Pola:

1. **Instance** To pole przypisane do instancji klasy. Każda instancja klasy ma swoją kopię tego pola.

2. **Static:** To pole przypisane do samej klasy, a nie do konkretnych instancji. Wszystkie instancje klasy mają dostęp do tego samego pola statycznego.

3. **Final:** To pole, które można zainicjować tylko raz i nie można go później zmienić. Jest to często używane do zadeklarowania stałych.

4.**Private:** To pole zadeklarowane jako "private", co oznacza, że jest dostępne tylko w obrębie klasy, w której zostało zdefiniowane.

5. **Protected:** To pole zadeklarowane jako "protected", co oznacza, że jest dostępne w obrębie klasy oraz jej podklas.

6. **Package-Private:** To pole, które nie ma określonego modyfikatora dostępu (nie jest "public", "private" ani "protected") i jest dostępne tylko w obrębie pakietu, w którym się znajduje.

Zmienne:

1. **byte:** Zmienna używana do przechowywania liczb całkowitych o rozmiarze 8 bitów.

2. **short:** Zmienna używana do przechowywania krótkich liczb całkowitych o rozmiarze 16 bitów.

3. **int:** Zmienna używana do przechowywania liczb całkowitych o rozmiarze 32 bitów.

4. **long:** Zmienna używana do przechowywania długich liczb całkowitych o rozmiarze 64 bitów.

5. **float:** Zmienna używana do przechowywania liczb zmiennoprzecinkowych o pojedynczej precyzji.

6. **double:** Zmienna używana do przechowywania liczb zmiennoprzecinkowych o podwójnej precyzji.

7. **char:** Zmienna używana do przechowywania pojedynczych znaków.

8. **boolean:** Zmienna używana do przechowywania wartości logicznych (prawda/fałsz).

9. Referencje: Zmienne referencyjne służą do przechowywania referencji do obiektów i innych typów referencyjnych.

Klasy:

1. **Klasa bazowa** (Base Class): Klasa, która może być używana jako punkt wyjścia do tworzenia innych klas. Inne klasy mogą dziedziczyć po klasie bazowej.

2. **Klasa pochodna** (Derived Class): Klasa, która dziedziczy cechy i metody od klasy bazowej. Może dodawać lub modyfikować zachowanie odziedziczone z klasy bazowej.

3. **Klasa abstrakcyjna** (Abstract Class): Klasa abstrakcyjna w języku Java to specjalny rodzaj klasy, która nie może być bezpośrednio instancjonowana. Jest to klasa, która służy jako szablon do tworzenia innych klas. Klasy abstrakcyjne są używane do wprowadzenia pewnego stopnia struktury i organizacji w kodzie oraz do narzucenia pewnych wymagań dla klas pochodnych.

Kluczowe cechy klas abstrakcyjnych:

1. Klasy abstrakcyjne są oznaczane słowem kluczowym `abstract`. Oznacza to, że deklaracja klasy rozpoczyna się od słowa `abstract`, na przykład: `abstract class AbstrakcyjnaKlasa`.

2. Klasy abstrakcyjne mogą zawierać metody abstrakcyjne. Metody abstrakcyjne to metody, które nie mają implementacji w samym ciele klasy abstrakcyjnej. Zamiast tego, deklarują tylko sygnaturę metody. Metody abstrakcyjne są oznaczane słowem kluczowym `abstract`.

3. Klasy abstrakcyjne mogą również zawierać metody z implementacją. Nie muszą jednak zawierać żadnych metod abstrakcyjnych. Klasy te mogą dziedziczyć po klasach abstrakcyjnych i nadpisywać metody z implementacją lub tworzyć nowe metody.

4. Klasy dziedziczące (podklasy) od klasy abstrakcyjnej muszą zaimplementować wszystkie jej metody abstrakcyjne, inaczej będą również musiały być oznaczone jako abstrakcyjne. Można to zrobić za pomocą słowa kluczowego `extends`.

Przykład klasy abstrakcyjnej i jej podklasy:

KlasaAbstrakcyjna to klasa abstrakcyjna, a PochodnaKlasa to jej podklasa, która implementuje metodę abstrakcyjną. W głównym programie możemy utworzyć obiekt typu KlasaAbstrakcyjna i przypisać mu instancję PochodnaKlasa. W ten sposób możemy korzystać z metod obu klas.

4. **Klasa finalna** (Final Class): Klasa, której nie można dziedziczyć. Jest to ostateczna wersja klasy i nie można tworzyć pochodnych klas.

5. **Klasa generyczna** (Generic Class): Klasa, która może operować na różnych typach danych. Parametry typu pozwalają na tworzenie bardziej ogólnych i elastycznych klas.

6. **Klasa anonimowa** (Anonymous Class): Klasa, która jest definiowana bez nazwy i jest używana do tworzenia instancji obiektu z jednorazową implementacją.

7. **Klasa wewnętrzna** (Inner Class): Klasa zdefiniowana wewnątrz innej klasy. Może mieć dostęp do prywatnych pól i metod klasy nadrzędnej.

8. **Klasa statyczna** (Static Class): Klasa, która jest związana z klasą nadrzędną, a nie z instancją. Może zawierać tylko statyczne pola i metody.

9. **Klasa publiczna** (Public Class): Klasa dostępna z dowolnego miejsca w kodzie Java.

10. **Klasa prywatna** (Private Class): Klasa dostępna tylko w obrębie klasy nadrzędnej, nie jest widoczna na zewnątrz.

11. **Klasa chroniona** (Protected Class): Klasa dostępna w obrębie pakietu i dla klas dziedziczących, ale nie jest publicznie dostępna.

12. **Klasa pakietowa** (Package-Private Class): Klasa dostępna tylko w obrębie tego samego pakietu, nie jest publicznie dostępna.

13. **Klasa enum** (Enum Class): Specjalny rodzaj klasy używany do definiowania stałych wartości lub wartości wyliczeniowych. Enumeracje są użyteczne w sytuacjach, gdy potrzebujemy reprezentować zestaw opcji, które są stałe i niezmienne w trakcie działania programu. Dzięki nim kod staje się bardziej czytelny i bardziej bezpieczny.

14. **Klasa interfejsu** (Interface Class): Klasa, która zawiera tylko sygnatury metod, ale nie implementacje. Służy do implementacji kontraktów i dziedziczenia wielokrotnego.

Interfejs w języku Java to abstrakcyjna struktura, która definiuje zestaw metod, które klasy implementujące ten interfejs muszą dostarczyć. Interfejsy są używane do ustalania kontraktów lub zobowiązań, które klasy muszą spełnić, aby być użyte w określony sposób. Oto kluczowe punkty związane z interfejsami w Javie:

1. Deklaracja interfejsu: Aby zdefiniować interfejs, używa się słowa kluczowego `interface`. Interfejsy są zazwyczaj nazwane zgodnie z konwencją zaczynającą się od dużej litery i zapisaną w notacji CamelCase. Interfejs nie zawiera implementacji metod, tylko ich sygnatury.

2. Implementacja interfejsu: Klasy w Javie mogą implementować (czyli "zrealizować") interfejsy, co oznacza, że muszą dostarczyć implementację wszystkich metod zdefiniowanych w interfejsie. Implementacja odbywa się za pomocą słowa kluczowego `implements`.

3. Wielokrotne dziedziczenie interfejsów: W Javie można implementować wiele interfejsów w jednej klasie. To oznacza, że klasa musi dostarczyć implementację wszystkich metod z tych interfejsów. Dzięki temu można korzystać z cech różnych interfejsów w jednej klasie.

4. Brak stanu: Interfejsy w Javie nie mogą zawierać pól (zmiennych instancji) ani konstruktorów. Służą one wyłącznie do definiowania metod, co oznacza, że nie mają stanu.

5. Polimorfizm: Interfejsy są często używane do osiągnięcia polimorfizmu. Oznacza to, że różne obiekty, które implementują ten sam interfejs, mogą być traktowane jednolicie. To pozwala na elastyczne korzystanie z różnych implementacji tego samego interfejsu w programie.

6. Kontrakt: Interfejsy definiują kontrakt lub umowę, które klasy implementujące muszą przestrzegać. Dzięki temu można zapewnić spójność w zachowaniu klas w programie.

Interfejsy są ważnym narzędziem w programowaniu obiektowym w Javie. Umożliwiają tworzenie abstrakcyjnych struktur, które pozwalają na rozdzielenie deklaracji (co klasa musi robić) od implementacji (jak to robi). Dzięki temu programiści mogą tworzyć bardziej elastyczne i skalowalne aplikacje, które łatwo można rozbudować i modyfikować.

15. **Klasa narzędziowa** (Utility Class): Klasa, która zawiera zestaw narzędziowych metod statycznych, często używanych do różnych operacji na danych.

16. **Klasa wzorca projektowego** (Design Pattern Class): Klasa, która jest często wykorzystywana jako szablon dla rozwiązań problemów w sposób sprecyzowany przez wzorzec projektowy.

17. **Klasa struktury danych** (Data Structure Class): Klasa zaprojektowana do przechowywania i zarządzania danymi w określony sposób, na przykład listy, stosy czy drzewa.

18. **Klasa obsługi wyjątków** (Exception Handling Class): W Javie klasa obsługi wyjątków to klasa, która jest używana do przechwytywania i obsługi wyjątków, czyli nieoczekiwanych błędów lub sytuacji, które mogą wystąpić w trakcie wykonywania programu. Wyjątki są specjalnymi obiektami, które reprezentują te sytuacje i pozwalają programowi reagować na nie w odpowiedni sposób, zamiast kończyć się awarią.

Klasy obsługi wyjątków w Javie dziedziczą z klasy `java.lang.Exception`. Klasy te zawierają metody i konstruktory, które pozwalają na zdefiniowanie, jak program ma reagować na konkretne rodzaje wyjątków. Główne metody w klasie `Exception` to:

1. `getMessage()`: Zwraca komunikat opisujący wyjątek.
2. `printStackTrace()`: Wypisuje ślad stosu (stack trace) wyjątku, co jest przydatne do debugowania.

Oprócz klasy `Exception`, w Javie istnieje wiele innych klas wyjątków, które dziedziczą z `Exception`, takie jak `RuntimeException`, `IOException`, `NullPointerException`, itp. Programista może także tworzyć własne klasy wyjątków, dziedzicząc je z `Exception` lub innych klas wyjątków, w zależności od potrzeb.

Jak działa obsługa wyjątków w Javie:

1. W kodzie źródłowym programu programista używa bloków `try-catch` do otoczenia kodu, który może wygenerować wyjątek. Blok `try` zawiera kod, który ma zostać wykonany, a blok `catch` zawiera kod, który zostanie wykonany w przypadku zgłoszenia wyjątku.

2. Jeśli w bloku `try` zostanie wygenerowany wyjątek, Java próbuje dopasować go do odpowiedniego bloku `catch` na podstawie typu wyjątku. Jeśli pasujący blok `catch` zostanie znaleziony, kod w tym bloku zostaje wykonany.

3. Po zakończeniu obsługi wyjątku, wykonanie programu kontynuuje się poza blokiem `try-catch`.

4. Można również używać bloków `finally`, które zawierają kod, który zostanie wykonany niezależnie od tego, czy wyjątek został zgłoszony, czy nie. To jest przydatne, aby upewnić się, że pewne zasoby zostaną zwolnione niezależnie od tego, co się wydarzy.

Obsługa wyjątków pozwala programowi na bardziej kontrolowane i bezpieczne reagowanie na nieoczekiwane sytuacje, co jest istotne w rozwijaniu stabilnych i niezawodnych aplikacji.

19. **Klasa do testów jednostkowych** (Unit Testing Class): Klasa, która zawiera testy jednostkowe do weryfikacji poprawności kodu.

20. **Rekord** (Record): Rekord w języku Java to specjalny rodzaj klasy wprowadzony w wersji Java 16 i dostępny w kolejnych wersjach. Jest to wygodny sposób tworzenia klas przeznaczonych do przechowywania danych. Oto główne cechy rekordów:

1. Deklaracja: Rekordy deklarowane są za pomocą słowa kluczowego `record`. Na przykład, `record Person(String name, int age)` tworzy rekord o nazwie `Person` z dwoma polami: `name` i `age`.

2. Automatyczne generowanie metod: Java automatycznie generuje metody takie jak `equals()`, `hashCode()`, `toString()`, a także metody dostępowe (gettery) dla pól rekordu.

3. Niemutowalność: Pola rekordu są niemutowalne, co oznacza, że po ich inicjalizacji nie można zmieniać ich wartości. Jeśli chcesz zmienić wartość pola, musisz utworzyć nowy obiekt rekordu z nowymi danymi.

Rekordy są przydatne do reprezentowania danych, takich jak osoby, produkty, daty, i wszędzie tam, gdzie ważna jest automatyczna generacja metod i niemutowalność danych.