) (исключение времени выполнения).
NumberFormatException	Неправильное преобразование между строками и числами (исключение времени выполнения).
InvalidParameterException	Передача неверного параметра методу (исключение времени выполнения).
AccessControlException	Приложение или апплет пытается выполнить действие, запрещенное режимом защиты (исключение времени выполнения).

Організація обробки винятків

После того как Java создаст объект-исключение, этот объект посылается прикладной программе; такая операция называется **броском (throw)** исключения. Прикладная программа должна перехватить или **поймать (catch)** исключение.
Программист должен организовать собственную обработку исключения, используя блоки **try, catch** и **finally.**

Для обработки исключения фрагмент программы, который может вызвать исключение, помещается в программный блок **try.** Если один из операторов блока бросает исключение, Java игнорирует остальные операторы в блоке **try** и переходит на непосредственно следующий за блоком **try** блок **catch**, в котором программа обрабатывает исключение. Если же все проходит нормально, весь код внутри блока **try** выполняется, а блок **catch** пропускается.

Программный блок **catch** перехватывает исключение, возбужденное системой Java, поэтому после ключевого слова **catch** задается вызов объекта-исключения заданного класса. При необходимости к методам объекта-исключения можно обратиться через этот объект.

В одном блоке **try** может быть сгенерировано несколько исключений. В этом случае каждое из этих исключений обрабатывается своим блоком **catch**, который следуют друг за другом после блока **try**.

Для того, чтобы выполнить некоторый блок кода вне зависимости от того, было исключение или нет, после последнего оператора **catch** можно добавить программный блок **finally**.

```
try {  // Операторы, которые могут вызвать исключения
}
catch (имя-класса-исключения-1 идентификатор-объекта-1) {
      // Операторы обработки исключения-1
}
catch (имя-класса-исключения-2 идентификатор-объекта-2) {
      // Операторы обработки исключения-2
}
…
finally { // Операторы, выполняемые после нормального завершения
      // блока try или по окончанию работы блоков catch
}
```

Между блоками try, catch и finally не должно быть никаких других предложений.

В блоке или блоках **catch** производится обработка исключения, например, вывод диагностического сообщения, завершение программы и другие необходимые действия.

Пример

```
String str1, str2 ="abc";
…
try {
      if (str1.equals(str2))
      System.out.println("Strings str1 and str2 are
equal");
}
catch(NullPointerException npe) {
      System.out.println("String str1 has no value");
      System.exit(2);
}
```

В блоке **catch** можно вообще не выполнять никаких действий. В этом случае блок не будет содержать операторов

Пример

```
int x = 0, x1 = 6, x2 = 0;
…
try {
      x = x1 / x2;
}
catch (ArithmeticException ae) { }
x++;
```

Можно перенести обработку исключения из метода, где оно бросается в метод, вызвавший данный метод. В этом случае в объявлении метода добавляется ключевое слово **throws**, за которым задается список исключений, которые данный метод бросает вызвавшему его методу,

```
void myMethod()
throws ArithmeticException,
NumberFormatException
```

Наличие в объявлении метода выбрасываемого исключения или списка выбрасываемых исключений означает, в вызывающем методе необходимо поместить вызов данного метода в блок **try** и задать и задать блоки **catch** для обработки каждого исключения, перечисленного в списке исключений вызываемого метода.

Пример

```
int index1;
String arg = "1.0r";
…
try {
index1= Integer.parseInt(arg);
}
catch (NumberFormatException e) {
System.out.println("Нечисловое значение " + arg
+
" переменной arg");
System.exit(7);
}
```

В приложении первым вызываемым методом является метод **main(),** поэтому, если в нем не организована обработка исключений, брошенных вызываемыми им методами, эта обработка выполняется обработчиком исключений среды выполнения Java.

Блок **finally** обычно используется в тех случаях, когда необходимо проанализировать, как был выполнен оператор, который мог бросить исключение.

Объекты-исключения передаются вверх от класса классу вверх по иерархии классов обработки ошибок и исключений до пор, пока какой-то из методов очередного класса не обработает их. Если исключение доходит до среды времени выполнения Java, то она обрабатывает исключения предусмотренным по умолчанию образом, обычно генерируя сообщение об ошибке.

В качестве параметра в блоке **catch** можно указать объект любого класса, который лежит в уровне иерархии выше, чем объект класса, бросившего исключение.

Створення власних виключень

Для создания и возбуждения собственных исключений сначала необходимо описать класс для этого исключения, причем этот класс обычно порождается от класса **Exception** или производных от него классов**.**

```
public class NumberRangeException extends RuntimeException {
public NumberRangeException() {
super("Numbers out of range");
}
}
```

Для описания нового исключения необходимо создать класс для данного исключения и в нем определить конструктор класса. Этот конструктор получает параметр, представляющий собой сообщение по умолчанию, которое класс возвращает, если вызывается методы, наследуемые от класса **Throwable.** Внутри конструктора вызывается конструктор суперкласса, т.е. в конце концов, конструктор класса **Throwable.**

В отличие от исключений, определенных в классах библиотеки Java, которые бросаются автоматически при возникновении ошибки в программе, пользовательские исключения необходимо бросать «вручную». Для этого в блоке **try** используется оператор **throw**:

throw *объект-исключение*;

Так же, как и для библиотечных исключений, брошенное исключение обрабатывается в своем блоке **catch**.

```
try {  // Числа не укладываются в заданный диапазон
if ((int1 < 10) || (int1 > 20) ||
int2 < 10) || (int2 > 20)) {
throw new NumberRangeException();
}
answer = int1 + int2;
}
catch (NumberRangeException e) {
System.out.println(e.toString());
}
```

Робота з потоками в Java

Современные операционные системы функционируют в многозадачном режиме. Каждая из решаемых на компьютере задач выполняется в **квазипараллельном** или в **параллельном** режиме. Кроме того, каждая выполняемая задача может быть разбита на отдельные **процессы**, которые также будут выполняться квазипараллельно или параллельно**.**

Квазипараллельность означает, что на одном процессоре имитируется работа нескольких процессоров. Внутри компьютера имеются часы-таймер. При его срабатывании происходит прерывание выполняющегося в данный момент процесса. В период интервала прерывания **диспетчер задач** просматривает таблицу процессов. В этой таблице хранятся указатели на все выполняющиеся в данный момент процессы. Затем определяется наличие процессов, ожидающих выполнения; если таковые отсутствуют, то управление передается процессу, выполнявшемуся ранее.

Выполнение одного процесса

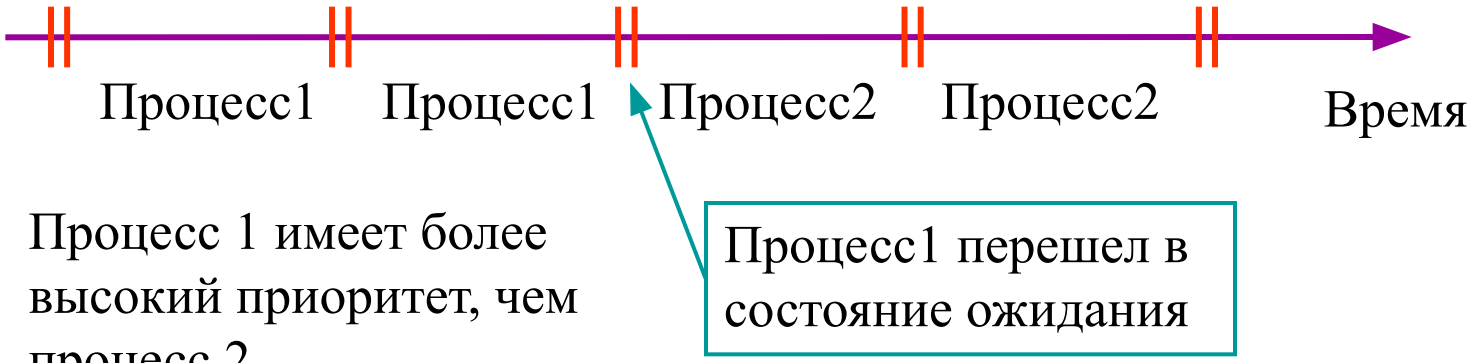
|| - интервал прерывания по таймеру

Когда в компьютере появляется другой процесс, то он может находиться в одном из двух состояний: **в состоянии готовности** или **в состоянии ожидания.** В первом случае процесс готов выполняться, как только освободится процессор компьютера, занятый выполнением других процессов. Во втором случае процесс не готов к выполнению и ожидает выполнения некоторого условия.

Выполнение двух процессов

На практике некоторые процессы являются более важными, чем другие. Для того, чтобы учесть это, для процессов введено понятие **приоритета**. **Чем более важным является поток, тем выше у него приоритет. В частности, диспетчер задач, поскольку он управляет выполнением всех остальных процессов, имеет наивысший приоритет.** Диспетчер задач просматривает очередь готовых к выполнению процессов и допускает к выполнению на процессоре компьютера в первую очередь более приоритетные процессы.

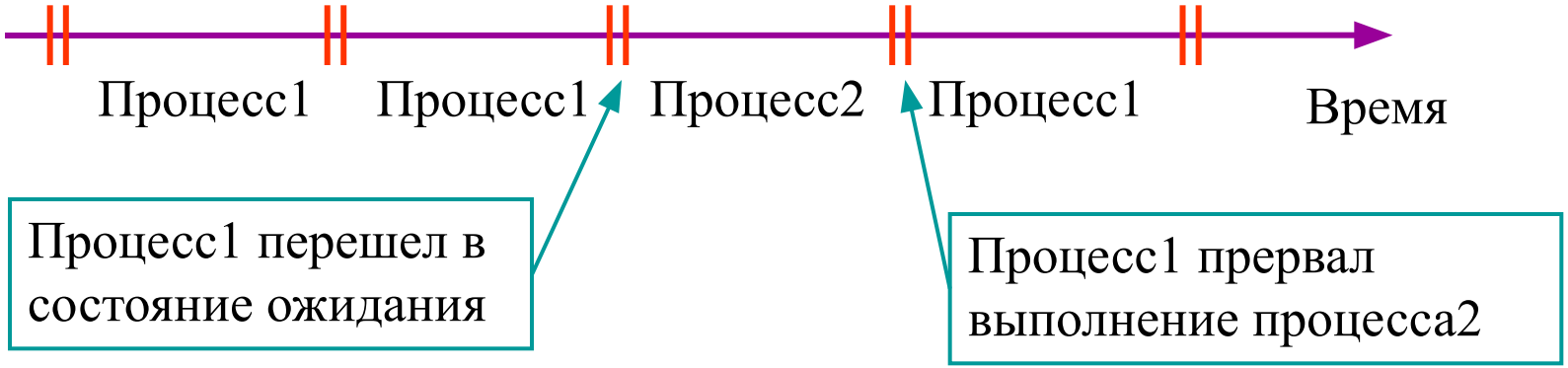
Выполнение двух процессов с учетом приоритета



Выполнение процесса 2 может быть принудительно прекращено при определенных условиях.

Принудительное прекращение процесса называется **прерыванием процесса** (process interrupt).

Выполнение двух процессов с прерываниями



Часто процессы разбиваются по каким-либо признакам на **группы.** Например, в одну группу можно выделить все процессы, относящиеся к определенной задаче. В этом случае диспетчер задач может управлять всей группой так же, как он управляет отдельным потоком.

Операционная система осуществляет защиту каждого процесса от других процессов.

Реалізація потоків в Java

В языке Java потоки обычно используются для того, чтобы **апплеты** могли выполнять действия в то время, как Web-браузер продолжает свою работу, однако потоки можно применить в любой программе при необходимости параллельного выполнения нескольких задач.

Реализация использования потоков в программах на языке Java может выполняться двумя способами:

расширением класса **Thread;**

реализацией интерфейса **Runnable.**

При первом способе класс становится поточным, если он создан как расширение класса **Thread,** который определен в пакете **java.lang:**

```
public class ClassName extends Thread
```

Когда необходимо, чтобы класс являлся расширением некоторого другого класса и в нем необходимо реализовать потоки, для данного класса нужно реализовать интерфейс **Runnable:**

```
public class ClassName extends Applet implements Runnable
```

Интерфейс **Runnable** имеет только один метод

```
public void run()
```

Когда создается класс, реализующий интерфейс **Runnable,** этот класс должен переопределить метод **run().** Именно этот метод выполняет фактическую работу, возложенную на конкретный поток.

Создать поток можно с помощью одного из следующих конструкторов:

```
public Thread()
public Thread(String name)
public Thread(Runnable target)
public Thread(Runnable target, String name)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable target)
public Thread(ThreadGroup group, Runnable target, String name)
```

В первом конструкторе создается поток, который использует самого себя в качестве такого интерфейса **Runnable.**

name – имя, которое присваивается новому потоку;

target – определение целевого объекта, который будет использоваться новым объектом Thread при запуске потоков;

group – предназначен для помещения нового объекта Thread в дерево объектов данного класса. Если опустить данный параметр или присвоить ему значение null, новый объект класса Thread станет членом текущей группы потоков **ThreadGroup**.

Методи для роботи з потоками

public String toString() - возвращает строковое представление потока, включая имя потока, приоритет и имя группы

public final String getName() - позволяет получить имя потока

public final void setName(String name) - установливает имя потока.

public void start() throws IllegalThreadStateException - выполняет запуск потока

Когда запускается программа на языке Java, один поток начинает выполняться немедленно. Этот поток с именем по умолчанию "**main**" обычно называют **главным потоком.** Все остальные потоки, порождаемые с помощью конструктора **Thread**, будут дочерними потоками главного потока. Главный поток является последним потоком, в котором заканчивается выполнение

Для остановки потока рекомендуется останавливаемому потоку присвоить значение **null**

```
Thread myThread;
…
myThread.start(); // Запуск потока
…
myThread = null; // Остановка или завершение потока
```

Потоки в Java могут **прерывать друг друга.** Механизм прерываний реализуется с помощью следующих методов:

public void interrupt() – прерывает данный поток;

public static boolean interrupted() – проверяет, был ли прерван данный поток;

public boolean isInterrupted() – аналогичен предыдущему методу, но не очищает признак прерывания для потока

Обычно программа на Java работает до завершения всех входящих в нее потоков. Иногда встречаются потоки, работающие в фоновом режиме, выполняя вспомогательные действия, которые никогда не заканчиваются. Можно пометить такой поток как **поток-демон (daemon thread),** что говорит JVM о том, что этот поток не надо принимать в расчет при определении, все ли потоки данной программы завершились. Приложение на Java выполняется до тех пор, пока не завершится последний поток, не являющийся демоном.

Потоки, не помеченные как демоны, называются **пользовательскими потоками (user threads).**

Чтобы поток считался демоном, надо воспользоваться методом

```
public final void setDaemon(boolean on) throws
IllegalThreadStateException.
```

Если параметр **on** равен **true**, поток получает статус демона, если **false** – статус пользовательского потока. Статус потока может быть изменен в процессе его выполнения

Пріоритети та групи потоків

Распределение процессорного времени между потоками в Java: когда поток блокируется, то есть приостановлен, переходит в состояние ожидания или должен дождаться какого-то события, Java выбирает другой поток из тех, которые готовы к выполнению. Выбирается поток, имеющий наибольший приоритет. Если таких потоков несколько, выбирается любой из них. Приоритет потока можно установить методом

```
public final void setPriority(int newPriority)
throws IllegalArgumentException.
```

Приоритет потока должен быть числом в диапазоне от **Thread.MIN_PRIORITY** до **Thread.MAX_PRIORITY**. Любое значение вне этих пределов вызывает исключение **IllegalArgumentException**. По умолчанию потоку приписывается приоритет **Thread.NORM_PRIORITY**. Значение приоритета потока можно выяснить с помощью метода

```
public final int getPriority().
```

Класс **ThreadGroup** реализует стратегию обеспечения безопасности, которая позволяет влиять друг на друга только потокам из одной группы. Например, поток может изменить приоритет другого потока из той же группы или перевести его в состояние ожидания. Если бы не было разбиения потоков на группы, один поток мог бы вызвать хаос в среде Java.

Группы потоков организованы в иерархическую структуру, где у каждой группы есть родительская группа. Потоки могут воздействовать на потоки из своей группы и из дочерних групп.

Группу потоков можно создать, просто задав ее имя, с помощью конструктора

```
public ThreadGroup(String groupName)
```

Можно также создать группу потоков, дочернюю по отношению к существующей, используя конструктор

```
public ThreadGroup (ThreadGroup existingGroup,
String groupName) throws NullPointerException.
```

Синхронізація потоків

Основное отличие потоков от процессов состоит в том, что потоки не защищены друг от друга средствами операционной системы. Поэтому любой из потоков может получить доступ и даже внести изменения в данные, которые другой поток считает своими. Решение этой проблемы состоит в **синхронизации потоков.**

Синхронизация потоков состоит в гарантии в каждый момент времени предоставления доступа к данным только к одному потоку. Для этого используется следующий оператор:

synchronized (*выражение*)
оператор

Взятое в скобки ***выражение*** должно указывать на блокируемые данные – обычно оно является ссылкой на объект. Чаще всего при блокировке объекта необходимо выполнить сразу несколько операторов, так что ***оператор***, как правило, представляет собой блок.

Если оказывается, что критический участок распространяется на весь метод, а разделяемым ресурсом является весь объект в целом, то можно просто указать модификатор **synchronized** в объявлении метода:

```
synchronized void myMethod() {
        …
}
```

Более гибкий и эффективный способ координации выполнения потоков обеспечивают методы **wait()** и **notify()** класса **Object.**

Метод **wait()** переводит поток в состояние ожидания выполнения определенного условия.

Метод **notify()** переводит в активное состояние один из потоков, установленных в состояние ожидания с помощью метода **wait().**

Інтерфейси в Java

Если класс потомок может иметь только один родительский класс, то такое наследование называется **одиночным наследованием.** Однако иногда необходимо, чтобы данный класс имел несколько суперклассов

Частично концепцию множественного наследования в Java можно реализовать с помощью интерфейсов.

Интерфейсы – это абстрактные классы, которые являются полностью нереализуемыми. Это означает, что в интерфейсе допустимы только объявления методов (без описания тела методов). Кроме того, данные интерфейса содержат только переменные с модификаторами **static final**, т. е. являются статическими константами.

Для классов могут быть реализованы несколько интерфейсов, что является альтернативой множественному наследованию. **Главное различие между наследованием интерфейсов и множественным наследованием заключается в том, что интерфейсы позволяют наследовать только описания методов, а не их реализацию.** Поэтому, если класс реализует множественные интерфейсы, этот **класс должен обеспечить реализацию всех методов, определенных в интерфейсе**. Хотя это более ограниченное свойство, чем множественное наследование, оно во многих случаях является очень полезным.

Между интерфейсами и абстрактными классами существует два важных отличия:

- интерфейсы предоставляют некоторую разновидность множественного наследования, поскольку класс может реализовать несколько интерфейсов.
- абстрактный класс может содержать частичную реализацию, защищенные компоненты, статические методы и т. д.

Оголошення інтерфейсів

```
модификаторы interface идентификатор-интерфейса
{
    тело-интерфейса
}
```

Идентификатор-интерфейса определяет имя интерфейса, а ***тело-интерфейса*** описывает абстрактные методы и переменные, составляющие интерфейс. Поскольку в интерфейсе определяются только абстрактные методы, поэтому сам интерфейс тоже является абстрактным и должен иметь модификатор **abstract.**

Однако задание модификатора **abstract** для интерфейса является излишним. Из модификаторов доступа для интерфейсов можно использовать либо модификатор доступа по умолчанию, либо модификатор **public.**

Модификаторы доступа **protected** и **private** можно использовать только для рассматриваемых далее внутренних интерфейсов.

Независимо от того, какие модификаторы используются при объявлении переменных, все переменные в интерфейсах могут быть только **public**, **final** и **static**. Все переменные в интерфейсе должны быть обязательно инициализированы с помощью констант, выражений или вызова статических методов.

Основная задача интерфейсов — объявлять абстрактные методы, которые будут описываться в других классах, поэтому синтаксис объявления метода в интерфейсе очень похож на объявление метода класса, но в отличие от него, у методов интерфейсов отсутствуют тела:

возвращаемое-значение имя-метода (***параметры***)
throws ***исключения***;

Так как предполагается, что все методы интерфейса являются абстрактными и доступными всем классам, нет необходимости указывать для них модификаторы **abstract** и **public.**

Интерфейсы, так же как и классы, компилируются в отдельные файлы с именами, совпадающими с именами интерфейсов и расширением **.class**.

Пример объявления интерфейса:

```
interface TestNumber {
      NULL = 0;
      ONE = 1;
      boolean testNull(int number);
      boolean testOne(int number);
}
```

Спадкування інтерфейсів

Интерфейсы так же, как и классы, могут расширяться с помощью ключевого слова **extends**, однако, в отличие от классов, допускается расширение интерфейсом сразу нескольких других интерфейсов. Например, для интерфейсов блоков прослушивания событий мыши, часть интерфейсов имеют следующие объявления:

```
public interface EventListener {
    // Тело интерфейса
}
public interface MouseListener extends EventListener {
    // Тело интерфейса
}
public interface MouseMotionListener extends EventListener {
    // Тело интерфейса
}
public interface MouseInputListener extends MouseListener,
MouseMotionListener {
// Тело интерфейса
}
```

Если какой-либо интерфейс объявляет переменную с именем, которая уже объявлена в одном из его интерфейсов-предков, то при использовании этого интерфейса значение переменной в интерфейсах-предках будет недоступным. Такая ситуация называется **сокрытием** (hiding) **переменной.**

Интерфейсы не имеют единого корневого интерфейса, наподобие класса **Object.** Поэтому интерфейсы могут образовывать множество связанных или не связанных между собой древовидных структур.

В связи с такой структурой наследования в интерфейсах могут возникнуть проблемы **неоднозначного и множественного наследования** (ambiguous and multiple inheritance):

```
interface I {
        i = 1; j = 0;
}
```

```
interface J extends I {
        j = 1;
}
```

```
interface K extends I, J {
        k = 2;
}
```

Реалізація інтерфейсів в класах

Объявленные в интерфейсе методы нельзя использовать до тех пор, пока некоторый класс или классы не реализуют данный интерфейс и не переопределят эти методы.

Для того, чтобы указать, что какой-либо класс реализует интерфейс или интерфейсы, класс объявляется следующим образом:

```
class идентификатор-класса implements список-интерфейсов
{
    тело-класса
}
```

Идентификатор определяет имя нового класса,

интерфейс – определяет имя реализуемого интерфейса (может быть задано несколько имен интерфейсов, разделенных запятыми),

тело-класса – определяет тело нового класса.

При реализации какого-либо интерфейса необходимо переопределить все методы, объявленные в нем. Это приходится делать даже в том случае, если некоторые методы в данном классе не используется. Если этого не сделать, класс будет считаться компилятором Java абстрактным классом.

Чтобы избежать этой ситуации, неиспользуемые методы обычно описывают в классе с пустым телом метода.

Пример использования интерфейса:

```
interface IntParmIO {
        void parmInput(String name, int value);
        void parmOutput(String name, int value);
}

Class MyClass implements IntParmIO {
        // …
        void parmOutput(String name, int value) {
        System.out.println(name + value);
        }
        void parmInput(String name, int value) {}
}
```

Змінні интерфейсного типу і їх перетворення

Можно определять **переменные интерфейсного типа** – ссылочные переменные, использующие в качестве типа переменной идентификатор интерфейса. В такой переменной можно сохранять экземпляр того класса, который реализует заданный интерфейс. При вызове метода для такой переменной будет вызываться та версия объявленного в интерфейсе метода, которая определена в данном классе.

Пример

```
Class InterfaceVar {
    // …
    IntParmIO varOut = new MyClass();
    varOut.parmOutput("x=", 15);
}
```

Переменная **varOut** объявлена как переменная интерфейсного типа **IntParmIO,** в которой хранится экземпляр класса **MyClass,** реализующего интерфейс **IntParmIO**

Так же, как и для классов, для переменных интерфейсного типа можно выполнять преобразования одного типа в другой.

Расширяющими преобразованиями для переменных интерфейсного типа являются следующие преобразования:

- из любой переменной класса в любую переменную заданного интерфейсного типа, при условии, что класс реализует заданный интерфейс;
- из типа **null** к любому интерфейсному типу;
- из любого интерфейсного типа к заданному интерфейсному типу при условии, что интерфейсный тип является потомком заданного интерфейсного типа;
- из любого интерфейсного тип к типу **Object.**

Расширяющие преобразования выполняются автоматически и не вызывают ошибок во время выполнения программы.

Сужающими преобразованиями для переменных интерфейсного типа являются следующие преобразования:

- из любой переменной класса в любую переменную заданного интерфейсного типа, при условии, что класс не имеет модификатор **final** и не реализует заданный интерфейс;
- из типа **Object** к любому интерфейсному типу;
- из любого интерфейсного типа к заданной переменной класса, который не имеет модификатора **final;**
- из любого интерфейсного типа к заданной переменной класса, который имеет модификатор **final** и реализует данный интерфейс;
- из любого интерфейсного типа к заданному интерфейсному типу при условии, что интерфейсный тип не является потомком заданного интерфейсного типа и оба интерфейсных типа не содержат метода с одним и тем же именем, но разными возвращаемыми значениями.

Выполнение сужающих преобразований требует использования оператора приведения типа

(идентификатор-типа)переменная