Biblioteki

Krzysztof Gębicz

RANDOM

Czym jest?

Klasa Random generuje liczby losowe.

Przydatna w grach, testach, symulacjach.

Dlaczego używać?

Do losowania danych, np. liczb, booleanów, indeksów listy.

Kiedy i po co?

Gdy potrzebujesz elementu przypadkowo wybranego.

Zalety i Wady

Zalety:

✅ Łatwy w użyciu
✅ Obsługuje różne typy danych
✅ Współpracuje z kolekcjami

Wady:

❌ Nie jest bezpieczny kryptograficznie
❌ Wymaga inicjalizacji (zależność od ziarna)

Przykład

MATH

Czym jest?

Klasa Math zawiera statyczne metody do obliczeń matematycznych.

Dlaczego używać?

Dostarcza gotowych funkcji: potęgi, pierwiastki, zaokrąglenia

Kiedy i po co?

Gdy potrzebujesz obliczeń matematycznych bez pisania własnych metod.

Zalety i Wady

Zalety:

✅ Szybka i prosta w użyciu
✅ Dostępna bez tworzenia obiektu (Math.pow())

Wady:

❌ Operuje tylko na liczbach
❌ Brak zaawansowanej matematyki (macierze, statystyka)

Przykład

JAVA.TIME

Czym jest?

Nowoczesne API do obsługi dat i czasu (zastępuje Date i Calendar).

Dlaczego używać?

- Łatwe obliczenia na datach (np. różnice dni).
- Czytelne, bezpieczne i zgodne z ISO.

Kiedy i po co?

Gdy musisz mierzyć czas, planować zdarzenia, ustalać okresy.

Zalety i Wady

Zalety:

✅ Łatwe dodawanie dni, miesięcy, lat
✅ Brak problemów ze strefami czasowymi
✅ Duża precyzja i bezpieczeństwo

Wady:

❌ Wymaga znajomości nowego API
❌ Trudniejsze formatowanie

Przykład

MAP i HASHMAP

Czym jest?

Map to interfejs z pakietu java.util, który przechowuje dane w parach klucz–wartość.

Każdy klucz jest unikalny, a wartości mogą się powtarzać.

Najczęściej używana implementacja to HashMap.

Dlaczego używać?

- Pozwala szybko wyszukiwać dane po kluczu.
- Jest idealna do przechowywania powiązań, np. nazwiska–oceny, produkty–ceny.
- Umożliwia łatwą iterację po parach danych.

Kiedy i po co?

- Gdy potrzebujesz przechowywać dane z jednoznacznym identyfikatorem.
- W systemach, gdzie często pobiera się dane po nazwie, ID, itp.
- Do liczenia wystąpień elementów, mapowania lub tworzenia słowników.

Zalety i Wady

Zalety

✅ Bardzo szybkie wyszukiwanie
✅ Elastyczność – różne implementacje (HashMap, TreeMap, LinkedHashMap)
✅ Łatwe dodawanie, usuwanie, aktualizowanie danych

Wady

❌ Brak zachowania kolejności (dla HashMap)
❌ Nie dopuszcza duplikatów kluczy
❌ Nie jest bezpieczna dla wielu wątków

Przykład

SET i HASHSET

Czym jest?

Set to kolekcja, która nie pozwala na duplikaty.
Najczęściej używana implementacja to HashSet.

Dlaczego używać?

- Gdy musisz przechować unikalne wartości.
- Zapewnia szybkie sprawdzanie, czy element już istnieje.

Kiedy i po co?

- Do filtrowania powtórzeń.
- Do przechowywania unikalnych identyfikatorów lub tagów.

Zalety i Wady

Zalety:

✅ Gwarancja unikalności
✅ Bardzo szybkie wyszukiwanie (HashSet)
✅ Możliwość sortowania (TreeSet)

Wady:

❌ Brak dostępu po indeksie
❌ Kolejność nie zawsze zachowana
❌ Mniej intuicyjna niż lista

Przykład

OPTIONAL

Czym jest?

Optional to kontener, który może przechowywać wartość lub null w bezpieczny sposób.

Zastępuje ryzykowne sprawdzanie if (x != null).

Dlaczego używać?

- Chroni przed NullPointerException.
- Zmusza programistę do świadomego sprawdzenia, czy wartość istnieje.

Kiedy i po co?

- Gdy metoda może nie zwrócić wyniku.
- W API i logice biznesowej, gdzie null jest możliwy, ale niepożądany.

Zalety i Wady

Zalety:

✅ Bezpieczny i czytelny kod
✅ Eliminuje błędy z nullami
✅ Integruje się ze Stream i Lambda

Wady:

❌ Może utrudniać debugowanie
❌ Nie zastępuje logiki biznesowej
❌ Lekki narzut na wydajność

Przykład

Comparator

Czym jest Comparator?

`Comparator` to interfejs funkcyjny z pakietu `java.util`, który definiuje sposób porównywania dwóch obiektów. Pozwala ustalić zasady, według których obiekty będą sortowane lub porównywane.

Comparator działa niezależnie od klasy obiektu — nie trzeba modyfikować samej klasy, by określić jej porządek.

Dlaczego potrzebujemy Comparatora?

Nie wszystkie obiekty mają naturalny porządek.
`Comparator` pozwala:

- ustalić własne zasady porównywania,
- definiować różne porządki dla tych samych danych,
- oddzielić logikę porównywania od klasy obiektu,
- kontrolować sposób sortowania w różnych sytuacjach.

Kiedy używać Comparatora?

Używamy, gdy:

- potrzebujemy różnych sposobów porównywania,
- nie chcemy lub nie możemy modyfikować klasy.

Zalety Comparatora

✅ Oddzielenie danych od logiki porównywania.

✅ Możliwość tworzenia wielu niezależnych sposobów porównania.

✅ Współpraca z metodami bibliotecznymi (`Collections`, `Arrays`).

✅ Elastyczność i ponowne użycie kodu.

✅ Łatwiejsze zarządzanie danymi w dużych projektach.

Wady Comparatora

❌ Wymaga pisania dodatkowych klas lub obiektów.

❌ Może być trudniejszy do zrozumienia.

❌ Wiele Comparatorów – trudniejsze utrzymanie kodu.

❌ Wprowadza dodatkową warstwę abstrakcji.

Statyczne metody w klasie Comparator

Od Javy 8 interfejs `Comparator` ma gotowe metody fabryczne:

- `Comparator.comparing(...)` – porównuje wg wybranej cechy,
- `Comparator.reverseOrder()` – odwraca porządek,
- `Comparator.nullsFirst(...)`, `Comparator.nullsLast(...)` – bezpieczne porównywanie z wartościami `null`.

Przykład

Collections

Wprowadzenie do Collections

Collections to narzędziowa klasa (utility class) w pakiecie java.util.
Zawiera zestaw statycznych metod ułatwiających pracę z kolekcjami – listami, zbiorami, mapami.

Nie tworzymy jej obiektów – korzystamy z metod klasy bezpośrednio.

Po co jest klasa Collections?

Ułatwia wykonywanie typowych operacji na kolekcjach:

- sortowanie,
- wyszukiwanie,
- kopiowanie,
- odwracanie,
- synchronizacja,
- tworzenie niezmiennych kolekcji.

Dzięki niej nie trzeba pisać tych operacji samodzielnie.

Przykładowe zastosowania Collections

- Sortowanie list obiektów według naturalnego porządku lub z Comparator.
- Wyszukiwanie elementów w posortowanych listach.
- Tworzenie niezmiennych kolekcji.
- Losowe tasowanie danych (shuffle).
- Odwrócenie kolejności elementów (reverse).
- Synchronizacja kolekcji w programach wielowątkowych.

Zalety klasy Collections

✅ Szybka i bezpieczna praca z kolekcjami.

✅ Gotowe, wydajne metody.

✅ Spójność w całej bibliotece Javy.

✅ Integracja z Comparator.

✅ Łatwość stosowania i czytelność kodu.

Wady klasy Collections

❌ Ograniczona elastyczność – tylko dla kolekcji.

❌ Nie zawsze najbardziej wydajna w dużych zbiorach.

❌ Brak pełnej kontroli nad procesem sortowania lub kopiowania.

Przykład

Objects

Wprowadzenie do klasy Objects

`Objects` to klasa narzędziowa (`utility class`) z pakietu `java.util`.

Jej głównym celem jest bezpieczna praca z obiektami, szczególnie w sytuacjach, gdy istnieje ryzyko wystąpienia `null`.

Dzięki niej można:

- bezpiecznie porównywać obiekty,
- obliczać skróty (`hash`),
- generować opisy (`toString`),
- porównywać złożone obiekty (np. tablice).

Najważniejsze metody klasy Objects

- `Objects.equals(a, b)` – bezpieczne porównanie dwóch obiektów.
- `Objects.deepEquals(a, b)` – porównuje struktury zagnieżdżone, np. tablice.
- `Objects.compare(a, b, comparator)` – używa wskazanego Comparatora.
- `Objects.hash(a, b, c...)` – tworzy kod skrótu (hash).
- `Objects.requireNonNull(obj)` – sprawdza, czy obiekt nie jest `null`.
- `Objects.toString(obj, default)` – bezpieczne wyświetlenie obiektu

Dlaczego warto używać Objects?

Ponieważ wiele operacji w Javie może powodować błędy, jeśli obiekt jest `null`.

Metody klasy `Objects` zapewniają:

- bezpieczeństwo (brak `NullPointerException`),
- czytelność kodu,
- uniwersalność – działają dla wszystkich typów obiektów.

Zalety klasy Objects

✅ Bezpieczne operacje na obiektach,

✅ Ochrona przed błędami `null`,

✅ Prostszy, krótszy kod,

✅ Wysoka czytelność,

✅ Współpraca z Comparator i kolekcjami,

✅ Pomocna w testach i walidacji danych.

Wady klasy Objects

❌ Działa tylko na poziomie podstawowych porównań (nie zawsze wystarcza dla skomplikowanych klas),

❌ Wymaga znajomości metod, by używać ich prawidłowo.

Kiedy używać klasy Objects

- Zawsze, gdy istnieje ryzyko wystąpienia `null`,
- Gdy chcemy uniknąć błędów przy porównywaniu obiektów,
- W testach, walidacji danych i metodach pomocniczych,
- Gdy potrzebujemy porównać złożone struktury, np. tablice lub listy,
- Gdy generujemy wartości hash lub chcemy je porównać.

Znaczenie klasy Objects w porównywaniu całych obiektów

Klasa `Objects` stanowi uzupełnienie Comparatora— umożliwia bezpieczne, neutralne i niezależne porównywanie obiektów oraz struktur danych.

Dzięki niej Java pozwala porównywać całe obiekty:

- bez ryzyka błędu,
- bez dodatkowego kodu,
- w sposób czytelny i uniwersalny.

Zależność między Comparator, Collections i Objects

Element	Główna funkcja	Współpraca
Comparator	Określa sposób porównywania	używany przez `Collections` i `Objects`
Collections	Wykonuje operacje na danych	korzysta z `Comparator`
Objects	Porównuje i zabezpiecza obiekty	używa `Comparator`

Praktyczne korzyści dla programisty

✅ Bezpieczny kod (mniej błędów `null`),

✅ Czytelna i spójna logika,

✅ Łatwość konserwacji i rozbudowy,

✅ Szybka praca z kolekcjami,

✅ Pełna kontrola nad sposobem porównywania danych.

Dobre praktyki

- Zawsze sprawdzaj możliwość wystąpienia `null`,
- Korzystaj z `Objects` przy walidacji danych,
- Oddzielaj logikę porównywania (Comparator) od logiki biznesowej,
- Używaj metod `Collections` zamiast własnych algorytmów,
- Dokumentuj i nazywaj swoje comparatory w sposób zrozumiały.

Podsumowanie

- Comparator – określa sposób porównywania obiektów,
- Collections – dostarcza narzędzia do operacji na kolekcjach,
- Objects – umożliwia bezpieczne porównywanie całych obiektów i obsługę `null`.

Razem tworzą trzy filary bezpiecznego i elastycznego zarządzania danymi w Javie.

Ich znajomość to podstawa profesjonalnego programowania obiektowego.