Podstawy języka JAVA

Wykład nr 2 – pliki, obiektowość w języku JAVA: klasy, obiekty, referencje

Wyższa Szkoła Informatyki Stosowanej i Zarządzania
pod auspicjami Polskiej Akademii Nauk

Java - pliki

- W celu wykonywania operacji na plikach należy utworzyć obiekt klasy File.
- Klasa File posiada następujące konstruktory:

Constructor Summary

Constructors

Constructor and Description
`File(File parent, String child)` Creates a new `File` instance from a parent abstract pathname and a child pathname string.
`File(String pathname)` Creates a new `File` instance by converting the given pathname string into an abstract pathname.
`File(String parent, String child)` Creates a new `File` instance from a parent pathname string and a child pathname string.
`File(URI uri)` Creates a new `File` instance by converting the given `file:` URI into an abstract pathname.

Java - pliki

- W celu uzyskania dostępu do danych z pliku należy utworzyć obiekt klasy Scanner.
- Jeden z konstruktorów klasy Scanner pozwala przekazać obiekt klasy File.

Constructors

Constructor and Description
`Scanner(File source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(File source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.

Java - pliki

- Kolejnym krokiem jest odczytanie informacji z pliku za pomocą odpowiedniej metody klasy Scanner.

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class Class1 {

    public static void main(String[] args)  throws FileNotFoundException {
        File plik = new File("d:\\lukasz\\plik.txt");
        Scanner scan = new Scanner(plik);
        String zdanie = scan.nextLine();
        System.out.println(zdanie);
    }

}
```

Java - pliki

Methods

Modifier and Type	Method and Description
boolean	**canExecute**() Tests whether the application can execute the file denoted by this abstract pathname.
boolean	**canRead**() Tests whether the application can read the file denoted by this abstract pathname.
boolean	**canWrite**() Tests whether the application can modify the file denoted by this abstract pathname.
int	**compareTo**(**File** pathname) Compares two abstract pathnames lexicographically.
boolean	**createNewFile**() Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
static **File**	**createTempFile**(**String** prefix, **String** suffix) Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
static **File**	**createTempFile**(**String** prefix, **String** suffix, **File** directory) Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.
boolean	**delete**() Deletes the file or directory denoted by this abstract pathname.
void	**deleteOnExit**() Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
boolean	**equals**(**Object** obj) Tests this abstract pathname for equality with the given object.
boolean	**exists**() Tests whether the file or directory denoted by this abstract pathname exists.

Java - pliki

File	**getCanonicalFile**() Returns the canonical form of this abstract pathname.
String	**getCanonicalPath**() Returns the canonical pathname string of this abstract pathname.
long	**getFreeSpace**() Returns the number of unallocated bytes in the partition **named** by this abstract path name.
String	**getName**() Returns the name of the file or directory denoted by this abstract pathname.
String	**getParent**() Returns the pathname string of this abstract pathname's parent, or `null` if this pathname does not name a parent directory.
File	**getParentFile**() Returns the abstract pathname of this abstract pathname's parent, or `null` if this pathname does not name a parent directory.
String	**getPath**() Converts this abstract pathname into a pathname string.
long	**getTotalSpace**() Returns the size of the partition **named** by this abstract pathname.
long	**getUsableSpace**() Returns the number of bytes available to this virtual machine on the partition **named** by this abstract pathname.
int	**hashCode**() Computes a hash code for this abstract pathname.
boolean	**isAbsolute**() Tests whether this abstract pathname is absolute.
boolean	**isDirectory**() Tests whether the file denoted by this abstract pathname is a directory.
boolean	**isFile**() Tests whether the file denoted by this abstract pathname is a normal file.

Java - pliki

boolean	**isHidden**() Tests whether the file named by this abstract pathname is a hidden file.
long	**lastModified**() Returns the time that the file denoted by this abstract pathname was last modified.
long	**length**() Returns the length of the file denoted by this abstract pathname.
String[]	**list**() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
String[]	**list**(**FilenameFilter** filter) Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]	**listFiles**() Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
File[]	**listFiles**(**FileFilter** filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]	**listFiles**(**FilenameFilter** filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
static **File**[]	**listRoots**() List the available filesystem roots.
boolean	**mkdir**() Creates the directory named by this abstract pathname.
boolean	**mkdirs**() Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories.
boolean	**renameTo**(**File** dest) Renames the file denoted by this abstract pathname.

Java - pliki

boolean	**setExecutable**(boolean executable) A convenience method to set the owner's execute permission for this abstract pathname.
boolean	**setExecutable**(boolean executable, boolean ownerOnly) Sets the owner's or everybody's execute permission for this abstract pathname.
boolean	**setLastModified**(long time) Sets the last-modified time of the file or directory named by this abstract pathname.
boolean	**setReadable**(boolean readable) A convenience method to set the owner's read permission for this abstract pathname.
boolean	**setReadable**(boolean readable, boolean ownerOnly) Sets the owner's or everybody's read permission for this abstract pathname.
boolean	**setReadOnly**() Marks the file or directory named by this abstract pathname so that only read operations are allowed.
boolean	**setWritable**(boolean writable) A convenience method to set the owner's write permission for this abstract pathname.
boolean	**setWritable**(boolean writable, boolean ownerOnly) Sets the owner's or everybody's write permission for this abstract pathname.
Path	**toPath**() Returns a **java.nio.file.Path** object constructed from the this abstract path.
String	**toString**() Returns the pathname string of this abstract pathname.
URI	**toURI**() Constructs a `file:` URI that represents this abstract pathname.
URL	**toURL**() **Deprecated.** *This method does not automatically escape characters that are illegal in URLs. It is recommended that new code convert an abstract pathname into a URL by first converting it into a URI, via the **toURI** method, and then converting the URI into a URL via the **URI.toURL** method.*

Java - pliki

- Zapis do pliku w Java realizuje się poprzez klasę PrintWriter.

Constructor Summary

Constructors

Constructor and Description
`PrintWriter(File file)` Creates a new PrintWriter, without automatic line flushing, with the specified file.
`PrintWriter(File file, String csn)` Creates a new PrintWriter, without automatic line flushing, with the specified file and charset.
`PrintWriter(OutputStream out)` Creates a new PrintWriter, without automatic line flushing, from an existing OutputStream.
`PrintWriter(OutputStream out, boolean autoFlush)` Creates a new PrintWriter from an existing OutputStream.
`PrintWriter(String fileName)` Creates a new PrintWriter, without automatic line flushing, with the specified file name.
`PrintWriter(String fileName, String csn)` Creates a new PrintWriter, without automatic line flushing, with the specified file name and charset.
`PrintWriter(Writer out)` Creates a new PrintWriter, without automatic line flushing.
`PrintWriter(Writer out, boolean autoFlush)` Creates a new PrintWriter.

Java - pliki

- Jeden z konstruktorów przyjmuje zmienną typu string, która wskazuje plik do zapisu.
- Metodami, które umożliwiają zapis do pliku są m.in. write oraz println.

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;
public class Class1 {

    public static void main(String[] args) throws FileNotFoundException {
        PrintWriter writer = new PrintWriter("d:\\lukasz\\plik_nowy.txt");
        writer.write("Lukasz");
        writer.println("Lukasz nowa linia");
        writer.close();
    }
}
```

Java - pliki

Methods

Modifier and Type	Method and Description
`PrintWriter`	`append(char c)` Appends the specified character to this writer.
`PrintWriter`	`append(CharSequence csq)` Appends the specified character sequence to this writer.
`PrintWriter`	`append(CharSequence csq, int start, int end)` Appends a subsequence of the specified character sequence to this writer.
`boolean`	`checkError()` Flushes the stream if it's not closed and checks its error state.
`protected void`	`clearError()` Clears the error state of this stream.
`void`	`close()` Closes the stream and releases any system resources associated with it.
`void`	`flush()` Flushes the stream.
`PrintWriter`	`format(Locale l, String format, Object... args)` Writes a formatted string to this writer using the specified format string and arguments.
`PrintWriter`	`format(String format, Object... args)` Writes a formatted string to this writer using the specified format string and arguments.
`void`	`print(boolean b)` Prints a boolean value.
`void`	`print(char c)` Prints a character.
`void`	`print(char[] s)` Prints an array of characters.
`void`	`print(double d)` Prints a double-precision floating-point number.

Java - pliki

void	**print**(float f) Prints a floating-point number.
void	**print**(int i) Prints an integer.
void	**print**(long l) Prints a long integer.
void	**print**(**Object** obj) Prints an object.
void	**print**(**String** s) Prints a string.
PrintWriter	**printf**(**Locale** l, **String** format, **Object**... args) A convenience method to write a formatted string to this writer using the specified format string and arguments.
PrintWriter	**printf**(**String** format, **Object**... args) A convenience method to write a formatted string to this writer using the specified format string and arguments.
void	**println**() Terminates the current line by writing the line separator string.
void	**println**(boolean x) Prints a boolean value and then terminates the line.
void	**println**(char x) Prints a character and then terminates the line.
void	**println**(char[] x) Prints an array of characters and then terminates the line.
void	**println**(double x) Prints a double-precision floating-point number and then terminates the line.
void	**println**(float x) Prints a floating-point number and then terminates the line.

Java - pliki

void	println(int x) Prints an integer and then terminates the line.
void	println(long x) Prints a long integer and then terminates the line.
void	println(Object x) Prints an Object and then terminates the line.
void	println(String x) Prints a String and then terminates the line.
protected void	setError() Indicates that an error has occurred.
void	write(char[] buf) Writes an array of characters.
void	write(char[] buf, int off, int len) Writes A Portion of an array of characters.
void	write(int c) Writes a single character.
void	write(String s) Writes a string.
void	write(String s, int off, int len) Writes a portion of a string.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Java - obiektowość

„Programowanie obiektowe jest obecnie najbardziej rozpowszechnionym paradygmatem programowania i zastąpiło techniki proceduralne opracowane jeszcze w latach 70. Java jest językiem w pełni obiektowym, a co za tym idzie – aby być efektywnym programistą Javy, trzeba znać techniki obiektowe.”

Java - obiektowość

- W programowaniu strukturalnym, buduje się zbiór procedur do rozwiązania konkretnego problemu, a dopiero później przychodzi czas na zapisanie danych.
- W programowaniu obiektowym najpierw są dane, a dopiero później algorytmy, które te dane przetwarzają.

Java - klasa

- **Klasa** jest szablonem, z którego tworzy się obiekty.
- „Jeżeli klasy są foremkami do robienia ciastek, to obiekty są samymi ciastkami.”
- **Konstruując obiekt**, tworzy się **instancję (egzemplarz) klasy**.
- Kluczowym elementem związanym z obiektami jest **hermetyzacja**, czyli ukrywanie danych.
- Dane zawarte w obiekcie nazywają się **składowymi obiektu**.
- Procedury operujące na danych w obiekcie nazywają się **metodami**.
- Zestaw składowych obiektu (ich wartości) określają aktualny stan obiektu.

Java - klasa

- W Java klasy można budować wykorzystując rozszerzanie (ang. extending) innych klas.
- Wszystkie klasy w Java dziedziczą po jednej klasie bazowej o nazwie Object.
- W przypadku budowania klasy wykorzystując rozszerzanie, nowo powstała klasa ma wszystkie cechy i metody klasy rozszerzanej.
- Nowe pola i metody dostępne są tylko w nowej klasie.
- Proces rozszerzania klasy w celu utworzenia nowej klasy nazywa się dziedziczeniem (ang. inheritance)

Java - obiekty

- Trzy podstawowe cechy obiektu:

1. Zachowanie obiektu – co można z obiektem zrobić i jakie metody można wywołać na jego rzecz.

- Wszystkie obiekty będące egzemplarzami tej same klasy są do siebie podobne pod tym względem, że charakteryzują się takim samym zachowaniem.
- Zachowanie obiektu definują metody, które można wywoływać.

Java - obiekty

- Trzy podstawowe cechy obiektu:

2. Stan obiektu – jak obiekt reaguje w odpowiedzi na wywoływane na jego rzecz metody.

- Stan obiektu może się zmieniać w czasie, jednak nie może tego robić samoczynnie.
- Zmiana stanu obiektu musi być spowodowana wywołaniem metod.
- Jeżeli stan obiektu uległ zmianie bez wywoływania metod, oznacza to że została złamana metoda hermetyzacji.

Java - obiekty

- Trzy podstawowe cechy obiektu:

3. Tożsamość obiektu – stan obiektu nie pozwala na jednoznaczne zidentyfikowanie obiektu.

- Na przykład w systemie przetwarzania zamówień dwa zamówienia są odrębne mimo tego, że dotyczą zakupu tego samego produktu.
- Poszczególne obiekty będące instancjami tej samej klasy zawsze mają inną tożsamość i zazwyczaj różnią się stanami.

Java - obiekty

- Przedstawione trzy kluczowe cechy obiektów mogą pomiędzy sobą oddziaływać.
- Przykładowo stan obiektu może mieć wpływa na jego zachowanie.
- W systemie zamówień np. internetowych, jeżeli zamówienie zostało wysłane to obiekt może odmówić wykonania metody dodanie lub usunięcia dodatkowych elementów zamówienia.
- W przypadku kiedy zamówienie jest puste tzn. nie zawiera żadnych elementów zamówiena – obiekt nie powinien umożliwić wysyłki takiego zamówienia.

Java – identyfikacja klas

- Nazwy klas zwyczajowo tworzy się z rzeczowników, które są obecne w analizowanym problemie.
- Nazwy metod natomiast odpowiadają czasownikom.
- Przykładowo w systemie obsługującym zamówienia możemy wyróżnić następujące rzeczowniki:
- Item (produkt), Order (zamówienie), Shipping address (Adres dostawy), Payment (płatność), Account (konto)
- Z powyższych rzeczowników można utworzyć następujące klasy: Item, Order, ShippingAddress, Payment itd.

Java – identyfikacja klas

- Kolejnym krokiem jest znalezienie czasowników, czyli czynności które można wykonać np. dodać produkt (add), usunąć produkt (remove), wysałać zamówienie (send), anulować zamówienie (cancel).
- Dla każdego z tych czasowników należy znaleźć obiekt, który jest odpowiedzialny za wykonywanie tych działań.
- Przykładowo do zamówienia dodawany jest produkt to w tą operację należy zaangażować obiekty klasy Order.
- To oznacza również, że metoda add (dodaj element do zamówienia), powinna być metodą klasy Order i przyjmować jako parametr obiekt klasy Item.

Java – relacje pomiędzy klasami

- Najczęściej w Java używa się następujących relacji pomiędzy klasami:

    1. Zależność (używa) – przykładowo klasa Order używa klasy Account, ponieważ obiekty klasy Order potrzebują dostępu do obiektów klasy Account w celu sprawdzenia danych klienta. Natomiast klasa Item nie jest zależna od klasy Account, ponieważ przedmioty nie potrzebują informacji o danych klienta. W związku z tym jedna klasa jest zależna od innej klasy, jeżeli metody tej pierwszą używają obiektów tej drugiej lub na nich operują.

Java – relacje pomiędzy klasami

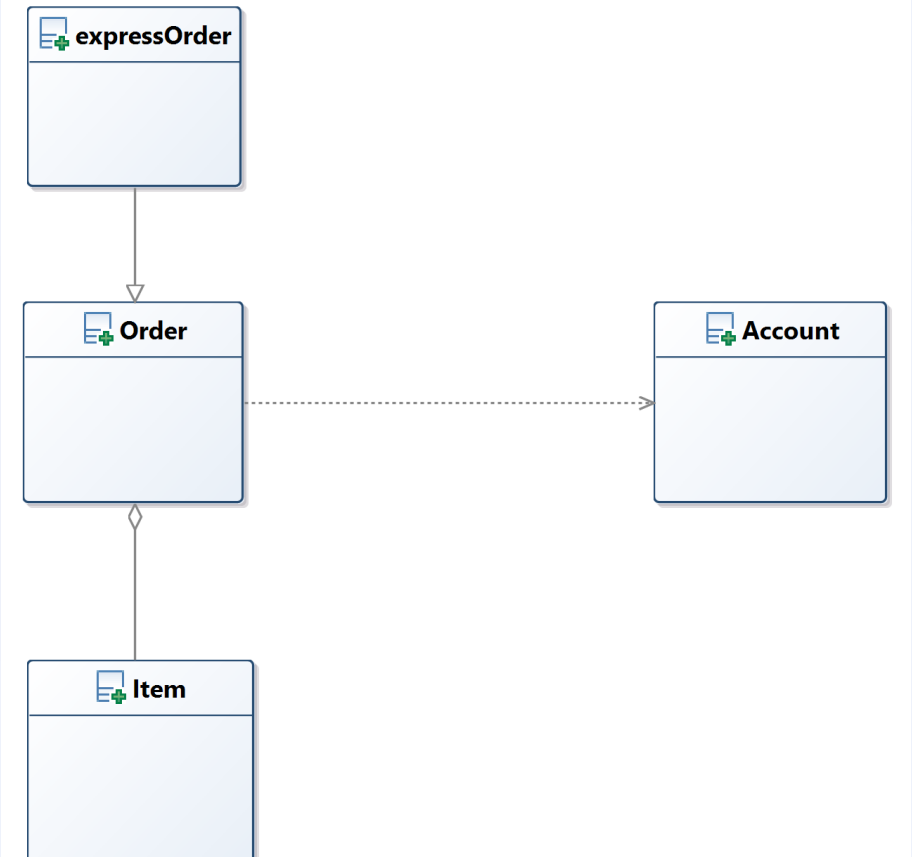
- Najczęsciej w Java używa się następujących relacji pomiędzy klasami:

2. Agregacja (zawiera) – opisuje zawieranie obiektów jednej klasy w obiektach innej klasy. Przykładowo obiekty klasy Order zawierają obiekty klasy Item.

3. Dziedziczenie (jest) – opisuje związek pomiędzy klasą ogólną i klasą specjalną. Na przykład klasa expressOrder dziedziczy po klasie Order. Klasa expressOrder posiada specjalne metody pozwalające obsługiwać ekspresowe zamówienie jednak pozostałe metody takie jak dodawanie i usuwanie produktów są takie same jak w klasie Order.

Java – relacje pomiędzy klasami

W związku z tym, jeżeli klasa A rozszerza klasę B to klasa A ma te same metody co klasa B plus dodatkowo posiada metody rozszerzające jej funkcjonalność (ma większe możliwości od klasy B).



Java – klasy predefiniowane

- W trakcie poprzedniego wykładu zaprezentowana została klasa Math, która umożliwia wykonywanie operacji matematycznych.
- Klasa Math nie wymaga powołania klasy obiektu w celu wykonania operacji matemtycznej np. Math.abs(wartość).
- Oznacza to, że klasa Math hermetyzuje swoja funkcjonalność.
- Klasa Math nie hermetyzuje swoich danych. Dlaczego?
- Ponieważ klasa Math nie posiada żadnych danych!

Java – obiekty i zmienne obiektów

- W celu używania obiektu, należy go na początku utworzyć i określić jego stan początkowy.
- W Java nowe egzemplarze klasy tworzy się za pomocą **konstruktorów.**
- **Konstruktor** jest specjalną metodą, której zadaniem jest utworzenie i inicjalizacja obiektu.
- **Konstruktor** ma zawsze taką samą nazwę jak klasa.
- W związku z tym konstruktor klasy Date (klasa predefiniowana) posiada kontruktor o nazwie Date.
- W celu utworzenia obiektu klasy Date należy użyć konstruktora tej klasy wraz z operatorem **new**.

Java – obiekty i zmienne obiektów

- Utworzenie obiektu klasy Date powoduje, że przypisywana jest do niego obecna data i godzina.
- Obiekt klasy Date można również przekazać do metody System.out.println w celu jego wypisania do konsoli.

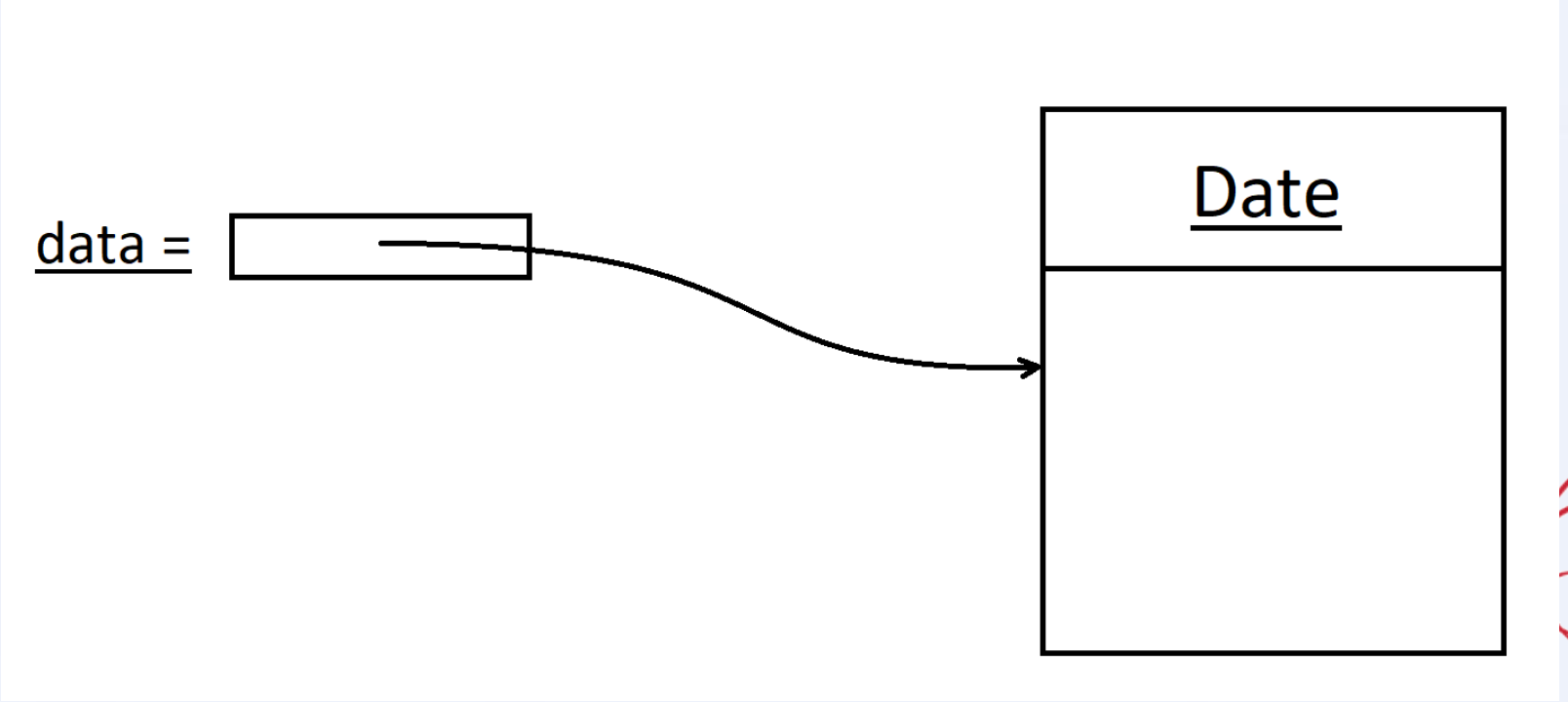
```
import java.util.Date;

public class Class1 {

    public static void main(String[] args){
        Date data = new Date();
        System.out.println(data);
    }
}
```

```
Thu Oct 18 22:35:17 CEST 2018
```

Java – obiekty i zmienne obiektów

- Przypisanie nowo utworzonego obiektu do zmiennej powoduje, że w zmiennej przetrzymywana jest referencja do nowo utworzonego obiektu.

```
Date data = new Date();
```



Java – obiekty i zmienne obiektów

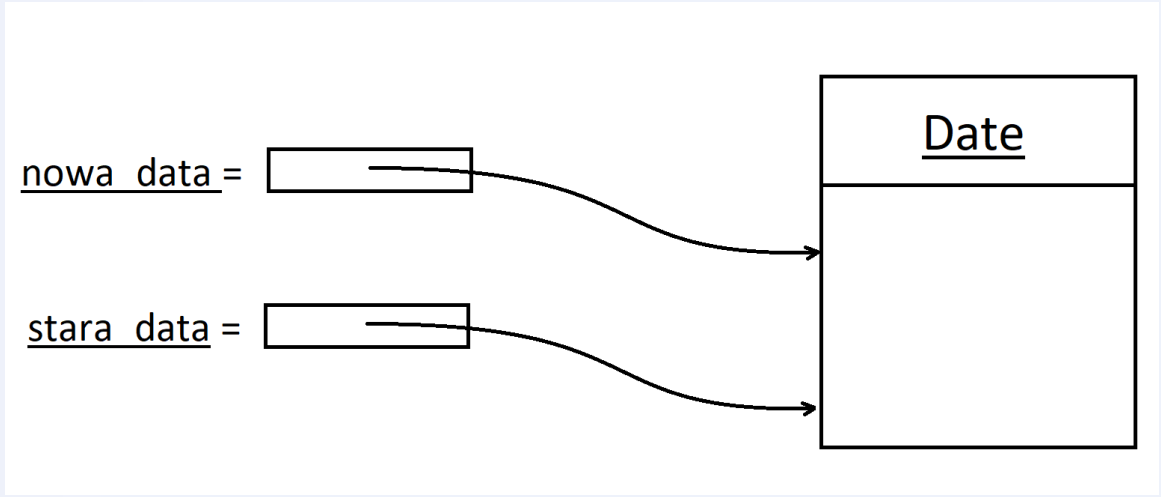
- Pomiędzy obiektem, a zmienną obiektową istnieje bardzo istotna różnica.
- Przykładowo `Date nowa_data;` definiuje zmienną obiektową, która moża odwoływać się do obiektów typu Date. Należy pamiętać, że zmienna nowa_data nie jest obiektem ani nawet nie odwołuje się jeszcze do żadnego obiektu.
- Po takiej definicji nie można nawet na zmiennej nowa_data wywoływać metod klasy Date.
- Zmienną nowa_data można zainicjalizować na dwa sposoby:

```
Date nowa_data;
nowa_data = new Date();
```

```
Date nowa_data;
Date stara_data = new Date();
nowa_data=stara_data;
```

Java – obiekty i zmienne obiektów

- W drugim przypadku obydwie zmienne są referencjami do tego samego obiektu!
- Wywołanie instrukcji `Date stara_data = new Date();` powoduje utworzenie nowego obiektu klasy Date
- Wynikiem jest referencja do tego obiektu, która przypisywana jest do zmiennej stara_data.



Java – obiekty i zmienne obiektów

- Co zostanie wypisane po wykonaniu poniższego kodu?

```
public static void main(String[] args){
    Object obj1 = new Object();
    Object obj2;
    Object obj3 = new Object();
    obj2=obj1;
    System.out.println(obj1.toString());
    System.out.println(obj2.toString());
    System.out.println(obj3.toString());
}
```

Java – obiekty i zmienne obiektów

- Co zostanie wypisane po wykonaniu poniższego kodu?

```
public static void main(String[] args){
    Object obj1 = new Object();
    Object obj2;
    Object obj3 = new Object();
    obj2=obj1;
    System.out.println(obj1.toString());
    System.out.println(obj2.toString());
    System.out.println(obj3.toString());
}
```

```
java.lang.Object@48cf768c
java.lang.Object@48cf768c
java.lang.Object@6a6824be
```

Java – obiekty i zmienne obiektów

- W celu zaznaczenia, że zmienna obiektowa nie odwołuje się do żadnego obiektu, należy jej wartość ustawić na null.

```
public static void main(String[] args){
    Date data;
    data=null;
    if(data!=null)
    {
        System.out.println(data.toString());
    }
}
```

- Odwołanie do zmiennej obiektowej, która ma wartość null spowoduje „rzucenie” wyjątku typu NullPointerException.
- Zmienne obiektowe nie są domyślnie inicjalizowane wartością typu null!

Java – klasa LocalDate

- Przedstawiona wcześniej klasa Date określa liczbę milisekund, które upłynęły od pewnego momentu, zwanego epoką.
- Jest to dokładnie 1 stycznia 1970 roku o godzinie 00:00:00 UTC.
- Klasa Date nie jest jednak użyteczna przy manipulowaniu datami.
- Projektanci Java postanowili więc rozdzielić kwestie zapisywania informacji o czasie od nadawania nazw momentom czasu.
- W tymcelu biblioteka standardowa Javy zawiera dwie klasy: Date oraz LocalDate.
- Klasa LocalDate wyraża daty w znanej wszystkim notacji kalendarzowej.

Java – klasa LocalDate

- Obiektów klasy LocalDate nie tworzy się za pomocą konstruktorów, a **metod fabrycznych**, które wywołują konstruktor za programistę.
- Przykładowo metoda LocalDate.Now() zwraca obecną datę, a metoda of(rok,miesiąc,dzień) pozwala uzyskać konkretną datę.

```
import java.time.LocalDate;
public class Class1 {

    public static void main(String[] args){
        LocalDate data = LocalDate.now();
        System.out.println(data);
        LocalDate nowa_data = LocalDate.of(2010,10,15);
        System.out.println(nowa_data);
    }
}
```

```
2018-10-19
2010-10-15
```

Java – klasa LocalDate

- Powstaje pytanie w jakim celu wykorzystywać klasę LocalDate?
- Przewaga klasy LocalDate nad Date ujawnia się w trakcie wykonywania obliczeń na datach.
- Klasa LocalDate posiada metodę umożliwiająca dodanie konkretnej liczby dni do utworzonego obiektu klasy LocalDate.
- Dodatkowo za pomocą odpowiednich metod po wykonaniu operacji dodawania dni można uzyskać z obiektu LocalDate wszystkie elementy daty tj. rok, miesiąc, dzień.

```
import java.time.LocalDate;
public class Class1 {

    public static void main(String[] args){
        LocalDate data = LocalDate.now();
        data = data.plusDays(1500);
        System.out.println(data.getYear());
        System.out.println(data.getMonth());
    }
}
```

```
2022
NOVEMBER
```

Java – klasa LocalDate

All Methods	Static Methods	Instance Methods	Concrete Methods

Modifier and Type	Method and Description
`Temporal`	**`adjustInto`**`(Temporal temporal)` Adjusts the specified temporal object to have the same date as this object.
`LocalDateTime`	**`atStartOfDay`**`()` Combines this date with the time of midnight to create a `LocalDateTime` at the start of this date.
`ZonedDateTime`	**`atStartOfDay`**`(ZoneId zone)` Returns a zoned date-time from this date at the earliest valid time according to the rules in the time-zone.
`LocalDateTime`	**`atTime`**`(int hour, int minute)` Combines this date with a time to create a `LocalDateTime`.
`LocalDateTime`	**`atTime`**`(int hour, int minute, int second)` Combines this date with a time to create a `LocalDateTime`.
`LocalDateTime`	**`atTime`**`(int hour, int minute, int second, int nanoOfSecond)` Combines this date with a time to create a `LocalDateTime`.
`LocalDateTime`	**`atTime`**`(LocalTime time)` Combines this date with a time to create a `LocalDateTime`.
`OffsetDateTime`	**`atTime`**`(OffsetTime time)` Combines this date with an offset time to create an `OffsetDateTime`.
`int`	**`compareTo`**`(ChronoLocalDate other)` Compares this date to another date.

Java – klasa LocalDate

boolean	**equals**(**Object** obj) Checks if this date is equal to another date.
String	**format**(**DateTimeFormatter** formatter) Formats this date using the specified formatter.
static **LocalDate**	**from**(**TemporalAccessor** temporal) Obtains an instance of LocalDate from a temporal object.
int	**get**(**TemporalField** field) Gets the value of the specified field from this date as an int.
IsoChronology	**getChronology**() Gets the chronology of this date, which is the ISO calendar system.
int	**getDayOfMonth**() Gets the day-of-month field.
DayOfWeek	**getDayOfWeek**() Gets the day-of-week field, which is an enum DayOfWeek.
int	**getDayOfYear**() Gets the day-of-year field.
Era	**getEra**() Gets the era applicable at this date.
long	**getLong**(**TemporalField** field) Gets the value of the specified field from this date as a long.

Java – klasa LocalDate

Month	**getMonth**() Gets the month-of-year field using the Month enum.
int	**getMonthValue**() Gets the month-of-year field from 1 to 12.
int	**getYear**() Gets the year field.
int	**hashCode**() A hash code for this date.
boolean	**isAfter**(**ChronoLocalDate** other) Checks if this date is after the specified date.
boolean	**isBefore**(**ChronoLocalDate** other) Checks if this date is before the specified date.
boolean	**isEqual**(**ChronoLocalDate** other) Checks if this date is equal to the specified date.
boolean	**isLeapYear**() Checks if the year is a leap year, according to the ISO proleptic calendar system rules.
boolean	**isSupported**(**TemporalField** field) Checks if the specified field is supported.
boolean	**isSupported**(**TemporalUnit** unit) Checks if the specified unit is supported.
int	**lengthOfMonth**() Returns the length of the month represented by this date.

Java – klasa LocalDate

int	**lengthOfYear**() Returns the length of the year represented by this date.
LocalDate	**minus**(long amountToSubtract, **TemporalUnit** unit) Returns a copy of this date with the specified amount subtracted.
LocalDate	**minus**(**TemporalAmount** amountToSubtract) Returns a copy of this date with the specified amount subtracted.
LocalDate	**minusDays**(long daysToSubtract) Returns a copy of this LocalDate with the specified number of days subtracted.
LocalDate	**minusMonths**(long monthsToSubtract) Returns a copy of this LocalDate with the specified number of months subtracted.
LocalDate	**minusWeeks**(long weeksToSubtract) Returns a copy of this LocalDate with the specified number of weeks subtracted.
LocalDate	**minusYears**(long yearsToSubtract) Returns a copy of this LocalDate with the specified number of years subtracted.
static **LocalDate**	**now**() Obtains the current date from the system clock in the default time-zone.
static **LocalDate**	**now**(**Clock** clock) Obtains the current date from the specified clock.
static **LocalDate**	**now**(**ZoneId** zone) Obtains the current date from the system clock in the specified time-zone.
static **LocalDate**	**of**(int year, int month, int dayOfMonth) Obtains an instance of LocalDate from a year, month and day.

Java – klasa LocalDate

static **LocalDate**	**of**(int year, **Month** month, int dayOfMonth) Obtains an instance of LocalDate from a year, month and day.
static **LocalDate**	**ofEpochDay**(long epochDay) Obtains an instance of LocalDate from the epoch day count.
static **LocalDate**	**ofYearDay**(int year, int dayOfYear) Obtains an instance of LocalDate from a year and day-of-year.
static **LocalDate**	**parse**(**CharSequence** text) Obtains an instance of LocalDate from a text string such as 2007-12-03.
static **LocalDate**	**parse**(**CharSequence** text, **DateTimeFormatter** formatter) Obtains an instance of LocalDate from a text string using a specific formatter.
LocalDate	**plus**(long amountToAdd, **TemporalUnit** unit) Returns a copy of this date with the specified amount added.
LocalDate	**plus**(**TemporalAmount** amountToAdd) Returns a copy of this date with the specified amount added.
LocalDate	**plusDays**(long daysToAdd) Returns a copy of this LocalDate with the specified number of days added.
LocalDate	**plusMonths**(long monthsToAdd) Returns a copy of this LocalDate with the specified number of months added.
LocalDate	**plusWeeks**(long weeksToAdd) Returns a copy of this LocalDate with the specified number of weeks added.

Java – klasa LocalDate

LocalDate	**plusYears**(long yearsToAdd) Returns a copy of this LocalDate with the specified number of years added.
<R> R	**query**(**TemporalQuery**<R> query) Queries this date using the specified query.
ValueRange	**range**(**TemporalField** field) Gets the range of valid values for the specified field.
long	**toEpochDay**() Converts this date to the Epoch Day.
String	**toString**() Outputs this date as a String, such as 2007-12-03.
Period	**until**(**ChronoLocalDate** endDateExclusive) Calculates the period between this date and another date as a Period.
long	**until**(**Temporal** endExclusive, **TemporalUnit** unit) Calculates the amount of time until another date in terms of the specified unit.
LocalDate	**with**(**TemporalAdjuster** adjuster) Returns an adjusted copy of this date.
LocalDate	**with**(**TemporalField** field, long newValue) Returns a copy of this date with the specified field set to a new value.
LocalDate	**withDayOfMonth**(int dayOfMonth) Returns a copy of this LocalDate with the day-of-month altered.

Java – klasa LocalDate

LocalDate	**withMonth**(int month) Returns a copy of this LocalDate with the month-of-year altered.
LocalDate	**withYear**(int year) Returns a copy of this LocalDate with the year altered.

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.time.chrono.ChronoLocalDate

timeLineOrder

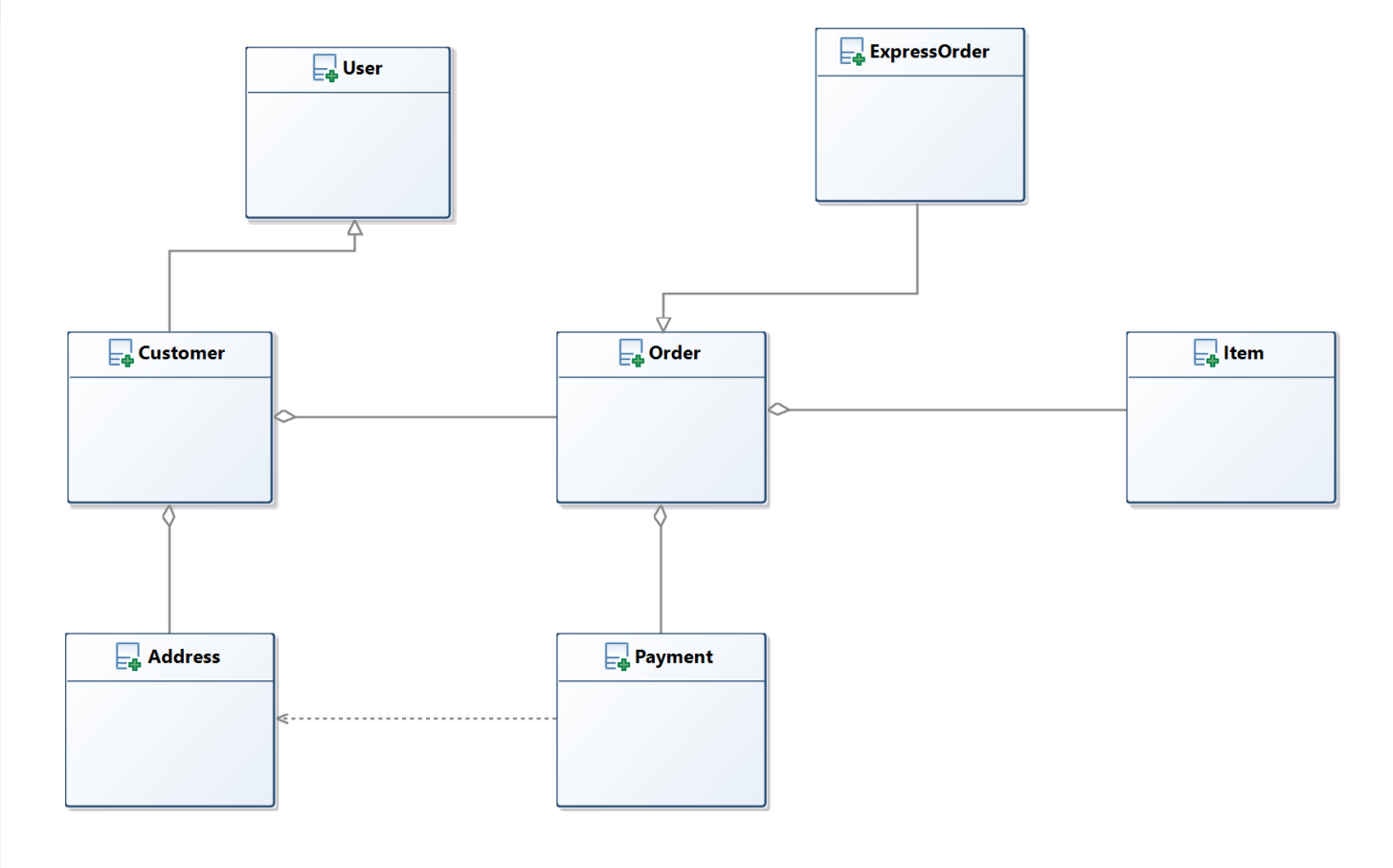
Java – klasy

- Rozwińmy przykład ze sklepem internetowym, obsługującym zamówienia.
- Jakie klasy powinny znaleźć się w rozwiązaniu problemu?

Java – klasy

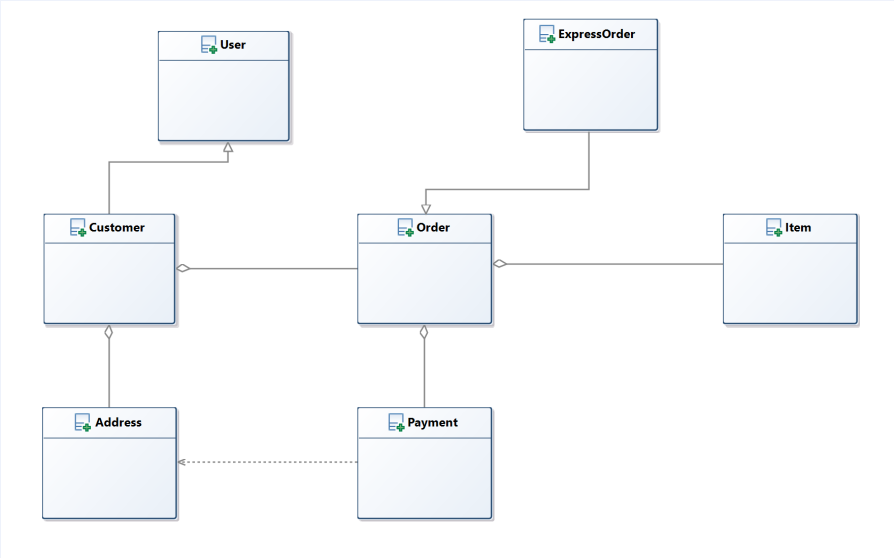
- Klasa Item – reprezentująca przedmiot, który można zakupić
- Klasa Order – reprezentująca zamówienie
- Klasa ExpressOrder – reprezentująca szybkie zamówienie
- Address – reprezentująca adres dostawy
- Payment – reprezentująca płatność
- Customer – reprezentująca klienta w sklepie
- User – reprezentująca użytkownka w systemie (login, hasło itp.)

Java – klasy



Java – klasy

- Jakie atrybuty będzie miała klasa User?
- Jakie atrybuty będzie miała klasa Customer?
- Jakie atrybuty będzie miała klasa Address?
- Jakie atrybuty będzie miała klasa Order?
- Jakie atrybuty będzie miała klasa ExpressOrder?
- Jakie atrybuty będzie miała klasa Payment?
- Jakie atrybuty będzie miała klasa Item?



DZIĘKUJĘ ZA UWAGĘ