Interfaces and Lambda Expressions:

interface:

- ממשק בא כדי להגדיר חוזה בין שני צדדים, כאשר צד אחד מתחייב לממש את הפונקציה שכתובות בממשק, אם הוא רוצה להשתמש בו.
- ממשק יכול להכיל פונקיציות אבסטרקטיות ומשתנים constant, בנוסף בג׳אווה 8, הוסיפו לממשקים default functions ו- static functions. ההבדל בינהם הוא שפונקציה סטטית אי אפשר לדרוס, ופונציה דיפולטית אפשר. בנוסף אי אפשר לקרוא לפונקציה סטטית מאינסטנס של הממשק.
- סכנות שחשוב לדעת: כשאר מממשים שני ממשקים עם אותה חתימה של פונקציה דיפולטית מקבלים שגיאת קומפילציה. כדי לגבור על זה צריך לדרוס את הפונקציה הדיפולטית הזאת

lambda expression:

- lambda expression מבטא מימוש של functional interface.
- lambda expression מממשת את הפונקציה האבסטרקטית היחידה שיש בממשק.
- lambda expression מספק את ההתנהגויות הבאות:
  - מאפשר להעביר מטודות כארגומנטים.
  - ליצור פונקציה שלא שלא שייכת לשום מחלקה
  - מאפשר להעביר מטודות כאובייקט ולהפעיל אותו כשצריך.

Best practices with lambda expressions and functional interfaces:

1. שימוש בstandard functional interfaces: המימושים הקיימים בחבילה java.util.function מפסק את הצרכים של רוב המפתחים. עדיף להשתמש בממשק קיים מאשר לכתוב אחד חדש שעושה את אותה הפעולה.
2. שימוש באנטוציה @FunctionalInterface: ממשקים שעוצבו כדי להיות ממשקים פונקציונלים, מובטחים להישאר ככה כאשר משתמשים באנוטציה, מכיוון שככה אי אפשר להוסיף פונקציה אבסטרקטית לממשק.
3. לא להשתמש יותר מדי בdefault functions:
   1. ניתן להוסיף כמה פונקציות דיפולטיות שרוצים, אבל כשאר מממשים שני ממשקים עם אותה חתימה של פונקציה דיפולטית מקבלים שגיאת קומפילציה. כדי לגבור על זה צריך לדרוס את הפונקציה הדיפולטית הזאת.
   2. שימוש מוגזם בפונקציות דיפולטיות מראה על עיצוב לא נכון של הממשק.
4. איתחול ממשקים פונקציונלים על ידי lambda expression: עדיף לאתחל ממשק פונקציונלי על ידי lambda expression מאשר להשתמש במחלקה אנונימית.
5. הימנע מoverload של פונקציות שמקבלות ממשקים פונקציונלים כפרמטר: בגלל שממשים את הממשקים עם lambda expression יכולה להיות התנגשות בחתימות, למשל callable ו supplier שמחזירות String. פתרונות להימנע:
   1. הכנסת סוג הממשק לשם הפונקציה
   2. לעשות cast לערך בקריאה לפונקציה
6. לא להתייחס לlambda expression כמו מחלקה פנימית: ההבדל בינהם הוא הscope:
   1. מחלקה פנימית: שימוש במחלקה פנימית מוסיף scope חדש. אפשר להסתיר מ

שתנים פנימיים של המחלקה. בנוסף שימוש בthis זה רפרנס לאובייקט עצמו.

2. lambda expression:: עובד עם enclosing scope, כלומר הscope שעוטף את הלמבדה. אי אפשר להסתיר משתנים פרטיי ם. שימוש בthisזה רפרנס לscope שעוטף.

דוגמא:

```
private String value = "Enclosing scope value";
```

```
    Foo fooLambda = parameter -> {
        String value = "Lambda value";
        return this.value;
    };
```

התוצאה שתחזור היא "Enclosing scope value".

7. אורך הlambda expression אמור להיות קצר וברור: עדיף לממש את הפונקציה בשורה (ביטוי) ולא בבלוק של פקודות. ניתן להגיע לזה בכמה דרכים:
   1. הוצאה לפונקציה: ברוב המקרים עדיף להוציא את הלוגיקה שרוצים לעשות לפונקציה וביטוי למבדה רק לקרוא לפונקציה. (לא תמיד זה עדיף)
   2. *לא לציין את הtype של המשתנים: הקומפיילר יכול להבין מה הtype, לכן זה מיותר.
   3. להימנע מהוספת סוגריים כשיש פרמטר אחד לפונקציה.
   4. להימנע מסולסלים וreturn בפונקציות של שורה אחת.
   5. להשתמש בmethod reference: ברוב המקרים במימוש של הlambda קוראים לפונקציה שכבר ממומשת, לכן בשביל קריאות עדיף להשתמש ב method reference.
8. השתמשו במשתנים שהם "effectively final": אם עורכים משתנים שהם לא final מתוך lambda expression מקבלים שגיאת קומפילציה. לפי הקונספט של "effectively final" הקומפיילר מתייחס למשתנים שמאותחלים פעם אחת בדיוק כמשתנים שהם final.
9. המנעו מעדכון אובייקט בlambda expression: ניתן לעדכן data בתוך אובייקט, לכן אפשר לשנות את המצב של האובייקט. כלומר במצב כזה האובייקט הוא לא thread safe.

Functional Interfaces:

- ממשק פונקציונלי הוא ממשק שמכיל פונקציה אחת שיש לממש (פונקציה אבסטרקטית אחת). הממשק מייחצן פונקציה התנהגותית אחת כלפי חוץ. הממשק יכול להכיל פונקציות פרטיות (default function) ככמה שהוא רוצה.
- האנטוציה @functionalinterface מבטיחה שלממשק המתאים אין יותר מפונקציה אחת למימוש (abstract function).
- האנוטציה היא לא חובה.
- ניתן לממשק ממשק פונקציונלי על ידי שימוש בביטויי למבדה (lambda expression).
- דוגמאות לממשקים כאלה: Runnable, Predicate, Comperator.

Function:

- ממשק שמייצג פונקציה שמקבלת ערך מסוג גנרי R ומחזירה ערך מסוג גנרי T.
- הפונקציה המרכזית בממשק היא apply.
- בגלל שזה ממשק פונקציונלי ניתן לממש אותו על ידי lambda expression, ולפנות אליו על ידי method reference.
- על ידי שימוש ב ()compose ניתן לשלב פונקציות להרצה אחת
- יש מימושים מוכנים של java לפונקציות נפוצות:
  - XXXFunction: שמקבלת type ידוע ומחזירה R, למשל IntFunction.
  - ToXXXFunction: שמקבלת T ומחזירה type ידוע, למשל ToDoubleFunction.
  - XXXToYYYFunction: שמקבלת type_x ידוע ומחזירה type_y ידוע, למשל IntToLongFunction.

BiFunction:

- ממשק שמייצג פונקציה שמקבלת שני ערכים מסוג T, U ומחזירה ערך מסוג R.
- הפונקציה המרכזית בממשק היא apply.
- אותו דבר כמו Function.
- שימוש נפוץ בפונקציה היא ב Map.replaceAll, שמקבלת את הkey והvalue של הרשומה ומכניסה ערך חדש לאותו key לפי החישוב בפונקציה.

Supplier:

- ממשק שמייצג פונקציה שלא מקבלת ארגומנטים ומייצרת ערך מסוג גנרי T.
- בעיקרון זה ייצור אובייקטים בצורה lazy, כלומר רק כשצריך (כמו generator)
- הסונקציה הראשית בממשק היא ()get.
- הממשק מתאים כאשר לא צריך לספק ארגומנטים וצריך לקבל ערך בחזרה.
- שימוש נוסף בממשק הוא בפונקציה Stream.generate שמקבל supplier
- יש מימושים מוכנים של java שה type שחוזר ידוע מראש. מימוש מהצורה XXXSupplier למשל IntSupplier.

Consumer:

- ממשק שמקבל ערך מסוג T ולא מחזירה כלום.
- הממשק מייצג פונקציה עם side effect.
- הפונקציה הראשית בממשק היא ()accept.
- שימוש נפוץ בפונקציה היא ב List.forEach שמקבלת Consumer ומפעילה אותו על כל הערכים ברשימה.
- יש מימושים מוכנים של java שה type שהפונקציה מקבלת ידוע מראש. מימוש מהצורה XXXConsumer למשל IntConsumer.

BiConsumer:

- ממשק שמקבל שני ערכים מסוג T,U ולא מחזירה כלום.
- הפונקציה הראשית בממשק היא ()accept.
- ש מימושים מוכנים של java שה type שהפונקציה מקבלת ידוע מראש, והערך שני הוא מסוג T. מימוש מהצורה ObjXXXConsumer למשל ObjIntConsumer.

Predicate:

- ממשק שמקבל ערך T ומחזירה ערך בוליאני.
- הפונקציה הראשית בממשק היא ()test.
- שימוש נפוץ הוא ב stream.filter שמקבלת Predicate ומפלטרץ את האוסף לפי הפרידקט.

Operators:

- זה מקרה פרטי של Function שמקבל ערך T ומחזיר ערך מאותו סוג T.
- יש מימושים מוכנים של java שה type שהפונקציה מקבלת ידוע מראש:
  - XXXUnaryOperator: פונקציות אלה מקבלות ערך מאותו סוג T ומחזירות ערך מסוג T. למשל IntUnaryOperator.
  - XXXBinaryOperator: פונקציות אלה מקבלות שני ערכים מאותו סוג T ומחזירות ערך מסוג T. למשל IntBinaryOperator.

AutoCloseable:

- **ממשק שמאפשר סגירה אוטומטית של משאבים על ידי שימוש ב-try-with-resources .**
- **הפונקציה הראשית בממשק היא ()close.**
- הפונקציות בחבילה Java.io ממשות את הממשק הזה.

Closeable:

- ממשק שמסמל מחלקה שאפשר לסגור.
- מ7 Java, המחלקה מממשת את הממשק autoClosable, לכן ניתן להשתמש במחלקות שמשמשות אותה **בtry-with-resources .**
- **הפונקציה הראשית בממשק היא ()close.**

Iterator:

- ממשק שמאפשר לעבור על איברים באוסף (List, Set, Map).
- פונקציות שמאפשרות מעבר על האוסף:
  - hasNext: מחזיר False אם הגענו לסוף האוסף, אחרת מחזיר True.
  - next: מחזיר את האיבר הבא באוסף.
  - remove: מסיר מהאוסף את האיבר האחרון שהוחזר על ידי הiterator.
  - forEachRemaining: מבצע פעולה על האיברים שנשארו באוסף, בצורה סינכרונית.

** כשעוברים על רשימה עם foreach וקוראים לפונקציה remove מקבלים שגיאת ConcurrentModificationException

ListIterator:

- ממשק שמאפשר לעבור על איברים ברשימה משני הכיוונים.
- מחזיק שני מצביעים לרשימה, ניתן לקבל את הערכים שלהם בפונקציות previous, next.

- הממשק מכיל את הפונקציו:
- add(E e): הכנסה של איבר לרשימה.
- set(E e): החלפת הערך האחרון שחזר מ previous או next.
- remove: הסרת הערך האחרון שחזר מ previous או next.

Spliterator:

- ממשק שמאפשר מעבר ופיצול של רצף של מידע. משתמשים בו ב Fork-Join וב Stream.
- הממשק מכיל את הפונקציות:
  - tryAdvance: הפונקציה הראשית שמשמשת לעבור על רצף של איברים.
  - trySplit: מפצל את האיברים לשני Spliterators, שמכילים חצי מהכמות של ה Spliterator הראשון.
  - estimatedSize : מחזיר את מספר האיברים שspliterator יעבור עליהם.
  - hasCharacteristics: מציג את המאפיינים של ה Spliterator.
- Spliterator Characteristics:
  - **SIZED:** אם זה מסוגל להחזיר מספר מדוייק של אלמנטים בפונקציה estimatedSize.
  - **SORTED:** אם זה עובר על אוסף ממויין
  - **SUBSIZED:** אם אנחנו מפצלים אינסטנס של Spliterator על ידי trySplit ומחזיר Spliterator-ים בעלי תכונה SIZED.
  - **CONCURRENT**: אם המקור יכול להיות מקבילי.
  - **DISTINCT**: אם כל זוג של אלמנטים שונים אחד מהשני
  - **IMMUTABLE**: אם אי אפשר לשנות את המקור
  - **NONNULL**: אם אין ערכים שהם null.
  - **ORDERED**: אם עוברים על אוסף ממויין.

Iterable:

- ממשק מרכזי ב Java Collections API.
  מחלקה שמממשת את הממשק הזה מאפשרת לעבור על האוסף ב for-each loop.
- הממשק מכיל את הפונקציות:
  - iterator: שמחזיר Iterator שמאפשר לעבור על האוסף.
  - foreach: פונקציה דיפולטית, שמפעילה פונקציה על כל אחד מהאיברים באוסף.
  - spliterator: פונקציה דיפולטית, שמחזירה Spliterator.