Wyższa Szkoła Informatyki Stosowanej i Zarządzania

pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

Kierunek INFORMATYKA

Studia I stopnia (dyplom inżyniera)

Język Java – wykład 3

dr inż. Łukasz Sosnowski
lukasz.sosnowski@wit.edu.pl
sosnowsl@ibspan.waw.pl
l.sosnowski@dituel.pl

www.lsosnowski.pl

Wyższa Szkoła Informatyki Stosowanej i Zarządzania
pod auspicjami Polskiej Akademii Nauk
WYDZIAŁ INFORMATYKI

Część 1 – Dziedziczenie klas w JAVA

Podstawy dziedziczenia

- Słowo kluczowe ***extends*** definiuje dziedziczenie pomiędzy klasą pochodną a klasą bazową.
- Klasa bazowa to klasa po której dziedziczy inna klasa.
- Klasa pochodna to klasa która dziedziczy po innej klasie bazowej.
- Klasa pochodna może dziedziczyć **tylko** po jednej klasie bazowej.
- Możliwe jest tworzenie hierarchii dziedziczenia, gdzie klasa pochodna staje się bazową dla innej klasy pochodnej.
- Dzięki dziedziczeniu można bardzo precyzyjnie grupować atrybuty wspólne obiektów i dodawać odrębne w klasie pochodnej
- Ogólna postać deklaracji klasy pochodnej:

```
class klasa-pochodna extends klasa-bazowa{
//Ciało klasy
}
```

Dostęp do składowych klasy przy dziedziczeniu

- Klasa pochodna nie ma dostępu bezpośredniego do zmiennych składowych klasy bazowej zadeklarowanych z poziomem dostępu *private.*
- Analogicznie dla metod.
- W przypadku używania zmiennych prywatnych dostęp odbywa się poprzez odpowiednie *settery* oraz *gettery.*
- Poziom dostępu *protected* zapewnia nam swobodny dostęp w klasie pochodnej do zmiennych składowanych zadeklarowanych w klasie bazowej.
- W klasie pochodnej widoczne są również wszystkie zmienne składowe klasy bazowej publiczne oraz jej metody.

Przykład

```
public class Person1 {
     private String firstName = null;
     private String lastName = null;
     private Date dateOfBirth = null;
     public String getFirstName() {
          return firstName;
     }
     public String getLastName() {
          return lastName;
     }
     public Date getDateOfBirth() {
          return dateOfBirth;
     }
     public void setFirstName(
                    String firstName) {
          this.firstName = firstName;
     }
     public void setLastName(
                    String lastName) {
          this.lastName = lastName;
     }
     public void setDateOfBirth(
                    Date dateOfBirth) {
          this.dateOfBirth = dateOfBirth;
     }
}
```

```
public class Employee1 extends Person1 {
     private Date employmentDate = null;
     private BigDecimal sallary = null;
     public Date getEmploymentDate() {
          return employmentDate;
     }
     public void setEmploymentDate(Date employmentDate) {
          this.employmentDate = employmentDate;
     }
     public BigDecimal getSallary() {
          return sallary;
     }
     public void setSallary(BigDecimal sallary) {
          this.sallary = sallary;
     }
}
public class Secretary1 extends Employee1{
     private Set<String> setLanguages = null;
     public Secretary1(String firstName, String lastName,
          Date dateOfBirth, Date employmentDate, BigDecimal sallary) {
          setFirstName(firstName);
          setLastName(lastName);
          setDateOfBirth(dateOfBirth);
          setEmploymentDate(employmentDate);
          setSallary(sallary);
          this.setLanguages = new HashSet<String>();;
     }
     public void addLanguage(String language) {
          setLanguages.add(language);
     }

     public Set<String> getLanguages() {
          return setLanguages;
     }
}
```

Konstruktory vs dziedziczenie

- Klasy bazowe i pochodne mogą posiadać swoje własne konstruktory.
- Za utworzenie obiektu odpowiadają konstruktory odpowiednich klas bazowych i pochodnej. Część obiektu pochodząca z klasy bazowej tworzona jest przez klasę bazową (jej konstruktor).
- Jeśli klasa bazowa posiada jedynie konstruktor domyślny proces jest analogiczny, jednakże wtedy należy zadbać o inicjalizację zmiennych składowych w konstruktorze klasy pochodnej.
- Jeśli klasa bazowa posiada konstruktory parametryczne, należy skorzystać z takiego konstruktora do inicjalizacji tej części obiektu, który definiowany jest przez tę klasę bazową.
- Ogólne zasady tworzenia konstruktorów zostają takie same jak zdefiniowane na poprzednich wykładach.

Słowo kluczowe *super*

- Klasa pochodna może wywołać dowolny konstruktor klasy bazowej z użyciem słowa kluczowego *super* z odpowiednią listą parametrów odpowiadającej sygnaturze konstruktora.
- Instrukcja *super(lista parametrów)* jeśli występuje to musi być pierwszą instrukcja konstruktora, w którym się znajduje.
- Zatem nie można wywołać jawnie wielu konstruktorów klasy bazowej (chyba że poprzez wzajemne zależności w klasie baz.).
- Kolejność wykonywania konstruktorów zgodna z hierarchą dziedziczenia.
- Słowo kluczowe *super* dodatkowo może również służyć do dostępu do składowych i metod klasy bazowej (analogicznie jak *this* w ramach danej klasy lecz w odniesieniu do klasy bazowej).
- Ogólna postać: *super.składowa;, super.metoda();*

Przykład

```
public class Person2 {
     private String firstName = null;
     private String lastName = null;
     private Date dateOfBirth = null;

     public Person2(String firstName,
                    String lastName,
                    Date dateOfBirth) {
          this.firstName = firstName;
          this.lastName = lastName;
          this.dateOfBirth = dateOfBirth;
     }

     public String getFirstName() {
          return firstName;
     }

     public String getLastName() {
          return lastName;
     }

     public Date getDateOfBirth() {
          return dateOfBirth;
     }
}
```

```
public class Employee2 extends Person2{
     private Date employmentDate = null;
     private BigDecimal sallary = null;
     public Employee2(String firstName, String lastName,
                      Date dateOfBirth,Date employmentDate,
                      BigDecimal sallary) {
          super(firstName, lastName, dateOfBirth);
          this.employmentDate = employmentDate;
          this.sallary = sallary;
     }

     public Date getEmploymentDate() {
          return employmentDate;
     }
     public BigDecimal getSallary() {
          return sallary;
     }
}

public class Secretary2 extends Emploee2 {
     private Set<String> setLanguages = null;
     public Secretary2(String firstName, String lastName,
                Date dateOfBirth, Date employmentDate,
                BigDecimal sallary) {
     super(firstName, lastName, dateOfBirth, employmentDate,sallary);
          this.setLanguages = new HashSet<String>();
     }
     public BigDecimal getSallary() {
          return sallary;
     }
     public void setSallary(BigDecimal sallary) {
          this.sallary = sallary;
     }
}
```

Referencje klasy bazowej i obiekty klasy pochodnej

- Zmiennej klasy bazowej można przypisać zmienną klasy pochodnej co stanowi ważny wyjątek w kontroli typów JAVA.
- Należy pamiętać, iż takie przypisanie zezwala jedynie na dostęp do metod i danych z poziomu klasy bazowej.
- Ponowne jawne rzutowanie na zmienną klasy pochodnej przywraca dostęp do metod tej klasy bez utraty danych.
- Przykład:

```
public class A {

      int x;
      public A(int x) {this.x=x;}

}
public class B extends A {
      int y;
      public B(int x, int y) {
            super(x);
            this.y=y;
      }
}
```

```
public void test() {

      A a = new A(9), aa;
      B b = new B(3,4);
      B bb = null;
      aa = a;  //Wszystko OK
      System.out.println("aa.x="+aa.x);
      aa = b;
      System.out.println("aa.x="+aa.x);
      //aa.y //Błąd!
      bb = (B) aa;//Jawne przywrócenie!
      System.out.println("bb.y="+bb.y);
}
```

→

```
aa.x=9
aa.x=3
bb.y=4
```

Część 2 – Polimorfizm, klasy abstrakcyjne

Przesłanianie metod

- W hierarchii dziedziczenia klas, klasa pochodna może mieć metodę o identycznej sygnaturze i zwracanym typie jak klasa bazowa.
- W takim przypadku mówimy o **przesłanianiu metody bazowej.**
- Java używa mechanizmu **dynamicznego wyboru metod,** polegającego na wyborze metody przesłoniętej podczas działania programu a nie podczas kompilacji!!!
- Metoda przesłonięta z klasy bazowej nie jest niewidoczna w klasie pochodnej, aczkolwiek może być jawnie wywołana przy użyciu słowa kluczowego *super* ale z zachowaniem restrykcji związanych z poziomem dostępu do metody.
- Przesłanianie metody jest pewnego rodzaju zastąpieniem jednej implementacji metody drugą (z klasy pochodnej).

Przeciążanie metod i konstruktorów

- Przeciążanie polega na zdefiniowaniu wielu metod o tej samej nazwie lecz z różnymi parametrami (różna liczba i/lub typy i/lub kolejność).
- Przeciążanie może dotyczyć również konstruktorów.
- Przeciążanie (ang. overload) nie jest przesłanianiem (ang. override)!!!
- Przeciążanie metody może zmieniać typ danych zwracany przez metodę.
- W momencie wywołania metody przeciążonej zostaje wybrana wersja, której parametry pasują do argumentów wywołania.
- Sygnatura metody wyznaczana jest na podstawie jej nazwy oraz parametrów (liczebności, typów, kolejności) ale **nie na podstawie zwracanego typu.**

Polimorfizm

- Polimorfizm to wielopostaciowość odnosząca się do metod klas pochodnych.
- Wyróżniamy polimorfizm statyczny i dynamiczny.
- Polimorfizm statyczny realizowany poprzez przeciążanie metod.
- Polimorfizm dynamiczny realizowany poprzez przesłanianie metod.
- Polimorfizm ogranicza złożoność programu poprzez zastosowanie tej „samej idei” do określenia realizacji ogólnej akcji oraz zrealizowaniu jej poprzez wiele postaci tej samej implementacji (np.. dla różnych typów obiektów).
- „Jeden interfejs wiele metod”.

Polimorfizm statyczny i dynamiczny

- Polimorfizm statyczny zachodzi gdy w danej klasie istnieje wiele metod o takiej samej nazwie różniących się deklarowanymi parametrami. Różnica w parametrach definiowana jest poprzez liczbę parametrów, ich typy oraz kolejność. Osiągany jest podczas kompilacji programu.
- Polimorfizm dynamiczny zachodzi wtedy gdy implementacja danej metody bazowej jest przesłaniana przez implementację metody w klasie pochodnej. Ten typ polimorfizmu realizowany jest podczas uruchamiania programu przy użyciu mechanizmu dynamicznego wyboru metod w JAVA. W potocznym użyciu pojęcie polimorfizmu rozumiane jest jako polimorfizm dynamiczny.

Przykład:

```
public class StaticPolymorphism {
     public void print(String msg) {
          System.out.println(msg);
     }

     public void print(int i) {
          System.out.println("i="+i);
     }

     public void print(Integer i) {
          System.out.println("i="+i.toString());
     }

     public void print(double d) {
          System.out.println("d="+d);
     }

     public void print(Double d) {
          System.out.println("d="+d.toString());
     }
}
```

```
@Test
public void StaticTest() {
     StaticPolymorphism sp = null;
     sp = new StaticPolymorphism();
     assertNotNull(sp);
     sp.print("Ala ma kota");
     sp.print(4.5);
     sp.print(2);
     sp.print(Integer.valueOf(5));
     sp.print(Double.valueOf(3.54));
}
```

```
Ala ma kota
d=4.5
i=2
i=5
d=3.54
```

Przykład:

```
public class DynP1 {
	private String message = null;
	public DynamicPolymorphism1(String message) {
		this.message = message;
	}

	public void show() {
		System.out.println("DP1:"+message);
	}

	public String getMessage() {
		return message;
	}
}
public class DynP2 extends DynP1{
	public DynP2(String message) {
		super(message);
	}

	@Override
	public void show() {
		System.out.println("DP2:"+getMessage());
	}
}

public class DynP3 extends DynP2{
	public DynP3(String message) {
		super(message);
	}
	@Override
	public void show() {
		System.out.println("DP3:"+getMessage());
	}
}
```

```
@Test
public void dynamicPolyTest() {
	DynP1 dp1 = null;
	DynP2 dp2 = null;
	DynP3 dp3 = null;
	dp1 = new DynP1("Ala");
	dp2 = new DynP2("Ela");
	dp3 = new DynP3("Ula");
	assertNotNull(dp1);
	assertNotNull(dp2);
	assertNotNull(dp3);
	DynP1 dps[] = new DynP1[]{dp1,dp2,dp3};
	for(DynP1 dp:dps)
		dp.show();
}
```

```
DP1:Ala
DP2:Ela
DP3:Ula
```

Klasa abstrakcyjna

- Klasa w której mogą istnieć metody abstrakcyjne.
- Metoda abstrakcyjna to taka która jest w pełni zadeklarowana ale nie dostarcza implementacji (ciała metody).
- Klasa oraz metoda abstrakcyjna tworzona jest z użyciem słowa kluczowego *abstract.*
- Nie można utworzyć obiektu klasy abstrakcyjnej z użyciem operatora *new.*
- Klasy abstrakcyjne stanowią klasy bazowe. Jeśli klasa pochodna nie dostarcza pełnej implementacji wszystkich metod abstrakcyjnych to również musi być zadeklarowana jako abstrakcyjna.
- Obiekt może być utworzony dopiero jako obiekt klasy zwykłej dostarczającej kompletną implementację wymaganych metod.

Klasa abstrakcyjna c.d.

- Klasa abstrakcyjna wskazuje które metody muszą być przesłonięte w klasach pochodnych.
- Metody abstrakcyjnej można używać wewnątrz klasy, ponieważ jest w pełni zadeklarowana łącznie ze zwracanym typem oraz sygnaturą.
- Metoda abstrakcyjna nie może być metodą statyczną.
- Metoda abstrakcyjna nie może być metodą prywatną.
- Nie można deklarować konstruktorów jako *abstract.*
- Klasa abstrakcyjna stanowi wygodną formę realizacji logiki biznesowej bez podawania szczegółów implementacyjnych często zależnych od różnych wariantów stosowanego typu obiektu lub pewnych złożonych zależności dostępnych w klasach pochodnych.

Przykład

```
public abstract class AbstractDemo {
	protected static final Logger log ...
	private String firstName=null;
	private String lastName=null;

	protected abstract String getPersonType();

	public AbstractDemo(String firstName, String lastName) {
		this.firstName=firstName;
		this.lastName=lastName;
	}

	public void printData() {
		String personType = getPersonType();

		log.info((firstName!=null?firstName:"")
				.concat(" ")
				.concat(lastName!=null?lastName:"")
				.concat(" - ")
				.concat(personType!=null?personType:""));
	}
}

public class StudentDemo extends AbstractDemo {
	public StudentDemo(String firstName, String lastName) {
		super(firstName, lastName);
	}
	@Override
	protected String getPersonType() {
		return "student";
	}
}
```

```
public class TeacherDemo extends AbstractDemo {
	public TeacherDemo(String firstName,
					String lastName) {
		super(firstName, lastName);
	}
	@Override
	protected String getPersonType() {
		return "nauczyciel";
	}
}

@Test
public void test() {
	StudentDemo sd = new
		StudentDemo("Janek","Kowalski");
	sd.printData();

	TeacherDemo td = new
		TeacherDemo("Janek","Nowak");
	td.printData();
}
```

```
Wynik:

Janek Kowalski - student
Janek Nowak - nauczyciel
```

Słowo kluczowe *final*

- Zabrania wykonywania programiście odpowiednich operacji w zależności od elementu dla którego jest definiowane.
- Klasa poprzedzona słowem kluczowym *final* nie może być klasą bazową, a zatem nie można z niej dziedziczyć.
- Metoda poprzedzona słowem kluczowym *final* nie może być przesłonięta w klasie pochodnej.
- Zmienna składowa poprzedzona słowem kluczowym *final* staje się stałą o nadanej nazwie. Wartość takiej „zmiennej” nie może zostać zmieniona podczas działania programu po pierwszej inicjalizacji.
- Zadeklarowanie parametru metody jako *final* zapobiega jego modyfikacji wewnątrz metody.
- Zadeklarowanie zmiennej lokalnej jako *final* zapobiega nadaniu wartości więcej niż raz.

Klasa *Object*

- Domyślna klasa bazowa dla wszystkich klas w JAVA, czyli wszystkie inne klasy są klasami pochodnymi tej klasy.
- Zmiennej referencyjnej lasy Object można przypisywać referencje obiektów dowolnych klas (w tym również tablic typów prostych!!!)
- Wybrane metody klasy Object (a więc wszystkich klas):
  - Object clone() - tworzy nowy obiekt o identycznym stanie
  - boolean equals(Object object) – porównuje obiekty
  - void finalize() - metoda wywoływana przy odzyskiwania pamięci
  - Class<?> getClass() - zwraca klasę obiektu podczas dz. prog.
  - int hashCode() - zwraca wartość funkcji skrótu
  - String toString() - zwraca łańcuch znaków opisujący obiekt

Przykład

```
public final class Parameter {
	private String name = null;
	private Object value = null;
	public Parameter(String name,Object value) {
		this.name=name;
		this.value = value;
	}
	public String getName() {
		return name;
	}
	public Object getValue() {
		return value;
	}
}

public class Parameter2 {
	private String name = null;
	private Object value = null;
	public Parameter2(String name,Object value) {
		this.name=name;
		this.value = value;
	}
	public final String getName() {
		return name;
	}
	public final Object getValue() {
		return value;
	}
}
```

```
public class Parameter3 {
	private final String name;
	private Object value = null;
	public Parameter3(String name,Object value) {
		this.name=name;
		this.value = value;
	}
	public String getName() {
		return name;
	}
	public Object getValue() {
		return value;
	}
	public boolean equals(final String name) {
		return this.name.equals(name);
	}
	public void setValue(Object value) {
		this.value = value;
	}
}
```

Część 3 – Obsługa wyjątków w JAVA

Podstawowe informacje o obsłudze wyjątków

- Obsługa wyjątków wiąże się z następującymi słowami kluczowymi: *try, catch, throw, throws, finally*.
- Instrukcje programu, które mają zostać poddane monitorowaniu wyjątków umieszczone powinny być w bloku *try*.
- W zależności od obsługiwanych wyjątków umieszczamy bloki catch, albo wyłapujące wiele wyjątków do jednej obsługi w danym bloku lub niezależnie do oddzielnych bloków obsługi wyjątku.
- W przypadku nie obsłużenia danego typu wyjątku można zdefiniować przesłanie go przez metodę i obsługę w innym miejscu programu. Takiej definicji dokonujemy przy użyciu klauzuli *throws*.
- JAVA posiada zestaw wbudowanych typów wyjątków stanowiących klasy.

Podstawowe informacje o obsłudze wyjątków c.d.

- Możliwe jest tworzenie własnych klas wyjątków dziedziczących po klasach wbudowanych.
- Podstawowa struktura obsługi wyjątków to:

```
try {

}catch(ArithmeticException | NullPointerException e) {

}
catch(Exception e) {

}
catch(Throwable e) {

}
finally {

}
```

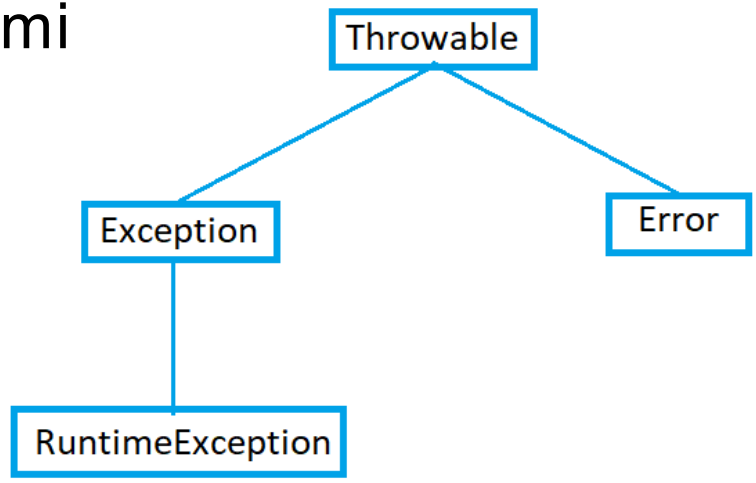
- Wyjątki dodatkowo można generować (rzucać) jawnie z użyciem instrukcji *throw*. Wyjątek można generować w wyniku wykrycia nieprawidłowego stanu obiektu lub poprzez przechwycenie innego wyjątku, jego częściowej obsługi i wygenerowanie innego.

Słowo kluczowe *finally*

- Pozwala na wykonanie bloku kodu po zakończeniu bloku try-catch
- Blok finally wykonuje się niezależnie od tego czy wyjątek został zgłoszony czy też nie
- Wykorzystywany jest do obsługi operacji niezależnych od obsługi wyjątku, np.. zwolnienia przydzielonych zasobów, typu otwarty plik, połączenie do bazy danych, itp.. Obsługa tego rodzaju operacji jest niezbędna w celu poprawnego działania programu (np.. zabezpieczeniem przed wyczerpaniem połączeń do bazy danych).

Typy wyjątków

- Wszystkie wyjątki są klasami pochodnymi klasy Throwable, która jest na szczycie hierarchii dziedziczenia
- Następnie wyjątki dzielą się na 2 kategorie, wywodzące się od klasy
- Exception oraz Error. Pierwsza dotyczy kategorii sytuacji wyjątkowych, druga dotyczy kategorii błędów w samym środowisku JAVA.
- Istnieje klasa pochodna klasy Exception o nazwie RuntimeException. Wyjątki tej klasy definiowane są automatycznie i dotyczą sytuacji typu dzielenia przez zero czy też użycia indeksu tablicy spoza zakresu.



Wyjątki wbudowane w język JAVA

Wyjątek	Opis
ArithmeticException	Błąd arytmetyczny, np.. dzielenie przez zero
ArrayIndexOutOfBoundsException	Indeks tablicy poza dopuszczalnymi wartościami
ArrayStoreException	Przypisanie do tablicy elementu niezgodnego typu
ClassCastException	Błąd rzutowania
EnumConstantNotPresentException	Użycie niezdefiniowanej wartości typu wyliczeniowego
IllegalArgumentException	Błędny argument użyty do wywołania metody
IllegalCallerException	Błąd wykonywania metody przed kod wywołujący
IllegalMonitorStateException	Błędna operacja monitorująca
IllegalStateException	Środowisko lub aplikacja znajduje się w błędnym stanie
IllegalThreadStateException	Żądana operacja nie jest zgodna ze stanem wątku
IndexOutOfBoundsException	Indeks poza dopuszczalnymi granicami

Wyjątki wbudowane w język JAVA c.d.

Wyjątek	Opis
LayerInstantiationException	Nie można utworzyć warstwy modułu
NegativeAraySizeException	Próba utworzenia tablic z ujemnym rozmiarem
NullPointerException	Użycie zmiennej referencyjnej z wartością **null**
NumberFormatException	Błędna konwersja z łańcucha na format liczbowy
SecurityException	Wyjątek bezpieczeństwa
StringIndexOutOfBoundsException	Indeks łańcucha znaków poza dopuszczalnym granicami
TypeNotPresentException	Użycie nie istniejącego typu
UnsupportedOperationException	Nieobsługiwana operacja

Tworzenie własnych klas wyjątków

- Własne klasy wyjątków tworzymy poprzez rozszerzenia klasy Exception
- Nie jest wymagane implementowanie żadnych metod, chociaż jest to możliwe
- Możliwe jest również przesłonięcie wybranych metod
- Wybrane metody klasy Throwable dostępne w klasach pochodnych:
  - *String getMessage()* - zraca opis wyjątku
  - *StackTraceElement[] getStackTrace()* - zwraca tablicę stosu wywołań
  - *Throwable getCause()* - zwraca wyjątek który poprzedził aktualny wyjątek. Jeśli brak zwraca null.

Przykład

```
public class SearchPersonException
                extends Exception{

}
public class SearchEmployeeException
                extends SearchPersonException{

}

public class SearchSecretaryException
                extends SearchEmployeeException{

}

public class SearchProgrammerException
                extends SearchEmployeeException{

}
```

```
public abstract class AbstractExceptionDemo {
      protected static final Logger log =
LogManager.getLogger(AbstractExceptionDemo.class.getName());
      abstract void searchPerson() throws SearchPersonException;
      abstract void searchEmployee() throws
                                          SearchEmployeeException;
      abstract void searchSecretary() throws
                                          SearchSecretaryException;
      abstract void searchProgrammer() throws
                                          SearchProgrammerException;

      public void search(){
            try {
                  searchPerson();
                  searchEmployee();
                  searchSecretary();
                  searchProgrammer();
            } catch (SearchProgrammerException e) {
                  log.error(e,e);
                  //obługa
            } catch (SearchSecretaryException e) {
                  log.error(e,e);
                  //obługa
            } catch (SearchEmployeeException e) {
                  log.error(e,e);
                  //obługa
            } catch (SearchPersonException e) {
                  log.error(e,e);
                  //obługa
            }
            finally {
                  log.info("...");
            }
      }
}
```

Podsumowanie

- Dziedziczenie klas
- Przesłanianie metod
- Przeciążanie metod
- Polimorfizm
- Klasy abstrakcyjne
- Słowo kluczowe final
- Klasa Object
- Obsługa wyjątków

Wyższa Szkoła Informatyki Stosowanej i Zarządzania

pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

Kierunek INFORMATYKA

Studia I stopnia (dyplom inżyniera)

Dziękuję za uwagę!