



A História das Versões LTS do Java

Desde seu lançamento em 1995, o Java tem evoluído constantemente para atender às demandas tecnológicas em constante mudança, mantendo-se uma das linguagens mais populares e robustas no desenvolvimento de software empresarial.

Dentro desse ciclo evolutivo, as versões *Long Term Support* (LTS) têm uma relevância especial. Estas versões são lançamentos estáveis que recebem suporte estendido da Oracle e de outras distribuições, proporcionando uma plataforma confiável para empresas que precisam de um ciclo de atualização mais longo e previsível.

As versões LTS se destacam não apenas por sua estabilidade, mas também por introduzirem mudanças estruturais e conceituais que moldam o futuro da linguagem.

Neste texto, vamos explorar a história das principais versões LTS do Java, com detalhes sobre suas inovações, fundamentos técnicos e impacto na comunidade de desenvolvimento. Vamos entender por que cada versão representou uma evolução significativa para o ecossistema Java e como essas mudanças moldaram o desenvolvimento de software ao longo do tempo.

1. Java 8 (Lançado em Março de 2014)

Antes do Java 8, o Java enfrentava críticas por ser uma linguagem verbosa e limitada em comparação com outras linguagens de programação emergentes, como Scala e Kotlin. Essas linguagens já haviam adotado um estilo de programação funcional, algo que faltava ao Java.

A Sun Microsystems, empresa responsável pelo Java até sua aquisição pela Oracle, havia lançado várias versões sem trazer mudanças significativas no modelo de programação.

A versão 8 trouxe uma revolução ao incorporar conceitos de programação funcional.

Principais Recursos:

- **Lambdas:** Introdução das expressões lambda permitiu que os desenvolvedores escrevessem código mais conciso e expressivo. Isso foi crucial para operações sobre coleções e programação paralela.



- **Streams:** Uma nova API que facilitou o processamento de grandes conjuntos de dados de maneira eficiente e funcional. Através de *Streams*, os desenvolvedores podem realizar operações como filtragem, mapeamento e redução de forma declarativa.
- **API de Data e Hora (java.time):** A antiga API de datas, baseada em `java.util.Date` e `Calendar`, era notoriamente difícil de usar. O Java 8 trouxe uma nova API baseada no padrão ISO, semelhante à biblioteca Joda-Time.
- **Optional:** Introdução da classe `Optional` para ajudar a evitar problemas comuns relacionados a valores nulos e reduzir o número de exceções `NullPointerException`.

Exemplo de Código (Streams e Lambdas):

```
import java.util.Arrays;  
import java.util.List;
```

```
public class Java8Example {  
    public static void main(String[] args) {  
        List<String> names = Arrays.asList("Ana", "Carlos", "Maria", "José");  
  
        // Usando Streams e Lambdas para filtrar e imprimir nomes que começam com "M"  
        names.stream()  
            .filter(name -> name.startsWith("M"))  
            .forEach(System.out::println); // Output: Maria  
    }  
}
```

Impacto e Curiosidade:

- O Java 8 foi amplamente adotado devido à melhoria na produtividade dos desenvolvedores, tornando o código mais legível e menos verboso. A introdução de *Lambdas* foi uma das mudanças mais aguardadas e revolucionárias da linguagem.
- O sucesso dessa versão foi tão grande que, mesmo após o lançamento de novas versões, o Java 8 continuou a ser amplamente utilizado no setor corporativo por muitos anos, sendo a versão preferida até a chegada do Java 11.

2. Java 11 (Lançado em Setembro de 2018)

Após o lançamento do Java 8, a Oracle adotou uma nova estratégia de lançamento com versões mais frequentes (semestrais). No entanto, apenas algumas dessas versões seriam consideradas LTS, garantindo suporte a longo prazo. O Java 9 e 10 trouxeram inovações experimentais, mas foi no Java 11 que muitas dessas inovações se consolidaram em um formato estável.

Principais Recursos:

- **Execução de Arquivos Java sem Compilação:** No Java 11, você pode executar diretamente arquivos `.java` sem compilar explicitamente. Isso simplifica o desenvolvimento de scripts curtos e testes rápidos.
- **HttpClient:** Introdução de uma nova API HTTP, `HttpClient`, que simplificou drasticamente o consumo de serviços web em comparação com o antigo `URLConnection`.



- **Remoção de Ferramentas Obsoletas:** Recursos como Applets, Java Web Start e APIs antigos foram removidos, marcando o fim de uma era para certos tipos de aplicativos baseados em Java.
- **Novos Métodos de String:** Como `isBlank()`, `lines()`, `repeat()` e outros, facilitando o trabalho com strings.

Exemplo de Código (HttpClient):

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class Java11HttpExample {
    public static void main(String[] args) throws Exception {
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("https://api.github.com"))
            .build();

        HttpResponse<String> response = client.send(request,
            HttpResponse.BodyHandlers.ofString());
        System.out.println(response.body());
    }
}
```

Impacto e Curiosidade:

- Java 11 foi visto como o sucessor natural do Java 8, trazendo muitas melhorias e mudanças significativas, embora não tão revolucionárias quanto as introduzidas no Java 8. Ainda assim, foi a primeira versão LTS após o Java 8 e adotada de forma lenta por empresas devido à robustez do Java 8 e a necessidade de atualizações mais complexas.
- O fim do suporte oficial para tecnologias antigas, como Applets, marcou uma mudança no foco do Java em direção ao desenvolvimento mais moderno, deixando para trás as tecnologias voltadas para navegadores.

3. Java 17 (Lançado em Setembro de 2021)

Com Java 17, a Oracle reafirmou seu compromisso com a estabilidade e inovação, ao mesmo tempo em que consolidou recursos que vinham sendo testados em versões intermediárias.

Essa versão solidificou o ciclo de lançamentos regulares, com melhorias incrementais, mas consistentes.

O Java 17 trouxe avanços significativos em termos de desempenho, simplicidade na linguagem e segurança.



Principais Recursos:

- **Records:** Uma nova maneira de declarar classes de dados imutáveis de forma concisa. Isso eliminou a necessidade de escrever manualmente getters, setters, métodos equals(), hashCode() e toString().
- **Sealed Classes:** Introdução das *sealed classes*, que permitem restringir quais classes podem herdar de uma determinada classe, melhorando o controle da hierarquia de classes.
- **Pattern Matching para instanceof:** Simplificação do uso de instanceof, permitindo que o desenvolvedor realize o *casting* diretamente após a verificação do tipo.
- **Garbage Collection:** Melhorias no desempenho e na coleta de lixo, especialmente com o ZGC (Z Garbage Collector), que foi otimizado para cenários de baixa latência.

Exemplo de Código (Records):

```
public record Person(String name, int age) {}

public class Java17Example {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30);
        System.out.println(person.name() + " is " + person.age() + " years old.");
    }
}
```

Impacto e Curiosidade:

- Java 17 é visto como um marco na modernização do Java. Recursos como *records* e *sealed classes* são considerados essenciais para desenvolvedores que trabalham com grandes sistemas corporativos e precisam de soluções concisas e seguras.
- Além disso, Java 17 também consolidou o padrão de *Pattern Matching*, algo que vinha sendo trabalhado desde versões anteriores e que simplifica a verificação de tipos e o manuseio de objetos.

Evolução e Tendências

A evolução das versões LTS do Java reflete uma busca contínua pela simplificação do desenvolvimento, pela melhoria da eficiência em tempo de execução e pela adoção de paradigmas modernos de programação.

Cada versão introduz mudanças que tornam o Java mais relevante para os desafios de desenvolvimento atuais.

Principais Motivações para a Evolução:

- **Eficiência e Produtividade:** Funções como *lambdas*, *records* e *streams* reduzem a verbosidade e promovem uma codificação mais rápida e eficiente.
- **Segurança e Estabilidade:** Cada nova versão LTS traz avanços na segurança, como a remoção de APIs antigas e vulneráveis e melhorias nos mecanismos de coleta de lixo (Garbage Collection).



- **Integração com Tecnologias Modernas:** O Java tem sido aprimorado para melhor lidar com arquiteturas de microserviços, APIs HTTP e computação em nuvem, facilitando a vida dos desenvolvedores que trabalham em ambientes modernos e distribuídos.

Conclusão

As versões LTS do Java não são apenas atualizações tecnológicas, mas marcos significativos na evolução de uma das linguagens de programação mais usadas no mundo.

Desde a introdução das expressões lambda em Java 8 até as mudanças estruturais em Java 17, a linguagem se modernizou, permanecendo relevante em um cenário de desenvolvimento em constante mudança.

Referências:

- Oracle. *Java SE Development Kit 8 Documentation*. Disponível em: <https://docs.oracle.com/javase/8/docs/>
- Oracle. *Java SE Development Kit 11 Documentation*. Disponível em: <https://docs.oracle.com/en/java/javase/11/>
- Oracle. *Java SE Development Kit 17 Documentation*. Disponível em: <https://docs.oracle.com/en/java/javase/17/>

EducaCiência FastCode para a comunidade