

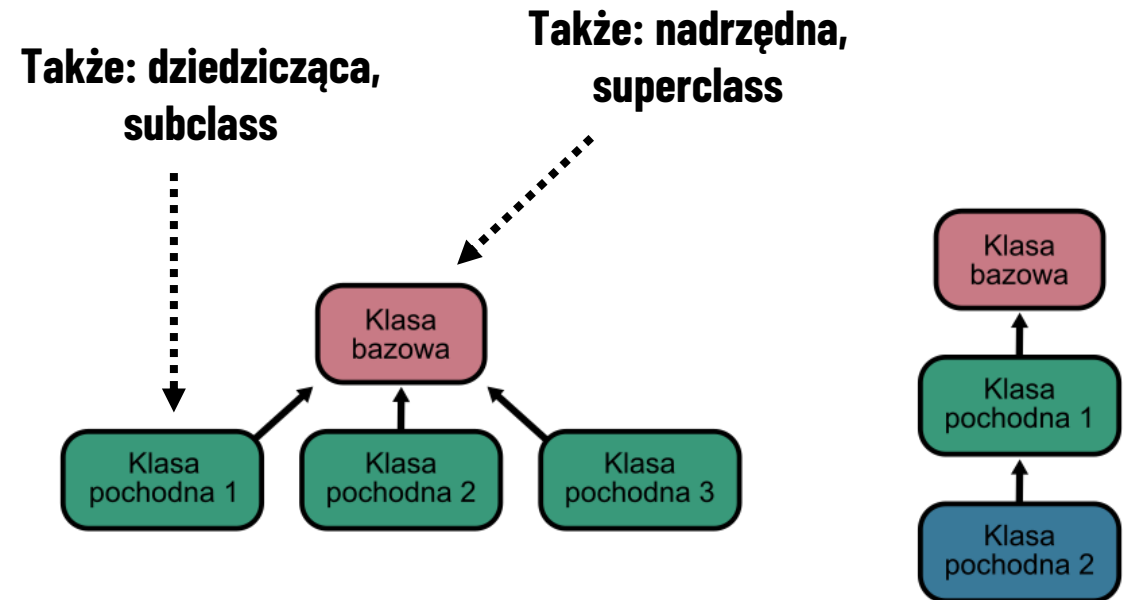
# DZIEDZICZENIE I ABSTRAKCJA

- Dziedziczenie
- Elementy UML
- Klasy abstrakcyjne
- Interfejsy

**Wykład częściowo oparty na materiałach A. Jaskot**

# DZIEDZICZENIE (ang. Inheritance)

- relacja typu “jest” (ang. IS-A)
- wyrażana przez słowo kluczowe *extends*
- wykorzystywana w sytuacji, gdy nowa klasa jest szczególnym rodzajem istniejącej klasy
- jest jednym ze sposobów powtórnego wykorzystania kodu



**Java wspiera dziedziczenie wielopoziomowe,  
ale nie wspiera dziedziczenia wielokrotnego  
(ang. multiple inheritance)**

# DZIEDZICZENIE (ang. Inheritance)

```
package lecture;

public class Animal {
    String type = "XXX";
    public String name;
    protected int age;
    private String ID;

    public Animal() {}

    public Animal(String name, int age, String ID) { // some code }

    public String getID() { // some code }

    public void setID(String ID) { // some code }

    protected void setAge(int age) { // some code }

    private void setID() { // some code }

    protected String makeSound() { // some code }

    public void eat() { // some code }
}
```

```
package exercises;

public class Dog extends Animal {
}
```

**Jakie pola i metody odziedziczy  
klasa Dog?**

# DZIEDZICZENIE (ang. Inheritance)

```
package lecture;

public class Animal {
    String type = "XXX";
    public String name;
    protected int age;
    private String ID;

    public Animal() {}

    public Animal(String name, int age, String ID) { // some code }

    public String getID() { // some code }

    public void setID(String ID) { // some code }

    protected void setAge(int age) { // some code }

    private void setID() { // some code }

    protected String makeSound() { // some code }

    public void eat() { // some code }
}
```

Co jest dziedziczone?

- wszystkie składowe nieprywatne

Co nie jest dziedziczone?

- składowe prywatne
- konstruktory

Jakich klas nie można rozszerzać?

- finalnych

# KONSTRUKTORY W KLASIE POCHODNEJ

```
public class Animal {  
    protected String name;  
    protected int age;  
  
    public Animal(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

```
public class Dog extends Animal {  
    private String breed;  
  
    public Dog(String name, int age, String breed) {  
        super(name, age);  
        this.breed = breed;  
    }  
}
```

## SŁOWO KLUCZOWE SUPER()

- służy do wywoływania konstruktora klasy nadrzędnej
- może być używane niejawnie, jeżeli klasa nadrzędna ma dostępny konstruktor bezparametrowy

```
public class Animal() {  
}
```

```
public class Animal {  
    public Animal() {  
    }  
}
```

```
public class Dog extends Animal {  
    private String breed;  
  
    public Dog(String breed) {  
        this.breed = breed;  
    }  
}
```

# DOSTĘP DO PÓL KLASY NADRZĘDNEJ

```
public class Animal {  
    protected String name;  
    protected int age;  
  
    public Animal(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

```
public class Dog extends Animal {  
    private String breed;  
  
    public Dog(String name, int age, String breed) {  
        super(name, age);  
        this.breed = breed;  
    }  
  
    public String getInfo() {  
        return "Name: " + name  
            + " Age: " + age + " Breed: " + breed;  
    }  
}
```

```
Dog d = new Dog("Fluffy", 5, "German shepherd");  
System.out.println(d.getInfo());
```



```
Name: Fluffy Age: 5 Breed: German shepherd
```

# DOSTĘP DO PÓL KLASY NADRZĘDNEJ

```
public class Animal {  
    private String name;  
    private int age;  
  
    public Animal(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

```
public class Dog extends Animal {  
    private String breed;  
  
    public Dog(String name, int age, String breed) {  
        super(name, age);  
        this.breed = breed;  
    }  
  
    public String getInfo() {  
        return "Name: " + name .....  
        + " Age: " + age + " Breed: " + breed;  
    }  
}
```

'name' has private access in 'Animal'

# DOSTĘP DO PÓL KLASY NADRZĘDNEJ

```
public class Animal {  
    private String name;  
    private int age;  
  
    public Animal(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

```
public class Dog extends Animal {  
    private String breed;  
  
    public Dog(String name, int age, String breed) {  
        super(name, age);  
        this.breed = breed;  
    }  
  
    public String getInfo() {  
        return "Name: " + getName()  
            + " Age: " + getAge() + " Breed: " + breed;  
    }  
}
```



# DOSTĘP DO PÓL KLASY NADRZĘDNEJ

```
public class Animal {  
    protected String name = "Animal";  
    protected int age;  
  
    public Animal(int age) {  
        this.age = age;  
    }  
}
```

```
public class Dog extends Animal {  
    private String name;  
  
    public Dog(String name, int age) {  
        super(age);  
        this.name = name;  
    }  
  
    public String getInfo() {  
        return "Name: " + name  
            + " Alternative name: " + super.name;  
    }  
}
```

Jeżeli klasa nadrzędna i klasa pochodna mają pola o tej samej nazwie:

- słowo kluczowe *this* lub brak słowa kluczowego przed nazwą zmiennej wskazują na pole klasy pochodnej
- słowo kluczowe *super* wskazuje na pole klasy nadrzędnej

```
Dog d = new Dog("Fluffy", 5);  
System.out.println(d.getInfo());
```



```
Name: Fluffy Alternative name: Animal
```

# PRZESŁANIANIE METOD (ang. Method overriding)

```
public class Animal {  
  
    protected String makeSound() {  
        return "Generic animal sound.";  
    }  
}
```

```
public class Dog extends Animal {  
  
    @Override  
    public String makeSound() {  
        return "Woof! Woof!";  
    }  
  
    public String makeMoreGenericSound() {  
        return super.makeSound();  
    }  
}
```

## PRZESŁANIANIE METOD

- jeżeli klasa pochodna “nadpisuje” metodę z klasy nadrzędnej, powinna mieć adnotację *@Override*
- modyfikator dostępu metody przesłaniającej nie może być bardziej restrykcyjny niż bazowej
- metody prywatne i finalne nie mogą być przesłaniane
- słowo kluczowe *super* wskazuje na bazową metodę

```
Dog d = new Dog();  
System.out.println(d.makeSound());  
System.out.println(d.makeMoreGenericSound());
```



```
Woof! Woof!  
Generic animal sound.
```

# SŁOWO KLUCZOWE FINAL

## Zmienne finalne

- ich wartość nie może być zmieniona po zainicjalizowaniu  
np. stałe do obliczeń

```
final int x;  
x = 10;
```

**Deklaracja,  
potem inicjalizacja**

```
final int x = 10;
```

**Deklaracja + inicjalizacja**

```
final int x;  
x = 10;
```

```
x = 5;
```

Variable 'x' might already have been assigned to

[Defer assignment to 'x' using temp variable](#) Alt+Shift+Enter [More actions...](#) Alt+Enter


```
final int x
```

Examples

# SŁOWO KLUCZOWE FINAL

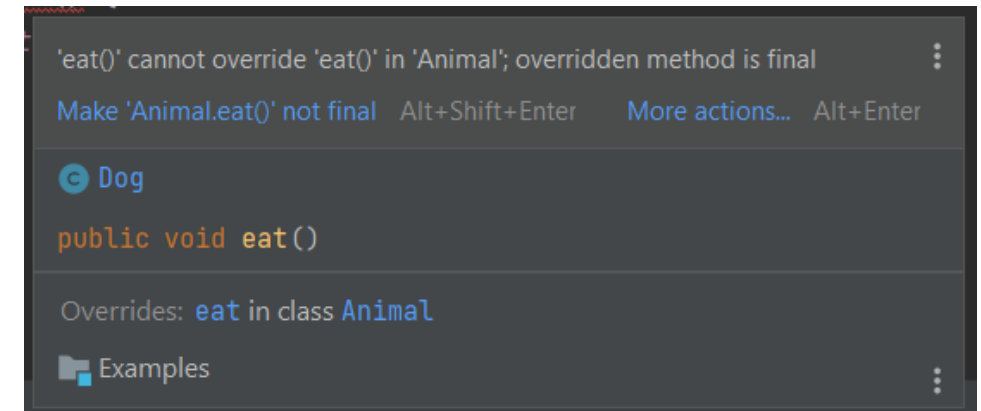
```
class Animal {  
    protected String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public final void eat() {  
        System.out.println(name + " is eating");  
    }  
}
```

```
class Dog extends Animal {  
  
    public Dog(String name) {  
        super(name);  
    }  
  
    @Override  
    public void eat() {  
        System.out.println(this.name + " is eating dog food");  
    }  
}
```



## Metody finalne

- nie mogą być przesłonięte  
np. metody widoczne “na zewnątrz”



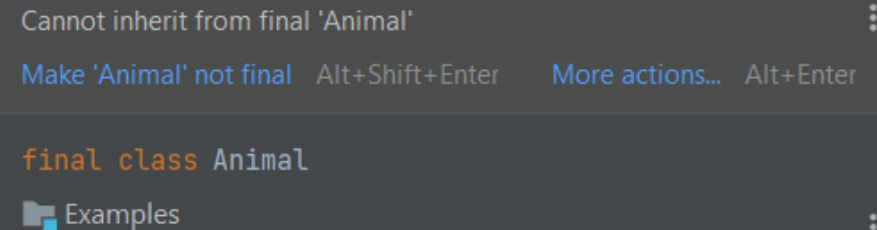
# SŁOWO KLUCZOWE FINAL

```
final class Animal {  
    protected String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
}
```

```
class Dog extends Animal {  
  
    public Dog(String name) {  
        super(name);  
    }  
}
```

## Klasy finalne

- nie mogą być rozszerzone poprzez dziedziczenie
- np. klasy “użytkowe” jak Math



```
Cannot inherit from final 'Animal'  
Make 'Animal' not final Alt+Shift+Enter More actions... Alt+Enter  
final class Animal  
Examples
```

# KLASA OBJECT

- każda klasa w Javie dziedziczy po klasie *Object* bezpośrednio lub pośrednio
- dziedziczenie po klasie *Object* jest niejawne
- klasa *Object* definiuje podstawowe metody, które muszą posiadać wszystkie obiekty

Methods	
Modifier and Type	Method and Description
protected Object	<b>clone()</b> Creates and returns a copy of this object.
boolean	<b>equals(Object obj)</b> Indicates whether some other object is "equal to" this one.
protected void	<b>finalize()</b> Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	<b>getClass()</b> Returns the runtime class of this Object.
int	<b>hashCode()</b> Returns a hash code value for the object.
void	<b>notify()</b> Wakes up a single thread that is waiting on this object's monitor.
void	<b>notifyAll()</b> Wakes up all threads that are waiting on this object's monitor.
String	<b>toString()</b> Returns a string representation of the object.
void	<b>wait()</b> Causes the current thread to wait until another thread invokes the <b>notify()</b> method or the <b>notifyAll()</b> method for this object.
void	<b>wait(long timeout)</b> Causes the current thread to wait until either another thread invokes the <b>notify()</b> method or the <b>notifyAll()</b> method for this object, or a specified amount of time has elapsed.
void	<b>wait(long timeout, int nanos)</b> Causes the current thread to wait until another thread invokes the <b>notify()</b> method or the <b>notifyAll()</b> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

# METODA EQUALS()

- służy do porównywania obiektów typów referencyjnych
- można ją przesłonić, żeby uzyskać własną metodę porównującą dwa obiekty

Methods	
Modifier and Type	Method and Description
protected Object	<code>clone()</code> Creates and returns a copy of this object.
boolean	<code>equals(Object obj)</code> Indicates whether some other object is "equal to" this one.
protected void	<code>finalize()</code> Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	<code>getClass()</code> Returns the runtime class of this Object.
int	<code>hashCode()</code> Returns a hash code value for the object.
void	<code>notify()</code> Wakes up a single thread that is waiting on this object's monitor.
void	<code>notifyAll()</code> Wakes up all threads that are waiting on this object's monitor.
String	<code>toString()</code> Returns a string representation of the object.
void	<code>wait()</code> Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object.
void	<code>wait(long timeout)</code> Causes the current thread to wait until either another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or a specified amount of time has elapsed.
void	<code>wait(long timeout, int nanos)</code> Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

# METODA EQUALS()

## Porównywanie typów prymitywnych

```
int x = 1;
int y = 1;
System.out.println
    ("Int comparison: " + (x == y));

int z = 0;
System.out.println
    ("Int comparison: " + (x == z));
```



```
Int comparison: true
Int comparison: false
```

## Porównywanie typów referencyjnych

```
Person p1 = new Person();
Person p2 = new Person();
System.out.println
    ("Person class comparison 1: " + (p1 == p2));

Person p3 = new Person();
Person p4 = p3;
System.out.println
    ("Person class comparison 2: " + (p3 == p4));
```



```
Person class comparison 1: false
Person class comparison 2: true
```



# METODA EQUALS()

```
public class Student {  
    String lastName;  
    String firstName;  
    int ID;  
  
    public Student(String lastName, String firstName, int ID) {  
        this.lastName = lastName;  
        this.firstName = firstName;  
        this.ID = ID;  
    }  
}
```

```
Student s1 = new Student("XXX", "YYY", 1234);  
Student s2 = new Student("XXX", "YYY", 1234);  
System.out.println("Student class comparison: " + (s1.equals(s2)));
```



**Domyślna implementacja metody equals()  
porównuje tylko referencje (jak operator ==)  
- w klasach własnych trzeba ją przestąpić**

Student class comparison: false

# METODA EQUALS()

```
@Override
public boolean equals(Object obj) {

    if (this == obj) {
        return true;
    }
    if (obj == null || this.getClass() != obj.getClass()) {
        return false;
    }

    Student other = (Student) obj;
    if (this.lastName == null) { // same for firstName
        if (other.lastName != null) {
            return false;
        }
    } else if (!this.lastName.equals(other.lastName)) {
        return false;
    }
    if (this.ID != other.ID) {
        return false;
    }

    return true;
}
```

**Własna metoda equals() powinna sprawdzać:**

- równość fizyczną (referencje)
- czy przekazany obiekt nie jest nullem
- typ przekazywanego obiektu
- wartości we wszystkich polach

i zwracać true tylko wtedy, gdy wszystkie porównania przejdą pozytywnie

```
Student s1 = new Student("XXX", "YYY", 1234);
Student s2 = new Student("XXX", "YYY", 1234);
System.out.println
("Student class comparison: " + (s1.equals(s2)));
```



Student class comparison: true

# PORÓWNYWANIE ŁAŃCUCHÓW ZNAKÓW

```
String s1 = "xxx";  
String s2 = "xxx";  
System.out.println("String comparison: " + (s1 == s2));
```



```
String comparison: true
```

```
Scanner scanner = new Scanner(System.in);  
String s3 = scanner.nextLine();  
String s4 = scanner.nextLine();  
System.out.println("String comparison with scanner - "  
    + s3 + " vs " + s4 + ": " + (s3 == s4));
```



```
xxx  
xxx  
String comparison with scanner - xxx vs xxx: false
```

```
System.out.println("String comparison with scanner - "  
    + s3 + " vs " + s4 + ": " + (s3.equals(s4)));
```



```
xxx  
xxx  
String comparison with scanner - xxx vs xxx: true
```

# METODA HASHCODE()

- powinna zwracać wartość typu `int` unikalną dla każdego różnego obiektu
- jeżeli metoda `equals()` dla danej pary obiektów zwraca `true`, to metoda `hashCode()` powinna zwrócić dla nich tę samą wartość
- powinna być przesłonięta przy przesłanianiu metody `equals()`

Methods	
Modifier and Type	Method and Description
protected Object	<code>clone()</code> Creates and returns a copy of this object.
boolean	<code>equals(Object obj)</code> Indicates whether some other object is "equal to" this one.
protected void	<code>finalize()</code> Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	<code>getClass()</code> Returns the runtime class of this Object.
int	<b><code>hashCode()</code></b> Returns a hash code value for the object.
void	<code>notify()</code> Wakes up a single thread that is waiting on this object's monitor.
void	<code>notifyAll()</code> Wakes up all threads that are waiting on this object's monitor.
String	<code>toString()</code> Returns a string representation of the object.
void	<code>wait()</code> Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object.
void	<code>wait(long timeout)</code> Causes the current thread to wait until either another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or a specified amount of time has elapsed.
void	<code>wait(long timeout, int nanos)</code> Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

# METODA HASHCODE()

- powinna zwracać wartość typu int unikalną dla każdego różnego obiektu
- jeżeli metoda equals() dla danej pary obiektów zwraca true, to metoda hashCode() powinna zwrócić dla nich tę samą wartość
- powinna być przesłonięta przy przesłanianiu metody equals()

```
@Override
public int hashCode() {
    int result = 17;
    result = 31 * result + ID;
    result = 31 * result + customHashCode(firstName);
    result = 31 * result + customHashCode(lastName);
    return result;
}

private int customHashCode(String str) {
    int hash = 0;
    for (char c : str.toCharArray()) {
        hash = 31 * hash + (int) c;
    }
    return hash;
}
```

# METODA toString()

- powinna zwracać reprezentację tekstową obiektu
- jest wywoływana domyślnie przez metody drukujące do konsoli (print, println, printf)

```
Student s1 = new Student("XXX", "YYY", 1234);  
System.out.println("Student s1: " + s1);
```



```
Student s1: Student@44f5e0
```

Methods	
Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
<b>String</b>	<b>toString()</b> Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

# METODA toString()

- powinna zwracać reprezentację tekstową obiektu
- jest wywoływana domyślnie przez metody drukujące do konsoli (print, println, printf)

```
@Override  
public String toString() {  
    return this.lastName + ", " + this.firstName  
        + " (" + this.ID + ")";  
}
```

```
Student s1 = new Student("XXX", "YYY", 1234);  
System.out.println("Student s1: " + s1);
```



```
Student s1: XXX, YYY (1234)
```

# METODA GETCLASS() I OPERATOR INSTANCEOF

Methods	
Modifier and Type	Method and Description
protected Object	<code>clone()</code> Creates and returns a copy of this object.
boolean	<code>equals(Object obj)</code> Indicates whether some other object is "equal to" this one.
protected void	<code>finalize()</code> Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
<b>Class&lt;?&gt;</b>	<b><code>getClass()</code></b> Returns the runtime class of this Object.
int	<code>hashCode()</code> Returns a hash code value for the object.
void	<code>notify()</code> Wakes up a single thread that is waiting on this object's monitor.
void	<code>notifyAll()</code> Wakes up all threads that are waiting on this object's monitor.
String	<code>toString()</code> Returns a string representation of the object.
void	<code>wait()</code> Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object.
void	<code>wait(long timeout)</code> Causes the current thread to wait until either another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or a specified amount of time has elapsed.
void	<code>wait(long timeout, int nanos)</code> Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.



# METODA GETCLASS() I OPERATOR INSTANCEOF


```
Animal a = new Animal();
System.out.println(a.getClass());

Animal d = new Dog();
System.out.println(d.getClass());

if (a.getClass() == d.getClass()) {
    System.out.println("Same class.");
} else {
    System.out.println("Different class.");
}


if (d instanceof Dog) {
    System.out.println("D is a Dog.");
} else {System.out.println("D is not a Dog."); }

if (d instanceof Animal) {
    System.out.println("D is an Animal.");
} else {System.out.println("D is not a Animal."); }
```




```
class Animal
class Dog
```

Porównanie przez getClass()



```
Different class.
```

Porównanie przez instanceof



```
D is a Dog.
D is an Animal.
```

# SŁOWO KLUCZOWE STATIC

## Zmienne i metody statyczne

- są związane bezpośrednio z klasą, a nie z jej instancjami (obiektami)
- każda instancja klasy ma do nich dostęp
- najczęściej używane w klasach “użytkowych” (ang. utility)

```
public class Program {  
    public static void main(String[] args) {  
        // program code  
    }  
}
```

**Po uruchomieniu programu nie jest dostępna instancja klasy Program – metoda main() jest wywoływana po załadowaniu samej klasy**

# SŁOWO KLUCZOWE STATIC

```
public class Counter {  
    public static int count = 0;  
  
    public void increment() {  
        count += 1;  
    }  
}
```

```
Counter c1 = new Counter();  
Counter c2 = new Counter();  
  
c1.increment();  
System.out.println("Value of c1: " + c1.count);  
System.out.println("Value of c2: " + c2.count);  
  
c2.increment();  
System.out.println("Value of c1: " + c1.count);  
System.out.println("Value of c2: " + c2.count);
```



```
Value of c1: 1  
Value of c2: 1  
Value of c1: 2  
Value of c2: 2
```

## Zmienne statyczne

- mają taką samą wartość dla wszystkich obiektów klasy

# SŁOWO KLUCZOWE STATIC

```
public class Calculator {  
    public static int add(int x, int y) {  
        return x + y;  
    }  
}
```

```
int result = Calculator.add(2, 3);  
System.out.println("Result: " + result);
```



Result: 5

## Metody statyczne

- mogą być wywoływane bez tworzenia obiektu danej klasy
- nie mają dostępu do elementów niestatycznych

# SŁOWO KLUCZOWE STATIC

```
public class Rectangle {  
    double length;  
    double width;  
  
    public Rectangle(double length, double width) {  
        this.length = length;  
        this.width = width;  
    }  
  
    public double calculateArea() {  
        return length * width;  
    }  
  
    public static double calculateArea(double length, double width) {  
        return length * width; // ale nie this.length * this.width  
    }  
}
```

## Metody statyczne

- mogą być wywoływane bez tworzenia obiektu danej klasy
- nie mają dostępu do elementów niestatycznych

```
Rectangle r = new Rectangle(2.2, 3.3);  
System.out.println("Area 1: "  
    + r.calculateArea());  
  
System.out.println("Area 2: "  
    + Rectangle.calculateArea(4.4, 5.0));
```



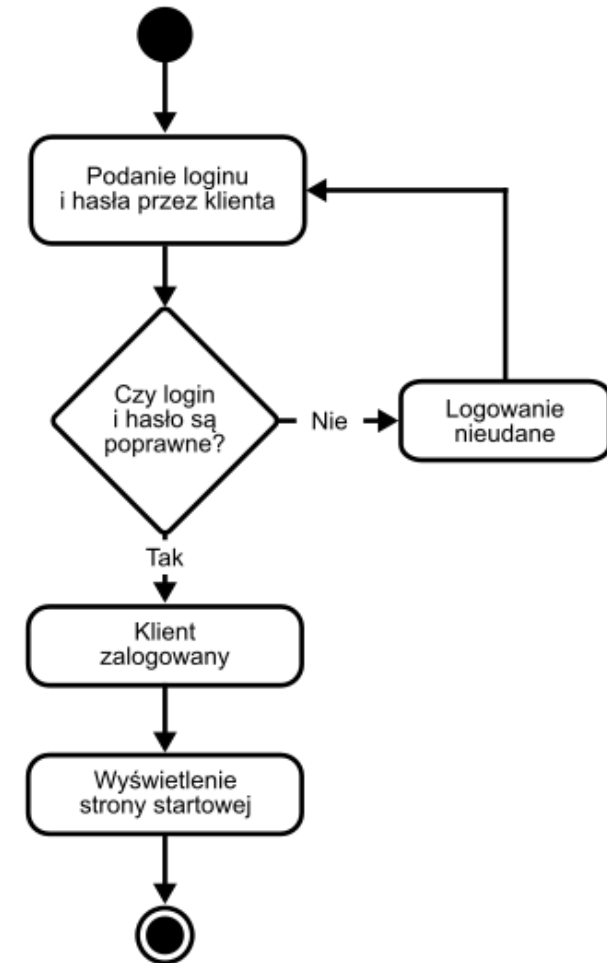
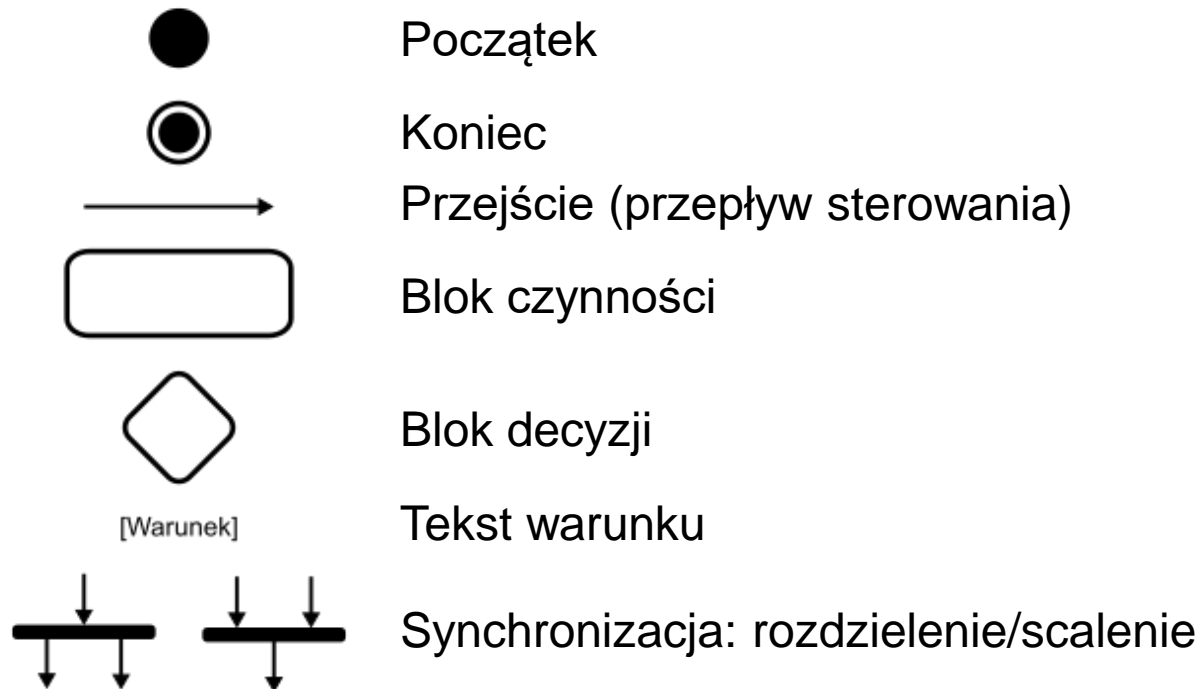
```
Area 1: 7.26  
Area 2: 22.0
```

# UML, z ang. Unified Modeling Language

- graficzny język ogólnego przeznaczenia służący do modelowania systemów
- przystosowany głównie do programowania zorientowanego obiektowo
- opisuje statyczne (diagramy struktury) oraz dynamiczne (diagramy zachowań) elementy systemu

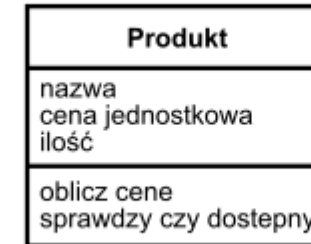
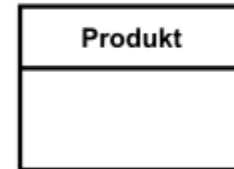
DIAGRAMY STRUKTURY	
<ul style="list-style-type: none"><li>• Diagram klas</li><li>• Diagram obiektów</li><li>• Diagram pakietów</li><li>• Diagram komponentów</li></ul>	<ul style="list-style-type: none"><li>• Diagram struktur połączonych</li><li>• Diagram wdrożeniowy</li></ul>
DIAGRAMY ZACHOWAŃ	
<ul style="list-style-type: none"><li>• Diagram aktywności</li><li>• Diagram maszyny stanowej</li><li>• Diagram przypadków użycia</li></ul>	<ul style="list-style-type: none"><li>• Diagram komunikacji</li><li>• Diagram sekwencji</li><li>• Diagram przebiegów czasowych</li><li>• Diagram stanów</li></ul>

# DIAGRAM CZYNNOŚCI (ang. Activity diagram)

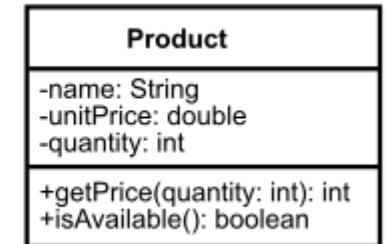


# DIAGRAM KLAS (ang. Class diagram)

- reprezentuje klasy oraz zależności pomiędzy klasami
- na poziomie pojedynczej klasy opisuje:
  - nazwę klasy,
  - pola i metody klasy,
  - poziom dostępu do atrybutów
- na poziomie relacji między klasami opisuje rodzaj zależności (np. dziedziczenie, implementacja, agregacja)



**Koncepcja**



**Implementacja**

Modyfikatory dostępu:

**+** public  
**#** protected  
**-** private



# DIAGRAM KLAS (ang. Class diagram)

