1)Inversion of Control: Objelerin veya programların kontrolünü, yazılım kütüphanesine veya konteyner yapısına bırakan yazılım prensibidir. Bu bağlamdaki kontrolden kasıt, program akışı, birbirine bağımlı objelerin yaratılması, bunların birbirine bağlanması vb. olarak örneklendirilebilir.

Dependency Injection: IOC prensibini uygulayan tasarım modeli.

2)Spring Bean yapısının kapsam alanı, bu yapının yaşam döngüsünü ve kullanılan alan içindeki görünürlüğünü tanımlar. Spring kütüphanesi 6 farklı kapsama alanı içerir:

- Singleton->Spring konteyner içinde sadece bir Spring Bean yapısı yaratmak için kullanılır. Ayrıca varsayılan kapsam alanıdır.
- Prototype->Spring konteynera atılan her istek için obje yaratılmasında kullanılır
- Request->Her http isteği için obje yaratılmasında kullanılır.
- Session->Her http session için obje yaratılmasında kullanılır.
- Application->Singleton mantığında ancak ServletContex yaşam döngüsü süresince kullanılır.
- Websocket-> Singleton mantığında ancak WebSocket yaşam döngüsü süresince kullanılır.

3) @EnableAutoConfiguration, @Configuration ve @ComponentScan yapılarını birleştiren bir anotasyondur. Sırasıyla Spring Boot otomatik konfigürasyon mekanizmasını aktif etme, yani bu bağlamda Spring Bean yapılarını otomatik oluşturma, bağlı olduğu paket ve onun alt paketlerinde bulunan anotasyonlu yapıları tanımlamak ve yapılandırmak, içerikte bulunan ekstra Bean yapılarını kaydetme veya yapılandırma sınıflarını içe aktarma işlemini yapar.

4)Spring Boot daha az kodla ve kolay bir şekilde Web uygulamaları geliştirmeye olanak verir. Kendi başına çalışan uygulamalar yazmaya yardımcı olurken, bu uygulamaların daha az yapılandırılmaya ihtiyaç duymasını sağlar. Ayrıca gömülü veri tabanı ve sunucular gibi özellikleri sağlaması, uygulama geliştirme açısından daha avantajlıdır.

5)Singleton bir tasarım modelidir. İlgili sınıftan yalnızca bir obje üretilerek, her yerde aynı objeyi kullanmak istenildiğinde tercih edilir. Veri tabanı, dosya vb. gibi paylaşılan kaynaklar için kullanılır.

6)@Controller ve @ResponseBody yapılarını birleştiren bir anotasyondur. Uygulandığı sınıfın, Spring tarafından kontrol edilen ve http isteklerini işleyen bir yapı olduğunu gösterir. Gelen istekleri ilgili yapılara gönderme işlemini yapar ve ilgili yapılardan dönen verileri JSON ya da XML türüne dönüştürür.

7)Spring kütüphanesinin ana özelliği bağımlılık enjeksiyonu yapmasıyken, Spring Boot aracının ana özelliği otomatik yapılandırmadır.

8)Versiyon kontrol sistemi, iş birliğini arttırarak birlikte çalışmayı kolaylaştırır. Proje üzerinde çalışan kişilerin asenkron bir şekilde çalışmasına olanak sağlayarak, kod üzerinde yapılan değişikliklerin takibini kolaylaştırarak ortaya çıkabilecek sorunların önüne geçmede yardımcı olur.

9)SOLID Prensipleri, nesneye yönelik programlama yapılırken kullanılan tasarım kavramlarıdır. Ortaya konulan işin daha anlaşılabilir, geliştirilebilir ve sürdürülebilir olmasına yardımcı olur. İsmini ise her bir prensibin baş harflerinin kısaltmasından almıştır.

Single Responsibility Principle(Tek Sorumluluk): Her sınıfın sadece ama sadece bir işlevi olmalıdır. Birden çok işleve sahip yapılar, kodun karmaşıklığını arttırarak kontrolünü zorlaştırır.

```
public class Book {
private String name;
private String author;
private String text;
//constructor, getters and setters
// methods that directly relate to the book properties
public String replaceWordInText(String word){
      return text.replaceAll(word, text); }
public boolean isWordInText(String word){
      return text.contains(word); }
}

public class BookPrinter {
// methods for outputting text
      void printTextToConsole(String text){
//our code for formatting and printing the text }
      void printTextToAnotherMedium(String text){
// code for writing to any other location.. } }
```

Bu prensibe uygun olan kitap için ayrı, kitap içindeki bilgileri yazdırmak için ayrı bir sınıf örneği.

Kaynak:https://www.baeldung.com/solid-principles

Open-Closed Principle(Açık-Kapalı): Sınıflar geliştirmeye açık ama değiştirmelere kapalı olmalıdır. Yeni ihtiyaçlar ortaya çıktığında bunları karşılayıp aynı zamanda da çalışan yapıyı bozmamak amaçlanır.

```
public class VehicleCalculations {
    public double calculateValue(Vehicle v) {
        if (v instanceof Car) {
            return v.getValue() * 0.8;
        if (v instanceof Bike) {
            return v.getValue() * 0.5;

    }
}

public class Vehicle {
    public double calculateValue() {...}
}
public class Car extends Vehicle {
    public double calculateValue() {
        return this.getValue() * 0.8;
}
public class Truck extends Vehicle{
    public double calculateValue() {
        return this.getValue() * 0.9;
}
```

İlk yapıda kullanılan mantığı tercih etmek yerine ikinci yapıdakini seçmek, hem var olan Vehicle sınıfını korur hem de yeni araç eklenmesinde gereken değişiklikleri yapmayı kolaylaştırır.

Kaynak: https://www.educative.io/answers/what-are-the-solid-principles-in-java

Liskov Substitution Prensible(Yerine Geçme): Kalıtım hiyerarşisi kullanılarak türetilen bir sınıfın, kendisinden türetilmiş olan sınıf yerine kullanılabilir olmalıdır.

```
public class Rectangle {
    private double height;
    private double width;
    public void setHeight(double h) { height = h; }
    public void setWidht(double w) { width = w; }
    ...
}

public class Square extends Rectangle {
    public void setHeight(double h) {
        super.setHeight(h);
        super.setWidth(h);
    }
    public void setWidth(double w) {
        super.setHeight(w);
        super.setWidth(w);
    }
```

Square sınıfı, Rectangle sınıfından türetilmiş ancak farklı metotlara sahip olması nedeniyle onun yerine kullanılamaz çünkü beklenmeyen bir sonuç verir. Bu yüzden de sorunu çözmek zaman kaybına yol açar. Kaynak: https://www.educative.io/answers/what-are-the-solid-principles-in-java

Interface Segregation Principle(Arayüz Ayırımı): Arayüzler bir bütün olarak ve bütün işlevi içinde barındırmak yerine, küçük ve birbirinden farklı işlevlere hizmet eder şekilde tasarlanmalıdır. Çünkü bir sınıfın, ihtiyacı olmayan metotları kullanmaya zorlanmaması gerekir.

```
public interface Messenger {
  askForCard();
  tellInvalidCard();
  askForPin();
  tellInvalidPin();
  tellCardWasSiezed();
  askForAccount();
  tellNotEnoughMoneyInAccount();
  tellAmountDeposited();
  tellBalance();
}
```

```
-----------------------------------
```

```
public interface LoginMessenger {
  askForCard();
  tellInvalidCard();
  askForPin();
  tellInvalidPin();
}
```

```
public interface WithdrawalMessenger {
  tellNotEnoughMoneyInAccount();
  askForFeeConfirmation();
}
```

```
public class EnglishMessenger implements LoginMessenger,
WithdrawalMessenger {
  ...
}
```

Bir ATM arayüzü tasarımı ilk örnekteki gibi kompakt yerine ikinci kısımdaki gibi ilgili bölümlere ayrılarak tasarlanması tercih edilmelidir.

Kaynak: https://www.jrebel.com/blog/solid-principles-in-java

Dependency Inversion Principle(Bağımlılığın Ters Çevrilmesi): Üst sınıfların alt sınıflara olan bağımlılığını azaltarak, bütün yapılar daha soyut bir şekilde birbirine bağlanmalıdır.

```
public interface Reader { char getchar(); }
public interface Writer { void putchar(char c)}
```

```
class CharCopier {
```

```
  void copy(Reader reader, Writer writer) {
    int c;
    while ((c = reader.getchar()) != EOF) {
      writer.putchar();
```

```
    }
  }
}
```

```
public Keyboard implements Reader {...}
public Printer implements Writer {…}
```

Kaynak: https://www.jrebel.com/blog/solid-principles-in-java

CharCopier sınıfı çalışmak için girdi ve çıktılara ihtiyaç duyar ancak bunun detaylarını belirleyen sınıflar Keyboard ve Printer ayrıca tanımlanmıştır. Soyutlama işlemi yapılarak CharCopier sınıfının somut olarak bağımlılığı ortadan kaldırılarak tasarlanmıştır.

10)Rapid Application Development bir yazılım geliştirme yaklaşımıdır. Belirli bir plana göre hareket etmekten ziyade prototipler yaparak ilerlenir. Proje küçük parçalara ayrılarak paralel bir şekilde geliştirilip, hızlı bir çıktı almak hedeflenir.

11)Spring Boot Starters, bağımlılık tanımlayıcıları olup projenin çalışması için gerekli olan dosyaları otomatik olarak yükleyen ve yöneten yapılardır. Yapılandırma sürecini azaltarak geliştiricilerin verimliliğini arttırır. Test edilmiş, çalışmaya hazır ve projeye eklenmesi kolaydır.

12)Spring Boot Annotations, ilgili kod hakkında üst veri(metadata) sağlayan yapılardır. Başka bir ifadeyle, program hakkında tamamlayıcı bilgiler veren yapılardır. Yazılıma direkt olarak etki etmez ya da değiştirmez.

13)Dependency Management, bağımlılık yönetiminin merkezi bir şekilde halledilmesine olanak sağlayan mekanizmanın kendisidir. Spring Boot, bağımlılıkları ve yapılandırmaları otomatik olarak yapan bir araçtır. Maven veya Gradle gibi araçlar bu bağımlılıkları yönetmede kullanılır.

14)Actuator, Spring Boot uygulamalarının değerlerini(metric) gözlemleme ve yönetme imkânı sağlayan bir özelliktir. HTTP veya JMX uç noktaları(endpoints) kullanılarak ilgili değerlere erişilebilir.