

Лекция 6.2

Особености при използването на методи в Java (част 2)

Основни теми

- Представяне на примитивни и референтни данни в компютърната памет.
- Организация на компютърната памет в процеса на програмно изпълнение.
- Предаване на данни между методи- изпълнение на методи и връщане на данни
- Обсег на валидност на декларации на данни
- Приложения- Генериране на случайни стойности
- Допълнително дефиниране на методи (*overloading*)
- Рисуване на запълнени геометрични фигури
- Задачи

Въведение

- 6.6 Машинно представяне локални данни
- 6.7 Преобразуване на типове данни
- 6.8 Генериране на случайни числа- приложения
- 6.9 Обсег на валидност за декларация на данни
- 6.10 Допълнителни дефиниции на методи

Задачи

Литература:

Java How to Program, Sixth Edition, глава 6

6.6 Преговор- Машинно представяне на изпълнението на методи

- Stack структура на паметта
 - Last-in, first-out (LIFO- *последния влязъл- излиза пръв*) структура данни
 - Елементи се вмъкват (push) отгоре
 - Елементи се извеждат (pop) отгоре
- Stack структура за изпълнение на методи
 - Наричана още, **execution stack** - Stack структура с изпълнения на методи, **процесорът има директен достъп до тази структура**
 - Когато един метод извиква друг метод, извиквания метод трябва да знае как да върне управлението на викащия метод. Затова **адресът на викащия метод се записва (push) в execution stack**. При последователни извиквания на методи, адресите на викащите методи се записват като **последния влязъл- излиза пръв структура**.
 - Адресите на викащите методи се вмъкват **execution stack** , когато те викат други методи и те се извеждат от **execution stack** , когато виканият метод връща управлението (*виканият метод взема адреса за връщане на управлението от execution stack, след което този адрес се изтрива от execution stack*)

6.6 Преговор- Машинно представяне на изпълнението на методи

- При извикването на метод, неговите локални променливи се вмъкват в част от **execution stack** известна като запис за активиране – “*activation record*”
- При връщане на управлението на викащия метод, записът за активиране се извежда (pop) от **execution stack** и тези локални променливи са недостъпни за изпълняваната програма.
- Когато **една единствена локална** променлива реферира обект, то при извеждане на тази променлива от **execution stack** този обект става недостъпен за програмата и евентуално се изчиства от паметта посредством “garbage collection.” на JVM

6.6 Примитивни и референтни данни-представяне в паметта

- Всяка компютърна програма трябва да се зареди в паметта, за да се изпълни. За разбиране на **разликите между примитивни и референтни данни** е важно да се разбере как тези данни се представят в паметта.
- ОС и JVM в частност разделят паметта на две области, всяка от които се обработва по специфичен начин.
- Тези области традиционно се наричат **stack** и **heap**.

6.6 Примитивни и референтни данни-представяне в паметта

- При извикване на метод се използва *stack – a* за съхранение на формалните аргументи и останалите локални променливи на метода. Тези локални променливи **могат да са от примитивен или референтен тип**. При завършване на работата си, извиканият метод освобождава автоматично заеманата памет *stack – a* и позволява тази памет да се използва от други викани методи. Съответно формалните аргументи и останалите локални променливи на метода не могат да се използват в останалата част от изпълнението на програмата.

6.6 Примитивни и референтни данни-представяне в паметта

- При **създаване на обект** (инстанция на клас) с използване на ключовата дума **new** и последвана от извикване на съответен **конструктор за инициализация** на обекта се отделя памет от **heap** областта на паметта.
- *Така един и същ **обект може да се реферира** с една или повече референтни променливи разположени в **stack** областта на паметта.*
- Когато и последната локална променлива, реферираща даден обект, освободи заеманата от нея памет в **stack** областта на паметта, то този обект става недостъпен за последващото изпълнение на програмата и става кандидат за освобождаване на заеманите ресурси в **heap** областта на паметта.

6.6 stack и heap

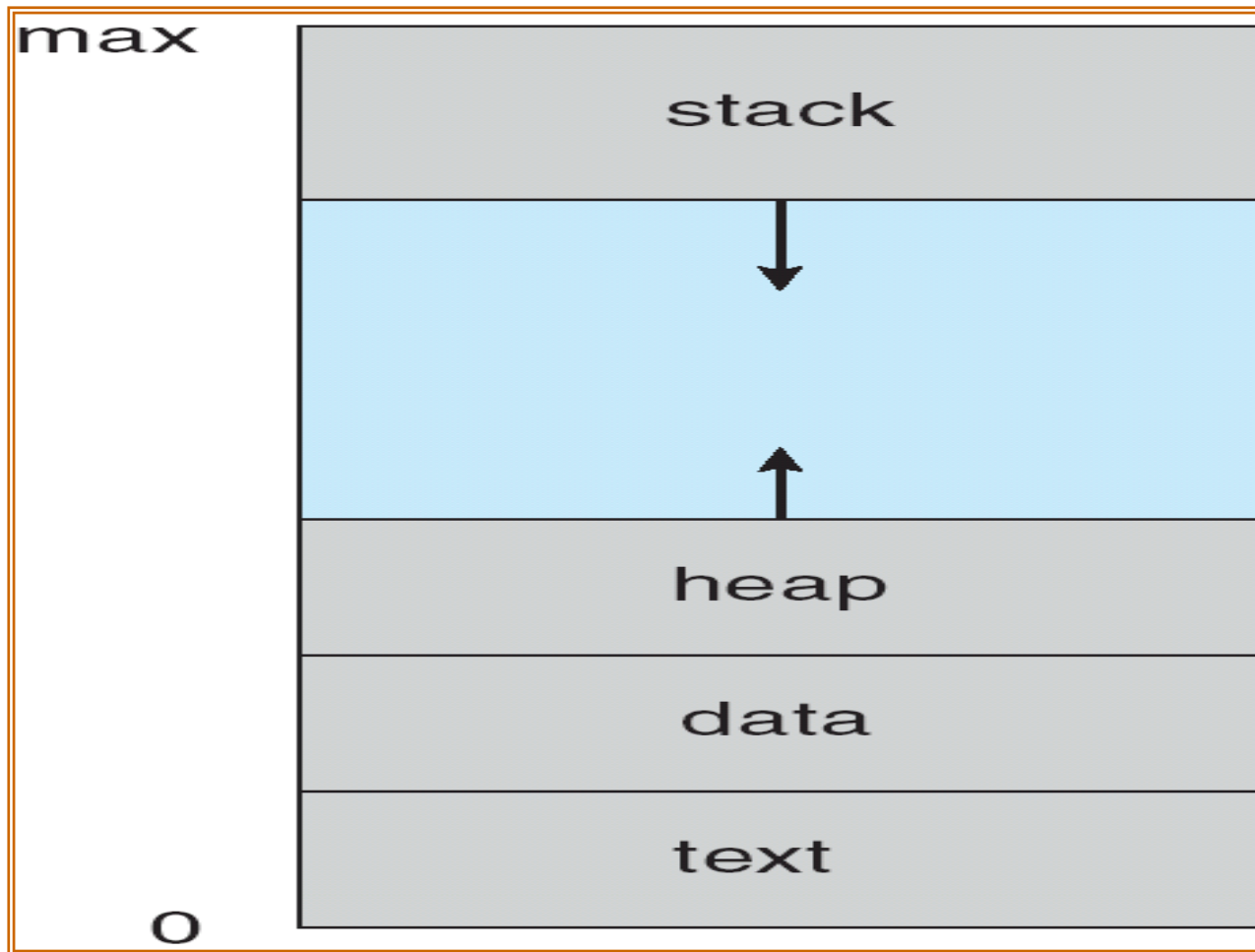
Имената **stack** и **heap** съответстват на организацията на паметта:

- **Stack** паметта е *организирана като поредица от кутии поставени една върху друга*. Когато се извиква метод първо се слага *адреса на викация метод* в кутия най-отгоре на stack-а. След това в отделни кутии върху stack-а се слагат последователно реалните аргументи и локалните променливи на викания метод. **При завършване на викания метод**, всички тези кутии се премахват в обратна последователност на поставянето им “*последен влязъл- пръв излязъл*” **LIFO структура**.

6.6 *stack* и *heap*

- ***Heap*** паметта е като **множество кутии разхвърляни** в стая, вместо да са внимателно подредени една върху друга. Всяка кутия **има етикет**, на който пише дали се **използва или не**. Когато се създава нов обект, JVM търси да намери празна кутия и разполага в нея създадения обект. **Референция** към така създадения обект се съхранява в **някоя от локалните променливи**, разположени някъде в ***stack***-а. JVM следи за броя на референциите към всеки обект (*спомнете си, че 2 и повече променливи могат да сочат към същия обект*). Когато и последната **референция към обекта освободи мястото си в *stack***-а, то JVM маркира кутията на този обект в ***Heap***-а като свободна и след време може отново да се използва за съхранение на друг обект.

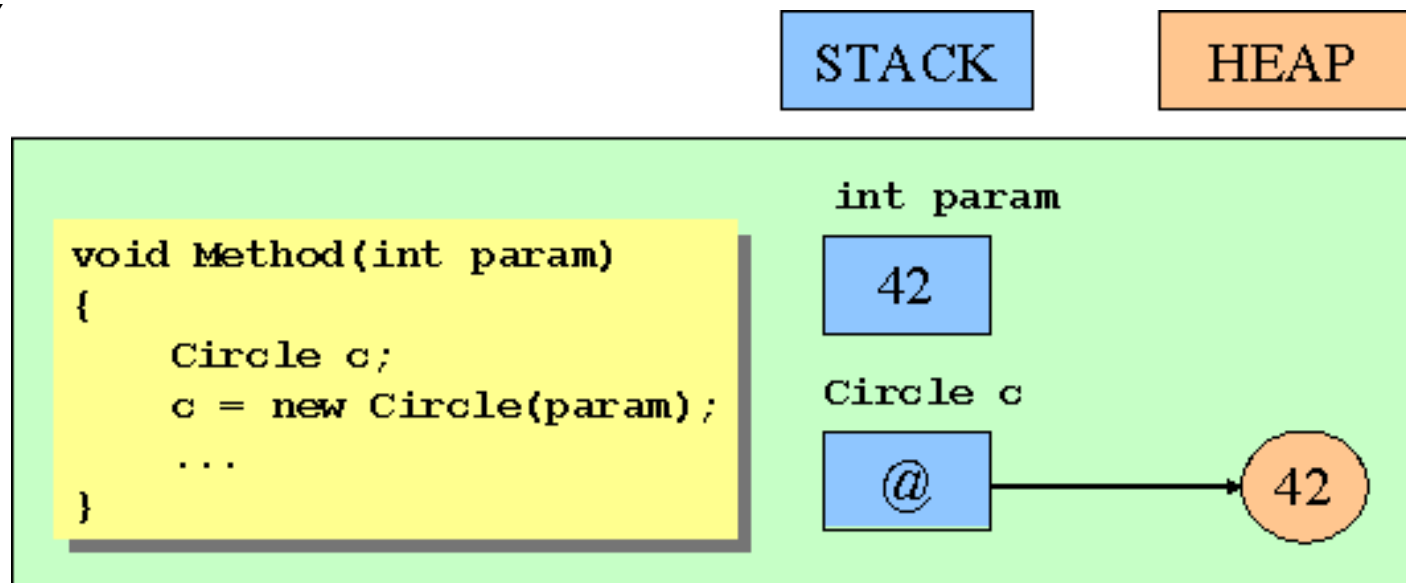
6.6 stack и heap на процес



6.6 stack и heap Примери

```
public void Method(int param)
{    //    викан метод
    Circle c;
    c = new Circle(param);
    ...
}

public void someMethod() //    викащ метод
{
    Method(42);
}
```



6.6 Копиране на примитивни и референтни данни

- Всички примитивни типове данни като *int* се наричат още тип- стойности. При деклариране на *int* променлива, компилаторът отделя *32 бита* достатъчни да съхранят *integer*. Една команда която присвоява стойност (примерно 42) на такава *int* променлива води до копиране на числото 42 в тези 32 бита отделени за тази *int* променлива.

6.6 Копиране на примитивни и референтни данни

- Class (референтните) типове, както примерно Circle, се обработват по различен начин. При деклариране на *Circle* променлива, компилаторът не генерира код за отделяне на блок от паметта достатъчна за един Circle обект. Вместо това се заделя **малка област от паметта за съхранение на адрес на друг блок от паметта съдържащ Circle обект**. Паметта за самият Circle обект се заделя едва при изпълнение създаване на обекта с ключовата дума *new*.
- Това оправдава названието за променливи за примитивните данни-**стойностен тип**- съдържат директно данните си, за разлика от **референтните променливи**, които съдържат референции към своите стойности.

6.6 Указатели и референции

Забележка

Ако сте имали практика с **С** или **С++** програмиране може да се **изкушавате** мислите за **референтните типове като за указатели**. Въпреки, че референтните типове в Java са сходни с указатели, те, референтните типове имат много повече възможности като функционалност.

- Например, в **С** и **С++** приложение може да се създаде **указател към почти произволен блок от паметта**, независимо от типа на данните в този блок. Понякога това е удобно, но много по-често това води до **трудни за откриване грешки в програмирането**.
- В Java и другите ООП езици всички референтни типове данни са явно типизирани (**strongly typed**); не е възможно да се декларира референция към един тип (примерно *Circle*), и после същата референция да се използва за достъп до блок от паметта съдържаща друг тип данни. Съществуват и други различия по това как се **заделя и освобождава паметта**

6.6 Копиране на примитивни и референтни данни- примери

```
int i      = 42; // declare and initialize i
int copyi = i; // copyi contains a copy of the data in i
i++; // incrementing i has no effect on copyi
```

```
int i;
i = 42;
```

```
int copyi;
copyi = i;
```

int i

42

int copyi

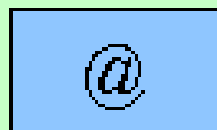
42

6.6 Копиране на примитивни и референтни данни- примери

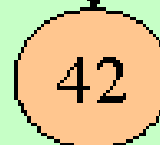
```
Circle c = new Circle(42);  
Circle refc = c;
```

```
Circle c;  
c = new Circle(42);  
  
Circle refc;  
refc = c;
```

Circle c



Circle refc



6.7 Преобразуване на типове данни

- **Преобразуване на типове данни на аргументи**
 - Java се старее да преобразува (ако може) типа на реалните аргументи до типа на данните на формалните аргументи в съответствие с **правилата за неявно преобразуване**
 - Стойностите в един израз се преобразуват до “най-високия” тип в израза (създава се временно копие на тази стойност)
 - Преобразуването до “по-ниски” типове данни води до грешки при компилация, освен ако не е зададено явно преобразуване от при задаване на реалните аргументи
 - пример: `(int) 4.5`

Type	Valid promotions
double	None
float	double
long	float or double
int	long, float or double
char	int, long, float or double
short	int, long, float or double (but not char)
byte	short, int, long, float or double (but not char)
boolean	None (boolean values are not considered to be numbers in Java)

Fig. 6.5 | Неявно преобразуване на примитивни типове данни.

6.8 Генериране на случайни числа-приложения

- Начини за генериране на случайни числа
 - `static` метод `random` на `class Math`
 - връща `double` стойности в интервала
 $0.0 \leq x < 1.0$
 - `class Random` от библиотека `java.util`
 - Може да генерира псевдослучайни числа `boolean, byte, float, double, int, long` използва `Gaussian` разпределение
 - За начална стойност в алгоритъма за генериране се използва текущото време- **променлива начална стойност**
 - Може да се задава **фиксирана начална стойност** на алгоритъма и тогава винаги се генерира една и съща последователност от случайни числа- използва се за сравнения при симулация на процеси

Outline

```

1 // Fig. 6.7: RandomIntegers.java
2 // Shifted and scaled random integers.
3 import java.util.Random; // program uses class Random
4
5 public class RandomIntegers
6 {
7     public static void main( String args[] )
8     {
9         Random randomNumbers = new Random(); // random number generator
10        int face; // stores each random integer generated
11
12        // loop 20 times
13        for ( int counter = 1; counter <= 20; counter++ )
14        {
15            // pick random integer from 1 to 6
16            face = 1 + randomNumbers.nextInt( 6 );
17
18            System.out.printf( "%d ", face ); // display generated value
19
20            // if counter is divisible by 5, start a new line of output
21            if ( counter % 5 == 0 )
22                System.out.println();
23        } // end for
24    } // end main
25 } // end class RandomIntegers

```

Import на class **Random** от **java.util** package

Създава **Random** обект

Генерира произволно
хвърляне на зарче
(random die roll)

RandomIntegers
.java
(1 of 2)



Outline

Randoml ntegers

.j ava

(2 of 2)

Различни можества от резултати
съдържащи числа в интервала 1-6



Outline

```
1 // Fig. 6.8: RollDie.java
2 // Roll a six-sided die 6000 times.
3 import java.util.Random;
4
5 public class RollDie
6 {
7     public static void main( String args[] )
8     {
9         Random randomNumbers = new Random(); // random number generator
10
11         int frequency1 = 0; // maintains count of 1s rolled
12         int frequency2 = 0; // count of 2s rolled
13         int frequency3 = 0; // count of 3s rolled
14         int frequency4 = 0; // count of 4s rolled
15         int frequency5 = 0; // count of 5s rolled
16         int frequency6 = 0; // count of 6s rolled
17     }
```

Import на class **Random** от **java.util** package

RollDie.java

(1 of 2)

Създава **Random** обект

Декларира броячи за
честота на резултатите



Outline

Roll Die.java

```

18 int face; // stores most recently rolled value
19
20 // summarize results of 6000 rolls of a die
21 for ( int roll = 1; roll <= 6000; roll++ )
22 {
23     face = 1 + randomNumbers.nextInt( 6 ); // number f
24
25     // determine roll value 1-6 and increment appropriate counter
26     switch ( face )
27     {
28         case 1:
29             ++frequency1; // increment the 1s counter
30             break;
31         case 2:
32             ++frequency2; // increment the 2s counter
33             break;
34         case 3:
35             ++frequency3; // increment the 3s counter
36             break;
37         case 4:
38             ++frequency4; // increment the 4s counter
39             break;
40         case 5:
41             ++frequency5; // increment the 5s counter
42             break;
43         case 6:
44             ++frequency6; // increment the 6s counter
45             break; // optional at end of switch
46     } // end switch
47 } // end for
48

```

Симулира 6000
хвърляния на
зарче

Генерира едно случайно
хвърляне на зарче

switch базирано преброяване на
честотите от паднали се 1, 2, 3, 4, 5, 6



Outline

RollDie.java

(3 of 3)

Извеждане на преброяванията

```

49      System.out.println( "Face\tFrequency" ); // output headers
50      System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
51          frequency1, frequency2, frequency3, frequency4,
52          frequency5, frequency6 );
53  } // end main
54 } // end class RollDie

```

Face	Frequency
1	982
2	1001
3	1015
4	1005
5	1009
6	988

Face	Frequency
1	1029
2	994
3	1017
4	1007
5	972
6	981



6.8.1 Генериране на случайни числа-приложения с отместване

- Генериране на случайни числа в даден интервал

– Използваме израза

shiftingValue + *differenceBetweenValues* * `randomNumbers.nextInt(scalingFactor)`

където:

- *shiftingValue* е първото число в желания интервал
- *differenceBetweenValues* е разликата между последователните числа от интервала
- *scalingFactor* определя колко числа има в този интервал

Например: да получим случайно число от редицата

2, 5, 8, 11 и 14,

използваме командата

```
number = 2 + 3 * randomNumbers.nextInt( 5 );
```

6.8.2 Генериране на случайни числа-приложения за Testing и Debugging

- За получаване на Random обект, позволяващ **генериране на същата последователност от случайни числа при всяко изпълнение** на програмата, инициализирайте обекта Random с фиксирана начална стойност
 - Например:
`Random randomNumbers =
 new Random(seedValue);`
 - Или използвайте `setSeed` метода:
`randomNumbers.setSeed(seedValue);`
 - Тук `seedValue` трябва да е от тип `long`

6.9 Обсег на валидност за декларация на данни

- **Основно правило**

- Обхватът на валидност на една декларация на данна е най- малкият обхващащ я блок от фигурни скоби

Следствия:

- Обхватът на валидност на аргументите на един метод е тялото на метода
- Обхватът на локална данна е блокът от команди, в които е декларирана
- Обхватът на данни на клас и метод на клас е тялото на дефиниция на този клас

6.9 Обсег на валидност за декларация на данни

- Скриване на декларации на данни
 - Една **данна** на **инстанцията** (текущия обект) се скрива (“**засенчва**”) от **локална променлива** или аргумент на **метод на инстанцията** със същото име
 - **Обхватът на скриването** е в рамките на обхватът на валидност на локалната променлива

Обичайна грешка при програмиране 6.10

Синтактична грешка е **дублиране на декларация** за локална променлива в един и същи метод.

Обичайна грешка при програмиране 6.3

За избягване на синтактични и логически грешки **именувайте по различен начин** данните на класа и локалните променливи.

Outline

Scope.java

(1 of 2)

```
1 // Fig. 6.11: Scope.java
2 // Scope class demonstrates field and local variable scopes.
3
4 public class Scope
5 {
6     // field that is accessible to all methods of this class
7     private int x = 1;
8
9     // method begin creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public void begin()
12    {
13        int x = 5; // method's local variable x shadows field x
14
15        System.out.printf( "local x in method begin is %d\n", x );
16
17        useLocalVariable(); // useLocalVariable has local x
18        useField(); // useField uses class Scope's field x
19        useLocalVariable(); // useLocalVariable reinitializes local x
20        useField(); // class Scope's field x retains its value
21    }
```

Скрива клас
данна **x**

Извежда
стойността на
локалната
променлива **x**



Outline

Scope. java

(2 of 2)

```

22     System.out.printf( "\nlocal x in method begin is %d\n", x );
23 } // end method begin
24
25 // create and initialize local variable x during each call
26 public void useLocalVariable()
27 {
28     int x = 25; // initialized each time useLocalVariable is called
29
30     System.out.printf(
31         "\nlocal x on entering method useLocalVariable is %d\n", x );
32     ++x; // modifies this method's local variable x
33     System.out.printf(
34         "\nlocal x before exiting method useLocalVariable is %d\n", x );
35 } // end method useLocalVariable
36
37 // modify class Scope's field x during each call
38 public void useField()
39 {
40     System.out.printf(
41         "\nfield x on entering method useField is %d\n", x );
42     x *= 10; // modifies class Scope's field x
43     System.out.printf(
44         "\nfield x before exiting method useField is %d\n", x );
45 } // end method useField
46 } // end class Scope

```

Скрива
локалната
променлива
x

Извежда
локалната **x**

Извежда клас
данната **x**



Outline

ScopeTest.java

```
1 // Fig. 6.12: ScopeTest.java
2 // Application to test class Scope.
3
4 public class ScopeTest
5 {
6     // application starting point
7     public static void main( String args[] )
8     {
9         Scope testScope = new Scope();
10        testScope.begin();
11    } // end main
12 } // end class ScopeTest
```

```
local x in method begin is 5
local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in method begin is 5
```



6.10 Допълнителни дефиниции на методи

- Различия между методи и тяхните допълнителни дефиниции (**overloading**)
 - Множество методи със същото име, но различен тип и наредба или брой на аргументите
 - Компиляторът решава коя дефиниция на метод да извика по типа и наредбата на аргументите (signature)
 - Разликите в типа на връщаните данни не позволява да се различат две дефиниции на метод
 - Използване на повече от една дефиниция на метод, различаващи се единствено по типа на връщаните данни води до синтактична грешка

Още за аргументи на методи лекция 3.2

- Списъкът с аргументи на един метод задава неговия “**подпис**” (method signature). Характеризира се с
 - Брой на аргументите
 - Поредност на типовете на аргументите
- Два метода са различни, ако имат
 - Различно име
 - Еднакво име, но различен “подпис”

Например, методите

```
public void test() {}
```

```
public int test() {}
```

са **еднакви** (имат еднакво име и еднакъв “подпис”), а методите

```
public void testA(String name) {}
```

```
public int testA() {}
```

са **различни**- имат еднакво име, **но** имат различен “подпис”

Outline

MethodOverload
.java

Правилно извиква метод за
“квадрат на цяло
число”

Правилно извиква метод
за
“квадрат на double”

Дефинира “квадрат на
цяло число”

Дефинира “квадрат на
double”

```

1 // Fig. 6.13: MethodOverload.java
2 // Overloaded method declarations.
3
4 public class MethodOverload
5 {
6     // test overloaded square methods
7     public void testOverloadedMethods()
8     {
9         System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
10        System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
11    } // end method testOverloadedMethods
12
13    // square method with int argument
14    public int square( int intValue )
15    {
16        System.out.printf( "\nCalled square with int argument: %d\n",
17                           intValue );
18        return intValue * intValue;
19    } // end method square with int argument
20
21    // square method with double argument
22    public double square( double doubleValue )
23    {
24        System.out.printf( "\nCalled square with double argument: %f\n",
25                           doubleValue );
26        return doubleValue * doubleValue;
27    } // end method square with double argument
28 } // end class MethodOverload
  
```



Outline

MethodOverloadTest
.java

```
1 // Fig. 6.14: MethodOverloadTest.java
2 // Application to test class MethodOverload.
3
4 public class MethodOverloadTest
5 {
6     public static void main( String args[] )
7     {
8         MethodOverload methodOverload = new MethodOverload();
9         methodOverload.testOverloadedMethods();
10    } // end main
11 } // end class MethodOverloadTest
```

Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000



Outline

MethodOverload

Error.java

```

1 // Fig. 6.15: MethodOverloadError.java
2 // Overloaded methods with identical signatures
3 // cause compilation errors, even if return types are different.
4
5 public class MethodOverloadError
6 {
7     // declaration of method square with int argument
8     public int square( int x )
9     {
10         return x * x;
11     }
12
13     // second declaration of method square with int argument
14     // causes compilation error even though return types are different
15     public double square( int y )
16     {
17         return y * y;
18     }
19 } // end class MethodOverloadError
  
```

Грешка: метод със
същия “подпис”

MethodOverloadError.java: 15: square(int) is already defined in
MethodOverloadError
public double square(int y)
 ^

1 error

Грешка при компилация



6.13 GUI и Graphics : Colors и запълнени фигури

- **Color class** от библиотека `java.awt`
 - Представя се като **RGB (red, green and blue)** стойност
 - Всяка от компонентите за цвят взема стойност в интервала **0 до 255**
 - Има **13** предефинирани **static Color** обекта:
 - `Color.Black`, `Color.BLUE`, `Color.CYAN`,
`Color.DARK_GRAY`, `Color.GRAY`, `Color.GREEN`,
`Color.LIGHT_GRAY`, `Color.MAGENTA`, `Color.ORANGE`,
`Color.PINK`, `Color.RED`, `Color.WHITE` и
`Color.YELLOW`

6.13 GUI и Graphics : Colors и запълнени фигури

- **fillRect** и **fillOval** методи на **class Graphics**
 - Аналогични на **drawRect** и **drawOval** но чертаят запълнени фигури с избрания цвят
 - Първите два аргумента задават координатите та горния ляв ъгъл , а вторите два определят дължината и височината н описващия ги правоъгълник
- **setColor** метод на **class Graphics**
 - Задава текущия цвят за рисуване и запълване на фигури

Outline

DrawSmiley.java

```
1 // Fig. 6.16: DrawSmiley.java
2 // Demonstrates filled shapes.
3 import java.awt.Color;
4 import java.awt.Graphics;
5 import javax.swing.JPanel;
6
7 public class DrawSmiley extends JPanel
8 {
9     public void paintComponent( Graphics g )
10    {
11        super.paintComponent( g );
12
13        // draw the face
14        g.setColor( Color.YELLOW );
15        g.fillRect( 10, 10, 200, 200 );
16
17        // draw the eyes
18        g.setColor( Color.BLACK );
19        g.fillRect( 55, 65, 30, 30 );
20        g.fillRect( 135, 65, 30, 30 );
21
22        // draw the mouth
23        g.fillRect( 50, 110, 120, 60 );
24
25        // "touch up" the mouth into a smile
26        g.setColor( Color.YELLOW );
27        g.fillRect( 50, 110, 120, 30 );
28        g.fillRect( 50, 120, 120, 40 );
29    } // end method paintComponent
30 } // end class DrawSmiley
```

Импорт на class Color

Задава цвят за запълване

Рисува запълнени фигури



Outline

DrawSmileyTest

.java

```
1 // Fig. 6.17: DrawSmileyTest.java
2 // Test application that displays a smiley face.
3 import javax.swing.JFrame;
4
5 public class DrawSmileyTest
6 {
7     public static void main( String args[] )
8     {
9         DrawSmiley panel = new DrawSmiley();
10        JFrame application = new JFrame();
11
12        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13        application.add( panel );
14        application.setSize( 230, 250 );
15        application.setVisible( true );
16    } // end main
17 } // end class DrawSmileyTest
```



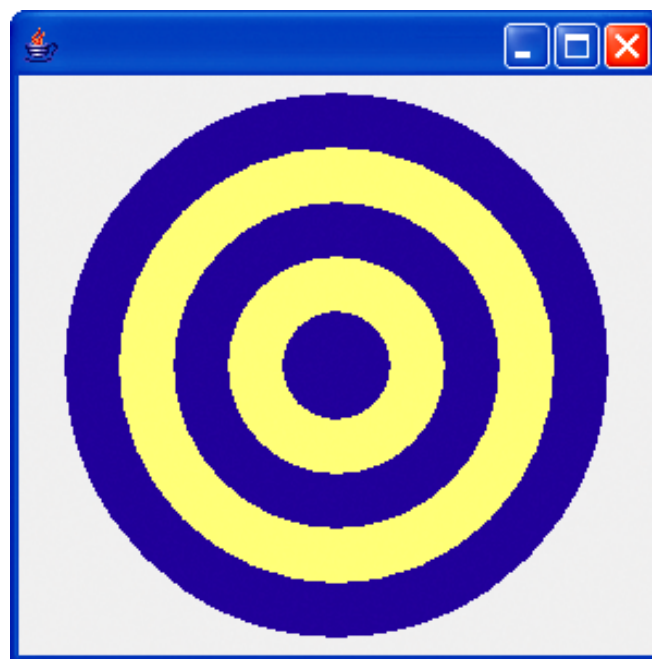


Fig. 6.18 | Концентрични кръгове с алтерниращи цветове.

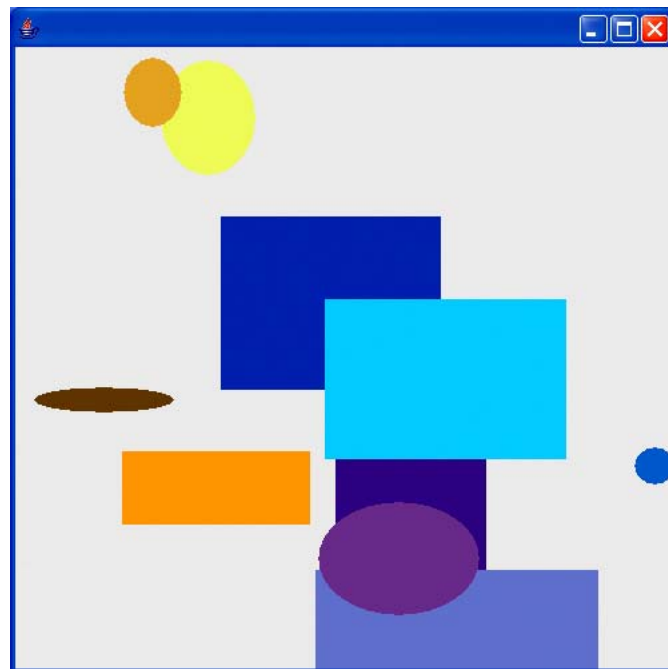


Fig. 6.19 | Произволно зададени фигури.

6.14 Идентифициране на клас методи

- **Идентифициране на клас методи**
 - Изследвайте фразите, изразяващи действие в описанието на изискванията на задачата
- **Моделиране на методите с UML**
 - На всяка операция се задава име, списък от аргументи и тип на връщаните данни:
 - *operationName (parameter1 , parameter2 , ... , parameterN)*
: *return type*
 - Всеки параметър има име и тип на данни
 - *parameterName : parameterType*

Class	Фрази изразяващи действие
ATM	executes financial transactions
Balance inquiry	[none in the requirements document]
Withdrawal	[none in the requirements document]
Deposit	[none in the requirements document]
BankDatabase	authenticates a user, retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
Account	retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
Screen	displays a message to the user
Keypad	receives numeric input from the user
CashDispenser	dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request
DepositSlot	receives a deposit envelope

Fig. 6.20 | Фрази изразяващи действие при програмно симулиране на АТМ система.



Fig. 6.21 | Класове на АТМ с данни и методи.

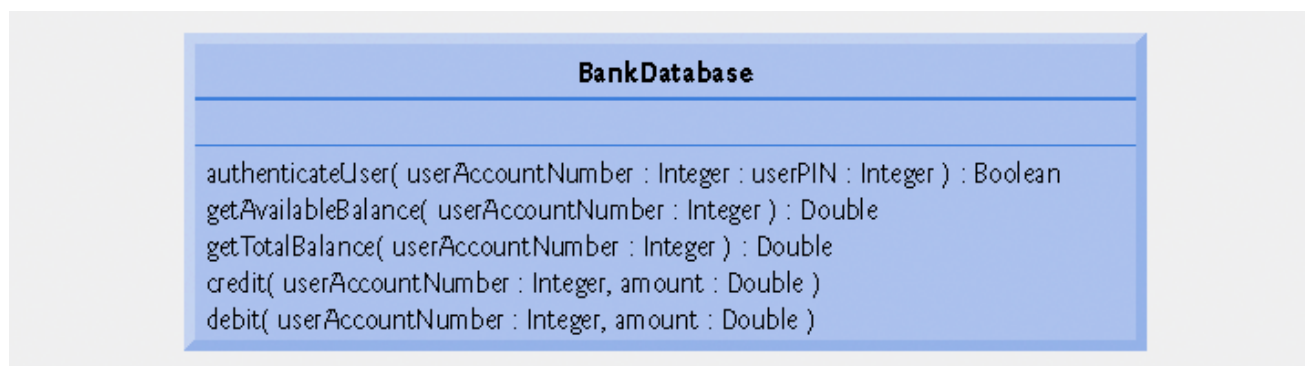


Fig. 6.22 | Class BankDatabase с аргументи на методи.

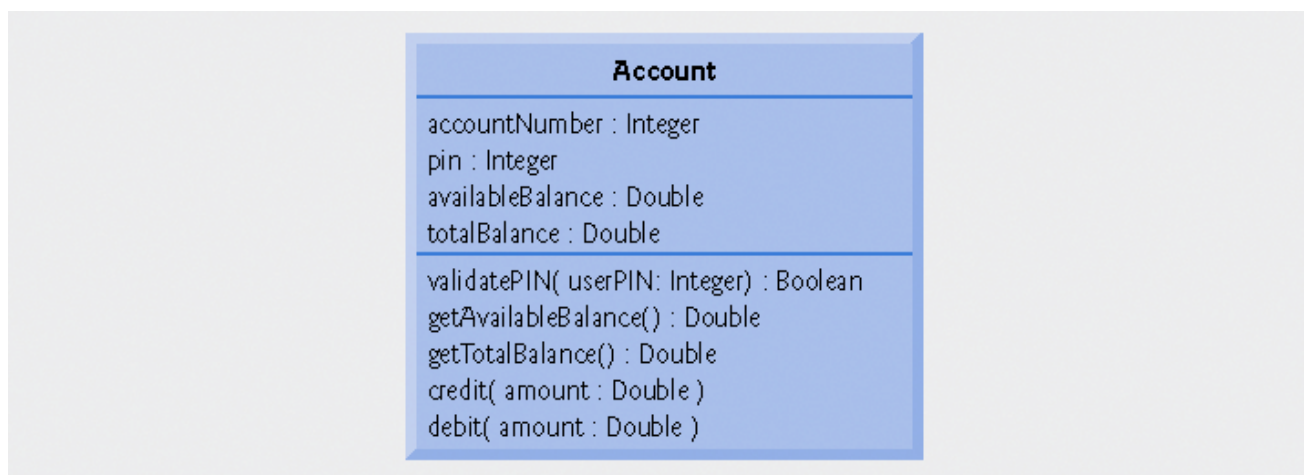


Fig. 6.23 | Class Account с аргументи на методи.

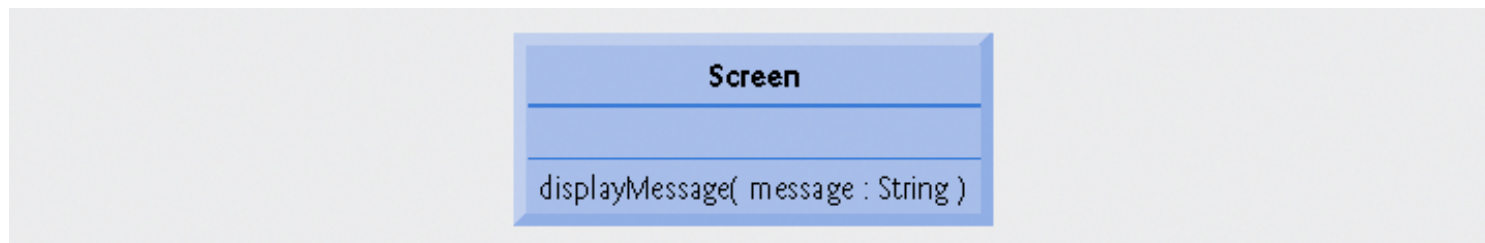


Fig. 6.24 | Class Screen с аргументи на методи.

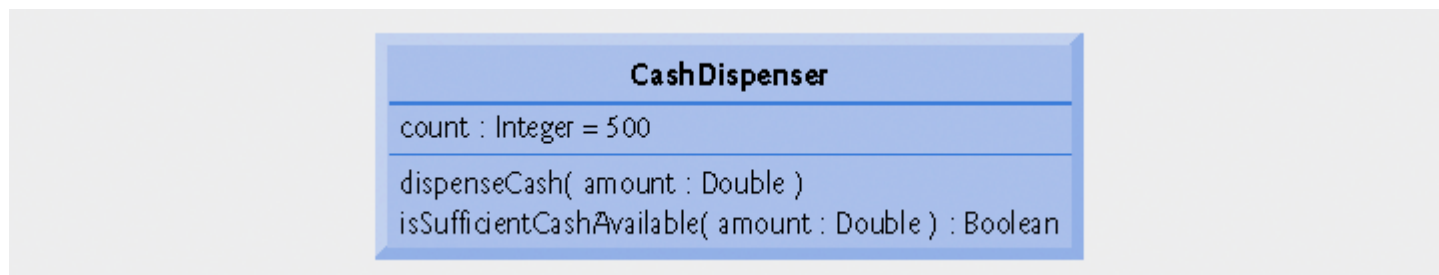


Fig. 6.25 | Class CashDispenser с аргументи на методи.

Задачи

1. В редица случаи се налага използване на меню, за представяне на различни възможности за избор при изпълнение на програма.

Да предположим, че имате за задача да позволите на потребителя да въведе сума за депозирание като число в плаваща запетая и той да може да пресметне еквивалента на тази сума в друга валута, например в евро.

Нека потребителят прави своя избор, посредством следното меню в текстов формат

Въведи сума за депозирание в лева

Изведи депозираната сума в евро

Край

Напишете алгоритъма, UML визуализацията за метод, който изобразява това меню, позволява на потребителя да въведе номер на желана опция и я изпълнява, докато не се избере опция 3. След изпълнение на всяка опция да се изчиства текстовия екран преди ново извеждане на менюто. Използвайте константа за курса за преобразуване от лева в евро

Напишете Java приложение, което реализира този алгоритъм

Задачи

2. Едно цяло число се нарича *перфектно*, ако се представя като сума на множителите си. Например, 6 е перфектно число, понеже,
- $$6 = 1 + 2 + 3.$$

Напишете метод

```
public boolean isPerfect( int number)
```

който определя дали зададения аргумент `number` е перфектно число или не е перфектно число като съответно връща *true* или *false*.

Нека са дефинирани *константите* `NUMBER_PER_LINE = 5` и `NUMBER_OF_PRIMES = 10000`

Напишете още метод

```
public void showMultipliers( int number)
```

който извежда табулирано в редове от по `NUMBER_PER_LINE` числа множителите на зададения аргумент `number`.

Напишете алгоритмите и тяхната UML визуализация за тези два метода

Използвайте тези методи в Java приложение, което определя и извежда всички перфектни числа между 1 и `NUMBER_OF_PRIMES`. За всяко *перфектно* число открито с `isPerfect()` изведете за проверка и съответните му множители, чрез `showMultipliers()`

Задачи

3. Едно цяло число се нарича просто, ако се дели само на 1 и на себе си. Например, 2, 3, 5 и 7 са прости числа, но 4, 6, 8 и 9 не са прости числа.

Напишете метод

```
public boolean isPrime(int number)
```

който определя дали зададения аргумент е просто число или не е просто число и съответно връща true или false.

Използвайте този метод в Java приложение за да намерите и изведете всички прости числа по-малки от `nLimit = 10 000`, където `nLimit` е константа, дефинирана в Java приложението. Колко е горната граница за числата по-малки от 10,000, които трябва да тествате?

Първоначално може да предположите, че търсената горната граница е $nLimit / 2$. В действителност тази горна граница е корен квадратен от `nLimit`. Защо? Изпълнете програмата с всяка от тези две горни граници и сравнете бързодействието и крайните резултати.

Задачи

4. Най- големият общ делител на две числа (*the greatest common divisor* GCD) е най- голямото цяло число, което дели без остатък двете числа.

Напишете метод *gcd()* който връща най- големият общ делител на две числа.

Напишете алгоритъма и UML визуализацията за метод *gcd()*

Използвайте този метод в Java приложение , което прочита две цели числа и извежда резултата от изпълнението на метод *gcd()* заедно с двете въведени числа (за проверка) .

Задачи

4. Най- големият общ делител на две числа $n1$ и $n2$ (*the greatest common divisor* GCD) е най- голямото цяло число, което дели без остатък двете числа по следния начин.

Първо определете d като минимума на $n1$ и $n2$, а след това проверете дали в следния ред някое от числата $d, d-1, d-2, \dots, 2, 1$ е делител едновременно на $n1$ и $n2$. Първото такова число е най- големият общ делител на двете числа $n1$ и $n2$.

Напишете алгоритъма и UML визуализацията му за така описания способ за пресмятане на най- големият общ делител на две числа $n1$ и $n2$

Напишете метод `gcd()` който прилага този алгоритъм.

Използвайте този метод в Java приложение , което прочита две цели числа и извежда резултата от изпълнението на метод `gcd()` заедно с двете въведени числа (за проверка).