

# JavaFX

Графический интерфейс



# Графический интерфейс

В Java применяются разные графические библиотеки:

- AWT (Abstract Window Toolkit)
- Swing
- SWT
- JavaFX

# AWT

AWT (Abstract Window Toolkit) — первая библиотека для работы с графикой, появилась еще в версии 1.0, и оказалась не совсем удачной.

Хоть она и была кроссплатформенной, AWT зависела от графической подсистемы.

Предполагалось, что компоненты будут выглядеть одинаково хорошо на разных платформах. На самом деле получилось, что все выглядит одинаково плохо и хуже того, по-разному.

# AWT

Разработчики оставили только те компоненты и только те функции, которые поддерживались во всех операционных системах.

В итоге получилась библиотека с очень урезанными возможностями. Компонентов было мало, при этом они поддерживали не все возможные функции.

Но зато эта библиотека работала для того времени довольно быстро.

# Swing

С версии 1.1 появилась библиотека Swing, сначала как отдельная библиотека, потом как часть Java.

Основная особенность Swing в том, что почти все компоненты написаны на Java. Поэтому в принципе они должны выглядеть одинаково везде.

Однако из-за того, что компоненты нужно отрисовывать, все стало работать медленнее, чем было в AWT.

Потом была проведена работа над ускорением и сейчас правильно настроенный Swing работает достаточно быстро.

# Swing

Полезной особенностью Swing является изменяемый вид компонентов, который можно менять на ходу.

Не сказать, что Swing является самой удачной библиотекой, но изучить на его примере основные принципы вполне можно.

# SWT

Библиотека SWT появилась как часть Eclipse, ее разработку поддержала компания IBM.

SWT есть не для всех платформ, но при ее создании разработчики удачно соединили лучшее из AWT и Swing.

В SWT компоненты и их функции, которые поддерживаются графической подсистемой, как и в AWT работают через адаптеры, а недостающие функции, как в Swing, дописаны на Java.

# JavaFX

JavaFX изначально позиционировалась как новая графическая библиотека, с поддержкой анимации, визуальных эффектов, возможностью задания интерфейса с помощью XML, поддержки стилей.

Сначала JavaFX была отдельной библиотекой, потом стала частью Java, сейчас снова выделена в отдельный проект под названием OpenJFX.



# JavaFX

- Новая библиотека для разработки RIA (Rich Internet Applications)
- Поддержка XML для создания интерфейса
- Поддержка стилей CSS
- Поддержка 2D- и 3D-графики
- Легковесные компоненты
- Интеграция с библиотекой Swing

# Создание графических приложений

- Создание основного окна
- Создание остальных элементов интерфейса
- Размещение элементов интерфейса в иерархии контейнеров
- Обеспечение реакции элементов на события
- Все заработало!

# JavaFX

Основной класс для приложений JavaFX - Application. Это класс-предок всех приложений.

Для написания своего приложения просто наследуемся от Application. У этого класса есть четыре основных метода:

1. `init()` - для инициализации, обычно туда помещается код, который задает начальные значения.
2. `stop()` освобождает ресурсы, вызывается при закрытии приложения.
3. абстрактный метод `start()`. В нем должен быть весь код приложения. Методу `start` передается объект класса `Stage`, создавать этот объект не надо.
4. `launch()` запускает приложение. Этому методу можно передать аргументы

# JavaFX

- Stage — основная платформа
- Контейнер верхнего уровня
- Предоставляется системой при запуске приложения
- Обеспечивает связь с графической подсистемой ОС

Основные методы:

- setTitle(String)
- setScene(Scene)
- show()

# JavaFX

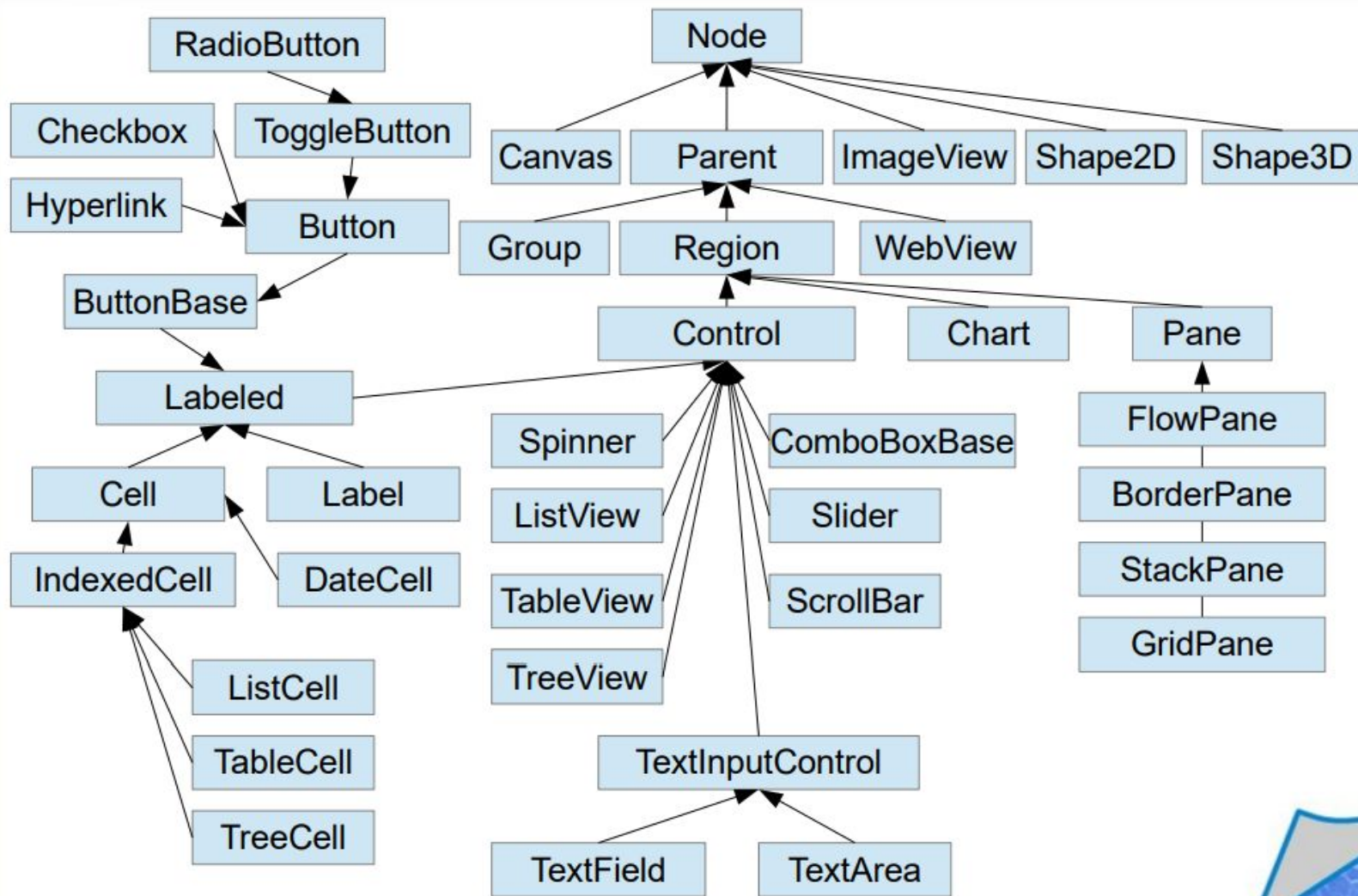
- Scene — контейнер для элементов сцены
- Должен быть хотя бы один объект класса Scene
- Элементы сцены — узлы (Node)
- Узлы образуют граф (scene graph)
- Граф включает не только контейнеры и компоненты, но также графические примитивы (текст и графические примитивы)
- Узел с дочерними узлами — Parent (extends Node)
- Корневой узел (root node) — узел без родительского узла
- `Scene sc = new Scene(root node, 300, 150);`

# JavaFX

Класс Node - узел. Его свойства (properties):

- String id
- Parent (только один)
- Scene
- Стиль (styleClass, style)
- Видимость, активность, прозрачность
- Размеры (min, max, preferred)
- Границы (boundsInLocal, boundsInParent, layoutBounds)
- Трансформации (сдвиг, вращение, масштаб, наклон)
- Эффекты
- События (mouse, key, drag, touch, rotate, scroll, swipe, zoom)

# Основные компоненты JavaFX



# Многопоточные приложения в Java





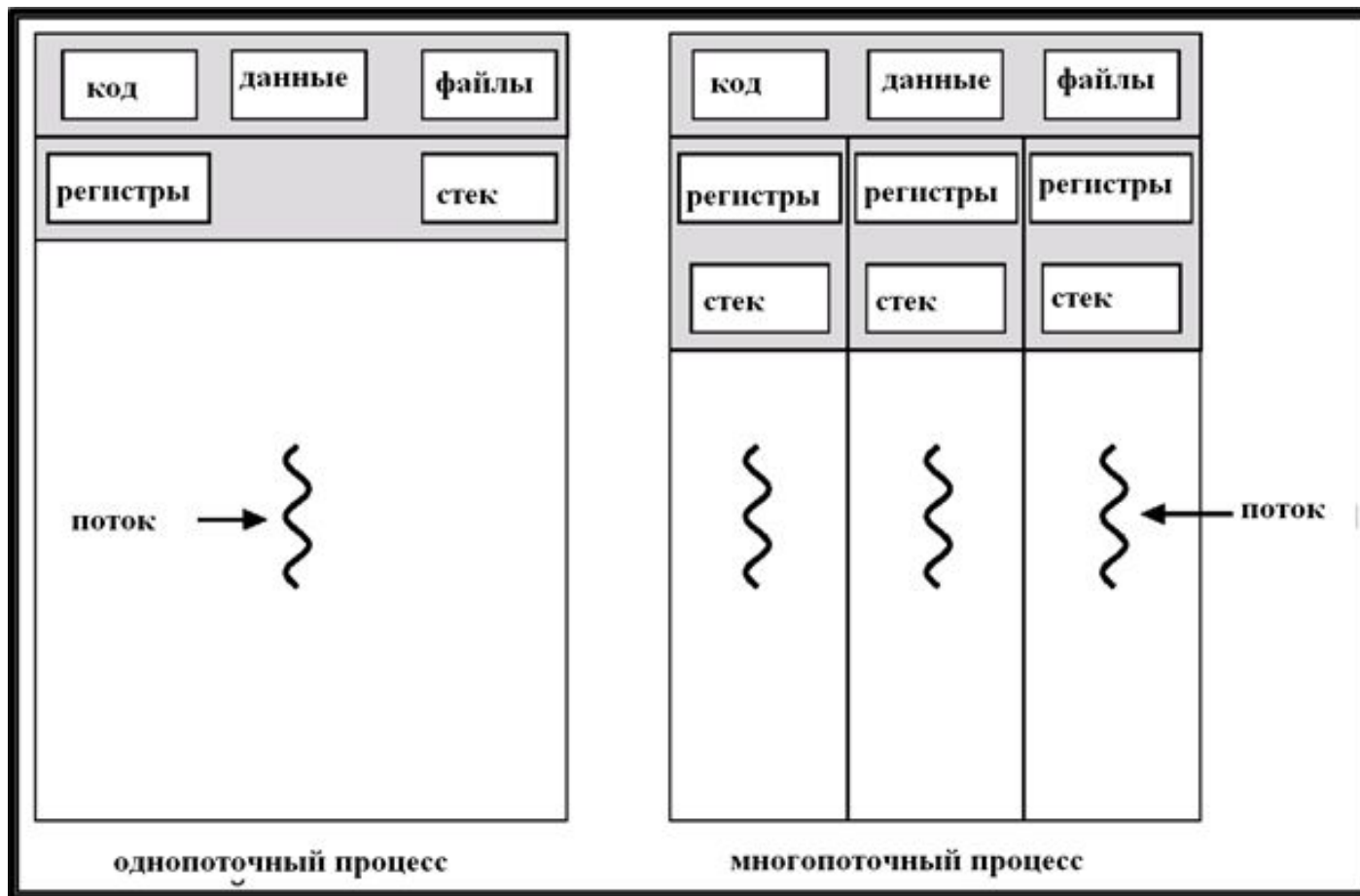
# Многопоточность

- **Многопоточность** операционной системы – возможность одновременного выполнения более чем одной программы.
- Число одновременно выполняющихся процессов не ограничено количеством процессоров.
- Многопоточные программы расширяют идею **многозадачности**. Индивидуальные приложения могут выполнять множество задач в одно и то же время.
- Каждая задача называется **поток**ом – **thread**.

# МНОГОПОТОЧНОСТЬ

- Процесс — это экземпляр программы, который запускается независимо от остальных, у него есть собственное адресное пространство.
- Поток – это одна из веток процесса. Все потоки разделяют адресное пространство породившего их процесса и имеют доступ к одним данным.
- Один из потоков – «**главный**» начинает выполняться первым при запуске Java-программы. Главный поток создается автоматически с именем `main` и приоритетом 5 по умолчанию.
- От него порождаются дочерние потоки.

# МНОГОПОТОЧНОСТЬ



# Многопоточность

- Существенная *разница* между *многими процессами* и *многими потоками* заключается в следующем:
  - каждый процесс имеет собственный набор переменных, потоки могут разделять одни и те же данные.
- Потоки являются более «легковесными», чем процессы.
- Пример многопоточных приложений – браузер, web-сервер, программы с графическим пользовательским интерфейсом.

# Процедура запуска задачи в отдельном потоке

- Класс *Thread* предназначен для создания нового потока.
- Он определяет следующие основные конструкторы :
- `Thread()`
- `Thread(Runnable object)`
- `Thread(Runnable object, String name)`
- `Thread(String name)`
- где *name* - имя, присваиваемое потоку, *object* - экземпляр объекта *Runnable* .
- Если имя не присвоено, система сгенерирует уникальное имя в виде *Thread-N*, где *N* - целое число. Для создания потока можно использовать также интерфейс *Runnable*
- ```
public interface Runnable {  
    public abstract void run();  
}
```

# Прерывание потоков

- Поток прерывается, когда его метод `run()` возвращает управление, выполнив оператор `return`, после последнего оператора или в случае возникновения исключения.
- Для принудительного прерывания вызовом метода ***interrupt()*** выставляется статус прерывания (`interrupted status`). Каждый поток периодически проверяет этот статус.
- Для проверки установки статуса прерывания применяется статический метод ***isInterrupted()***:

```
while (!Thread.currentThread().isInterrupted() && есть еще работа){  
    выполнять работу}
```

# Прерывание потоков

- Если поток заблокирован, то он не может проверить статус прерывания.
- Когда метод `interrupt()` вызывается для потока, который заблокирован таким вызовом как `sleep()` или `wait()`, то блокирующий вызов прерывается исключением `InterruptedException`.

```
public void run(){
    try{...
    while(еще есть работа){
        Выполнять работу;
        Thread.sleep(delay);}
    }
catch(InterruptedException e){ поток прерван во время ожидания}
    finally{при необходимости что-то сделать}
}
```

# Прерывание потоков

- Не игнорируйте `InterruptedException`! Необходимо поступить одним из двух способов:
- Выставьте флаг прерывания  
`catch(InterruptedException e)`  
`{Thread().currentThread().interrupt();}`
- Или предупредите метод о возможном исключении через `throws`  
`InterruptedException`



# Состояние потока

Существует 6 состояний потока.

- **Новый**. Как только поток был создан операцией `new`, он находится в состоянии “НОВЫЙ”.
- **Работоспособный**. Как только вызывается метод `start`, поток оказывается в работоспособном состоянии. Работоспособный поток может в данный момент выполняться, а может и нет (зависит от ОС, поэтому он не называется работающим).

# Состояние потока

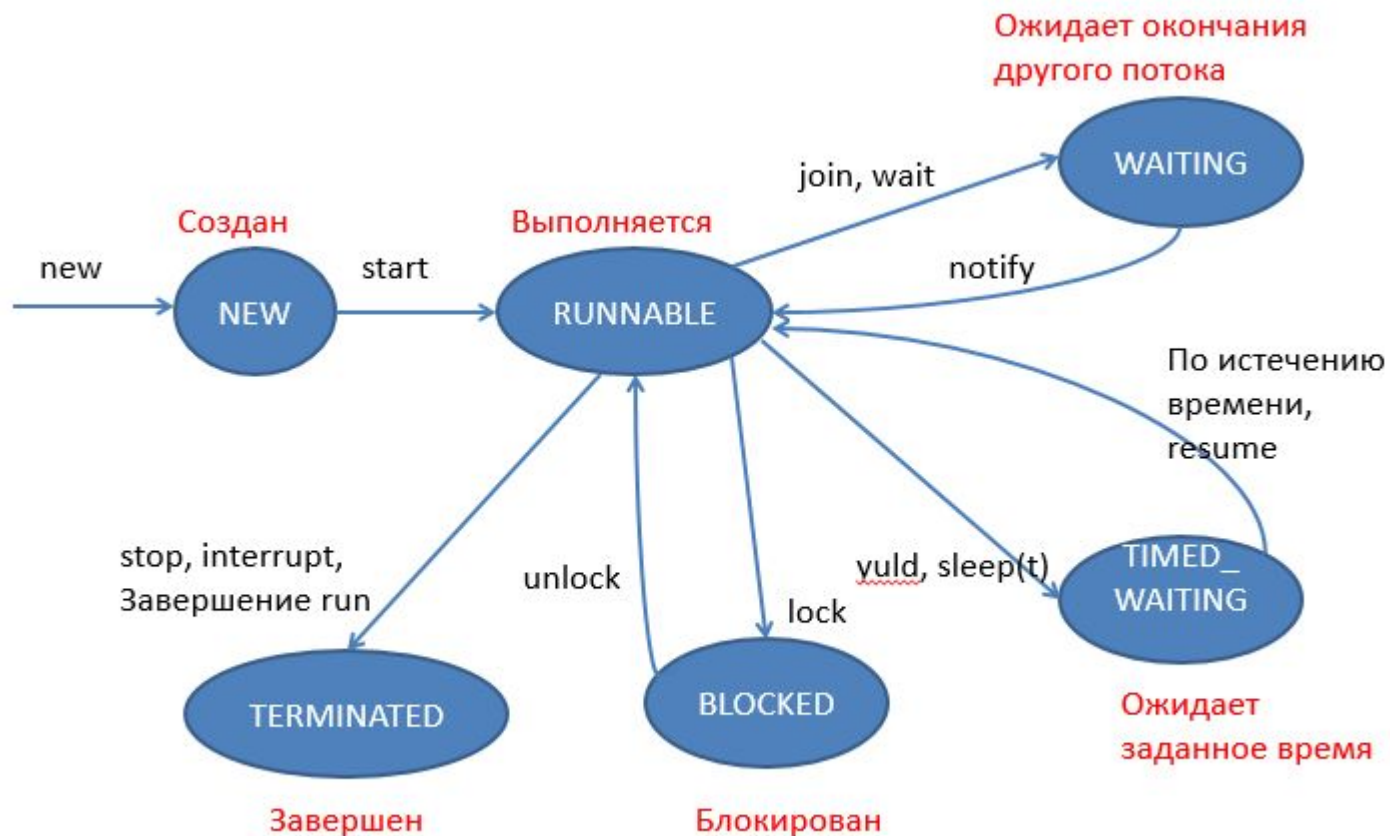
Когда поток заблокирован или находится в состоянии ожидания, он временно не активен. Он не выполняет никакого кода, потребляет минимум ресурсов.

- **Блокированный.** Когда поток пытается получить внутренний объект блокировки, он становится заблокированным. Поток разблокируется, когда все остальные потоки освобождают объект блокировки и планировщик потоков позволяет захватить его.
- **Ожидание.** Когда поток ожидает другого потока для уведомления планировщика о наступлении некоторого условия, он входит в состояние ожидания. Разница между заблокированным и ожидающим на практике не велика.

# Состояние потока

- **Временное ожидание.** Поток входит в это состояние вызовом некоторых методов, имеющих параметр таймаута.
- **Завершенный поток.** Поток завершается по одной из следующих причин:
  - при нормальном выходе из `run()`
  - неперехваченное исключение прерывает метод `run()`
- Можно уничтожить поток, вызвав метод `stop()`, который генерирует ошибку `ThreadDeath`. Но данный метод не рекомендуется к использованию.

# Состояние потока



# Свойства потока. Приоритет

- Поток наследует приоритет потока, который его создал.
- Метод ***setPriority()*** устанавливает приоритет между MIN\_PRIORITY (равен 1) и MAX\_PRIORITY (равен 10). NORM\_PRIORITY равен 5.
- Когда планировщик потоков выбирает поток для выполнения, он предпочитает потоки с более высоким приоритетом (вытесняющее планирование).
- Приоритеты потоков в значительной мере ***зависимы от системы***. В Windows 7 приоритетов, в Sun – приоритеты игнорируются!

# Потоки-демоны

- Обычный поток можно превратить в поток-демона вызовом метода **setDaemon(true)**.
- Демон – поток, основное назначение которого - служить другим.
- Примеры – поток таймера, отсчитывающего тики, очистка кэша...

# Синхронизация

- В практических многопоточных приложениях часто необходимо двум или более потокам разделить доступ к одним и тем же данным.
- В такой ситуации возникает ошибка – *состояние гонки*.
- Чтобы избежать этого, необходимо синхронизировать доступ!