Java - opis języka

N. V. Ha
M. Kalbarczyk
M. Kowalczyk
M. Patelak

19 stycznia 2006

Java jest obiektowym językiem programowania stworzonym przez grupę roboczą pod kierunkiem Jamesa Goslinga z firmy Sun Microsystems.

"This represents the end result of nearly 15 years of trying to come up with a better programming language and environment for building simpler and more reliable software."

-Sun Microsystems cofounder Bill Joy

Historia

- Rok 1991 - projekt Green firmy Sun Microsystems.

Historia

- Rok 1991 - projekt Green firmy Sun Microsystems.
- Rok 1992 - pierwsza wersja Green z językiem Oak(James Gosling).

Historia

- Rok 1991 - projekt Green firmy Sun Microsystems.
- Rok 1992 - pierwsza wersja Green z językiem Oak(James Gosling).
- czerwiec 1994 - Java podbija Internet

Historia

- Rok 1991 - projekt Green firmy Sun Microsystems.
- Rok 1992 - pierwsza wersja Green z językiem Oak(James Gosling).
- czerwiec 1994 - Java podbija Internet
- maj 1995 - Oficjalne wydanie HotJava i Java 1.0a

Historia

- Rok 1991 - projekt Green firmy Sun Microsystems.
- Rok 1992 - pierwsza wersja Green z językiem Oak(James Gosling).
- czerwiec 1994 - Java podbija Internet
- maj 1995 - Oficjalne wydanie HotJava i Java 1.0a
- sierpień 1995 - Netscape staje się pierwszą przeglądarką obsługującą Javę

Wersje

- 1.0 -(1996) Wydanie oficjalne
- 1.1 -(1997) Małe dodatki
- 1.2 (2.0) -(1998) Playground
- 1.3 -(2000) Kestrel
- 1.4 -(2002) Merlin
- 1.5 (5.0) -(2004) Tiger

Założenia i cele przyświecające tworzeniu Javy

- Prostota.

Założenia i cele przyświecające tworzeniu Javy

- Prostota.
  - Język ma być prosty.

Założenia i cele przyświecające tworzeniu Javy

- Prostota.
  - Język ma być prosty.
  - Składnia zapożyczona z C++.

Założenia i cele przyświecające tworzeniu Javy

- Prostota.
  - Język ma być prosty.
  - Składnia zapożyczona z C++.
  - Automatyczny odśmiecacz.

Założenia i cele przyświecające tworzeniu Javy

- Prostota.
  - Język ma być prosty.
  - Składnia zapożyczona z C++.
  - Automatyczny odśmiecacz.
- Obiektowość.

Założenia i cele przyświecające tworzeniu Javy

- Prostota.
  - Język ma być prosty.
  - Składnia zapożyczona z C++.
  - Automatyczny odśmiecacz.
- Obiektowość.
  - Język od początku projektowany z myślą o obiektowości.

Założenia i cele przyświecające tworzeniu Javy

- Prostota.
  - Język ma być prosty.
  - Składnia zapożyczona z C++.
  - Automatyczny odśmiecacz.
- Obiektowość.
  - Język od początku projektowany z myślą o obiektowości.
  - Mechanizm klas, które również są obiektami.

Założenia i cele przyświecające tworzeniu Javy

- Prostota.
  - Język ma być prosty.
  - Składnia zapożyczona z C++.
  - Automatyczny odśmiecacz.
- Obiektowość.
  - Język od początku projektowany z myślą o obiektowości.
  - Mechanizm klas, które również są obiektami.
- Niezależność od architektury.

Założenia i cele przyświecające tworzeniu Javy

- Prostota.
  - Język ma być prosty.
  - Składnia zapożyczona z C++.
  - Automatyczny odśmiecacz.
- Obiektowość.
  - Język od początku projektowany z myślą o obiektowości.
  - Mechanizm klas, które również są obiektami.
- Niezależność od architektury.
  - Zastosowanie maszyny wirtualnej.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.
  - Silna kontrola typów.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.
  - Silna kontrola typów.
  - Brak wskaźników.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.
  - Silna kontrola typów.
  - Brak wskaźników.
  - Mechanizm wyjątków.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.
  - Silna kontrola typów.
  - Brak wskaźników.
  - Mechanizm wyjątków.
  - Automatyczna kontrola zakresów w tablicach.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.
  - Silna kontrola typów.
  - Brak wskaźników.
  - Mechanizm wyjątków.
  - Automatyczna kontrola zakresów w tablicach.
- Wydajność.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.
  - Silna kontrola typów.
  - Brak wskaźników.
  - Mechanizm wyjątków.
  - Automatyczna kontrola zakresów w tablicach.
- Wydajność.
  - Odwołania tylko przez referencje.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.
  - Silna kontrola typów.
  - Brak wskaźników.
  - Mechanizm wyjątków.
  - Automatyczna kontrola zakresów w tablicach.
- Wydajność.
  - Odwołania tylko przez referencje.
  - Parametry metod przekazywane także przez referencje.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.
  - Silna kontrola typów.
  - Brak wskaźników.
  - Mechanizm wyjątków.
  - Automatyczna kontrola zakresów w tablicach.
- Wydajność.
  - Odwołania tylko przez referencje.
  - Parametry metod przekazywane także przez referencje.
  - Pozostały niektóre mechanizmy nieobiektowe np. typy wbudowane.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.
  - Silna kontrola typów.
  - Brak wskaźników.
  - Mechanizm wyjątków.
  - Automatyczna kontrola zakresów w tablicach.
- Wydajność.
  - Odwołania tylko przez referencje.
  - Parametry metod przekazywane także przez referencje.
  - Pozostały niektóre mechanizmy nieobiektowe np. typy wbudowane.
  - Maszyna wirtualna - optymalizacja dla platformy.

Założenia i cele przyświecające tworzeniu Javy

- Niezawodność.
  - Automatyczny odśmiecacz.
- Bezpieczeństwo.
  - Silna kontrola typów.
  - Brak wskaźników.
  - Mechanizm wyjątków.
  - Automatyczna kontrola zakresów w tablicach.
- Wydajność.
  - Odwołania tylko przez referencje.
  - Parametry metod przekazywane także przez referencje.
  - Pozostały niektóre mechanizmy nieobiektowe np. typy wbudowane.
  - Maszyna wirtualna - optymalizacja dla platformy.
  - Maszyna wirtualna - mechanizm HotSpotów.

Typy w języku Java

1. Typy wbudowane:

Typy w języku Java

1. Typy wbudowane:
   - byte (8-bit), short (16-bit), int (32-bit), long (64-bit)

Typy w języku Java

1. Typy wbudowane:
   - byte (8-bit), short (16-bit), int (32-bit), long (64-bit)
   - float (pojedynczej precyzji), double (podwójnej precyzji)

Typy w języku Java

1. Typy wbudowane:
   - byte (8-bit), short (16-bit), int (32-bit), long (64-bit)
   - float (pojedynczej precyzji), double (podwójnej precyzji)
   - bool (true, false)

Typy w języku Java

1. Typy wbudowane:
   - byte (8-bit), short (16-bit), int (32-bit), long (64-bit)
   - float (pojedynczej precyzji), double (podwójnej precyzji)
   - bool (true, false)
   - char (UCS-2, stałej szerokości 16-bitowy format Unicode)

Typy w języku Java

1. Typy wbudowane:
   - byte (8-bit), short (16-bit), int (32-bit), long (64-bit)
   - float (pojedynczej precyzji), double (podwójnej precyzji)
   - bool (true, false)
   - char (UCS-2, stałej szerokości 16-bitowy format Unicode)
   - void

Typy w języku Java

1. Typy wbudowane:
   - byte (8-bit), short (16-bit), int (32-bit), long (64-bit)
   - float (pojedynczej precyzji), double (podwójnej precyzji)
   - bool (true, false)
   - char (UCS-2, stałej szerokości 16-bitowy format Unicode)
   - void
2. Obiekt

Typy w języku Java

1. Typy wbudowane:
   - byte (8-bit), short (16-bit), int (32-bit), long (64-bit)
   - float (pojedynczej precyzji), double (podwójnej precyzji)
   - bool (true, false)
   - char (UCS-2, stałej szerokości 16-bitowy format Unicode)
   - void
2. Obiekt
3. Typ tablicowy
   (obiekt zawierający jedno publiczne pole `length`)

Operatory

Operatory w Javie są w większości zapożyczone z języka C++. Istnieją zatem operatory o różnej arności, umiejscowieniu, czy leniwości tj.

Operatory

Operatory w Javie są w większości zapożyczone z języka C++. Istnieją zatem operatory o różnej arności, umiejscowieniu, czy leniwości tj.

- unarne - np. $-$ $(-x)$

Operatory

Operatory w Javie są w większości zapożyczone z języka C++. Istnieją zatem operatory o różnej arności, umiejscowieniu, czy leniwości tj.

- unarne - np. $-$ $(-x)$
- binarne - np. $-$ $(x - 17)$

Operatory

Operatory w Javie są w większości zapożyczone z języka C++. Istnieją zatem operatory o różnej arności, umiejscowieniu, czy leniwości tj.

- unarne - np. $-$ $(-x)$
- binarne - np. $-$ $(x - 17)$
- ternarne - np. ? : (x?jestPrawda():jestFalsz())

Operatory

Operatory w Javie są w większości zapożyczone z języka C++. Istnieją zatem operatory o różnej arności, umiejscowieniu, czy leniwości tj.

- unarne - np. $-$ $(-x)$
- binarne - np. $-$ $(x - 17)$
- ternarne - np. ? : (x?jestPrawda():jestFalsz())
- prefiksowe - np. ++ (++x)

Operatory

Operatory w Javie są w większości zapożyczone z języka C++. Istnieją zatem operatory o różnej arności, umiejscowieniu, czy leniwości tj.

- unarne - np. $-$ $(-x)$
- binarne - np. $-$ $(x - 17)$
- ternarne - np. ? : (x?jestPrawda():jestFalsz())
- prefiksowe - np. ++ (++x)
- postfiksowe - np. ++ (x++)

Operatory

Operatory w Javie są w większości zapożyczone z języka C++. Istnieją zatem operatory o różnej arności, umiejscowieniu, czy leniwości tj.

- unarne - np. $-$ $(-x)$
- binarne - np. $-$ $(x - 17)$
- ternarne - np. ? : (x?jestPrawda():jestFalsz())
- prefiksowe - np. ++ (++x)
- postfiksowe - np. ++ (x++)
- infiksowe - np. instanceof (k instanceof Klasa)

Operatory

Operatory w Javie są w większości zapożyczone z języka C++. Istnieją zatem operatory o różnej arności, umiejscowieniu, czy leniwości tj.

- unarne - np. $-$ $(-x)$
- binarne - np. $-$ $(x - 17)$
- ternarne - np. ? : (x?jestPrawda():jestFalsz())
- prefiksowe - np. ++ (++x)
- postfiksowe - np. ++ (x++)
- infiksowe - np. instanceof (k instanceof Klasa)
- leniwe - np. || lub &&

Operatory, a promocja wartości

Gdy stosujemy operator do różnych typów to:

Operatory, a promocja wartości

Gdy stosujemy operator do różnych typów to:

- Jeśli którykolwiek był typu String, to wynik będzie napisem (stosowana metoda toString()).
- Jeśli którykolwiek był typu double, to drugi też jest zmieniany na double.
- Jeśli którykolwiek był typu float, to drugi też jest zmieniany na float.
- Jeśli którykolwiek był typu long, to drugi też jest zmieniany na long.
- Jeśli żadne z powyższych nie jest spełnione to obydwa są traktowane jako int.

Operatory, a promocja wartości

Gdy stosujemy operator do różnych typów to:

- Jeśli którykolwiek był typu String, to wynik będzie napisem (stosowana metoda toString()).
- Jeśli którykolwiek był typu double, to drugi też jest zmieniany na double.
- Jeśli którykolwiek był typu float, to drugi też jest zmieniany na float.
- Jeśli którykolwiek był typu long, to drugi też jest zmieniany na long.
- Jeśli żadne z powyższych nie jest spełnione to obydwa są traktowane jako int.

Operatory, a promocja wartości

Gdy stosujemy operator do różnych typów to:

- Jeśli którykolwiek był typu String, to wynik będzie napisem (stosowana metoda toString()).
- Jeśli którykolwiek był typu double, to drugi też jest zmieniany na double.
- Jeśli którykolwiek był typu float, to drugi też jest zmieniany na float.
- Jeśli którykolwiek był typu long, to drugi też jest zmieniany na long.
- Jeśli żadne z powyższych nie jest spełnione to obydwa są traktowane jako int.

Operatory, a promocja wartości

Gdy stosujemy operator do różnych typów to:

- Jeśli którykolwiek był typu String, to wynik będzie napisem (stosowana metoda toString()).
- Jeśli którykolwiek był typu double, to drugi też jest zmieniany na double.
- Jeśli którykolwiek był typu float, to drugi też jest zmieniany na float.
- Jeśli którykolwiek był typu long, to drugi też jest zmieniany na long.
- Jeśli żadne z powyższych nie jest spełnione to obydwa są traktowane jako int.

Operatory, a promocja wartości

Gdy stosujemy operator do różnych typów to:

- Jeśli którykolwiek był typu String, to wynik będzie napisem (stosowana metoda toString()).
- Jeśli którykolwiek był typu double, to drugi też jest zmieniany na double.
- Jeśli którykolwiek był typu float, to drugi też jest zmieniany na float.
- Jeśli którykolwiek był typu long, to drugi też jest zmieniany na long.
- Jeśli żadne z powyższych nie jest spełnione to obydwa są traktowane jako int.

Operatory, a promocja wartości

Gdy stosujemy operator do różnych typów to:

- Jeśli którykolwiek był typu String, to wynik będzie napisem (stosowana metoda toString()).
- Jeśli którykolwiek był typu double, to drugi też jest zmieniany na double.
- Jeśli którykolwiek był typu float, to drugi też jest zmieniany na float.
- Jeśli którykolwiek był typu long, to drugi też jest zmieniany na long.
- Jeśli żadne z powyższych nie jest spełnione to obydwa są traktowane jako int.

Operatory, inne

Operatory, inne

- >>> - przesunięcie bitowe z uzupełnianiem zerami

Operatory, inne

- >>> - przesunięcie bitowe z uzupełnianiem zerami
- [ ] - deklarowanie i dostęp do typów tablicowych

Operatory, inne

- >>> - przesunięcie bitowe z uzupełnianiem zerami
- [ ] - deklarowanie i dostęp do typów tablicowych
- . - dostęp do składowych obiektów

Operatory, inne

- >>> - przesunięcie bitowe z uzupełnianiem zerami
- [ ] - deklarowanie i dostęp do typów tablicowych
- . - dostęp do składowych obiektów
- (parametry) - deklarowanie i przekazywanie parametrów metod

Operatory, inne

- >>> - przesunięcie bitowe z uzupełnianiem zerami
- [ ] - deklarowanie i dostęp do typów tablicowych
- . - dostęp do składowych obiektów
- (parametry) - deklarowanie i przekazywanie parametrów metod
- (typ) - rzutowanie obiektu na typ

Operatory, inne

- $>>>$ - przesunięcie bitowe z uzupełnianiem zerami
- [ ] - deklarowanie i dostęp do typów tablicowych
- . - dostęp do składowych obiektów
- (parametry) - deklarowanie i przekazywanie parametrów metod
- (typ) - rzutowanie obiektu na typ
- $< typ >$ - parametryzowanie klas typami

Operatory, inne

- $>>>$ - przesunięcie bitowe z uzupełnianiem zerami
- [ ] - deklarowanie i dostęp do typów tablicowych
- . - dostęp do składowych obiektów
- (parametry) - deklarowanie i przekazywanie parametrów metod
- (typ) - rzutowanie obiektu na typ
- $< typ >$ - parametryzowanie klas typami
- new - tworzenie nowego obiektu

Operatory, inne

- $>>>$ - przesunięcie bitowe z uzupełnianiem zerami
- [ ] - deklarowanie i dostęp do typów tablicowych
- . - dostęp do składowych obiektów
- (parametry) - deklarowanie i przekazywanie parametrów metod
- (typ) - rzutowanie obiektu na typ
- $< typ >$ - parametryzowanie klas typami
- new - tworzenie nowego obiektu
- instanceof - sprawdzanie klasy dla obiektu

Operatory, inne

- $>>>$ - przesunięcie bitowe z uzupełnianiem zerami
- [ ] - deklarowanie i dostęp do typów tablicowych
- . - dostęp do składowych obiektów
- (parametry) - deklarowanie i przekazywanie parametrów metod
- (typ) - rzutowanie obiektu na typ
- $< typ >$ - parametryzowanie klas typami
- new - tworzenie nowego obiektu
- instanceof - sprawdzanie klasy dla obiektu

Pozostałe operatory

Pełną listę operatorów wraz z opisem można znaleźć np. na stronie:
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/operators.html

Modyfikatory

- `private` - elementy widoczne jedynie wewnątrz klasy

Modyfikatory

- `private` - elementy widoczne jedynie wewnątrz klasy
- (domyślny) - tzw. "pakietowy", widoczne jedynie w obrębie pakietu

Modyfikatory

- `private` - elementy widoczne jedynie wewnątrz klasy
- (domyślny) - tzw. "pakietowy", widoczne jedynie w obrębie pakietu
- `protected` - jw. + podklasy widzą takie elementy "u siebie"

Modyfikatory

- `private` - elementy widoczne jedynie wewnątrz klasy
- (domyślny) - tzw. "pakietowy", widoczne jedynie w obrębie pakietu
- `protected` - jw. + podklasy widzą takie elementy "u siebie"
- `public` - wszystkie klasy maja dostęp

Modyfikatory

- `private` - elementy widoczne jedynie wewnątrz klasy
- (domyślny) - tzw. "pakietowy", widoczne jedynie w obrębie pakietu
- `protected` - jw. + podklasy widzą takie elementy "u siebie"
- `public` - wszystkie klasy maja dostęp
- `final` - odpowiednio dla:
  - metody - nie mogą być przedefiniowane w podklasach, inline

Modyfikatory

- `private` - elementy widoczne jedynie wewnątrz klasy
- (domyślny) - tzw. "pakietowy", widoczne jedynie w obrębie pakietu
- `protected` - jw. + podklasy widzą takie elementy "u siebie"
- `public` - wszystkie klasy maja dostęp
- `final` - odpowiednio dla:
  - metody - nie mogą być przedefiniowane w podklasach, inline
  - klasy - nie można po niej dziedziczyć

Modyfikatory

- `private` - elementy widoczne jedynie wewnątrz klasy
- (domyślny) - tzw. "pakietowy", widoczne jedynie w obrębie pakietu
- `protected` - jw. + podklasy widzą takie elementy "u siebie"
- `public` - wszystkie klasy maja dostęp
- `final` - odpowiednio dla:
  - metody - nie mogą być przedefiniowane w podklasach, inline
  - klasy - nie można po niej dziedziczyć
  - pola - (referencja) nie może być modyfikowana

Modyfikatory

- `private` - elementy widoczne jedynie wewnątrz klasy
- (domyślny) - tzw. "pakietowy", widoczne jedynie w obrębie pakietu
- `protected` - jw. + podklasy widzą takie elementy "u siebie"
- `public` - wszystkie klasy maja dostęp
- `final` - odpowiednio dla:
  - metody - nie mogą być przedefiniowane w podklasach, inline
  - klasy - nie można po niej dziedziczyć
  - pola - (referencja) nie może być modyfikowana
- `static`
  - statyczna inicjalizacja (pola, bloki kodu) tylko przy pierwszym instancjonowaniu klasy

Modyfikatory

- `private` - elementy widoczne jedynie wewnątrz klasy
- (domyślny) - tzw. "pakietowy", widoczne jedynie w obrębie pakietu
- `protected` - jw. + podklasy widzą takie elementy "u siebie"
- `public` - wszystkie klasy maja dostęp
- `final` - odpowiednio dla:
  - metody - nie mogą być przedefiniowane w podklasach, inline
  - klasy - nie można po niej dziedziczyć
  - pola - (referencja) nie może być modyfikowana
- `static`
  - statyczna inicjalizacja (pola, bloki kodu) tylko przy pierwszym instancjonowaniu klasy
  - statyczne pola, metody, klasy wewnętrzne są własnością klasy, a nie jej instancji oraz nie mogą się do nich odwoływać

Modyfikatory cd.

- `abstract` - odpowiednio dla:
  - klasy - nieinstancjonowalna

Modyfikatory cd.

- `abstract` - odpowiednio dla:
  - klasy - nieinstancjonowalna
  - metody - niezdefiniowana, wymusza abstrakcyjność klasy, w nieabstrakcyjnej podklasie musi być zdefiniowana

Modyfikatory cd.

- `abstract` - odpowiednio dla:
  - klasy - nieinstancjonowalna
  - metody - niezdefiniowana, wymusza abstrakcyjność klasy, w nieabstrakcyjnej podklasie musi być zdefiniowana
- synchronized - używany dla oznaczenia sekcji krytycznej

Modyfikatory cd.

- `abstract` - odpowiednio dla:
  - klasy - nieinstancjonowalna
  - metody - niezdefiniowana, wymusza abstrakcyjność klasy, w nieabstrakcyjnej podklasie musi być zdefiniowana
- synchronized - używany dla oznaczenia sekcji krytycznej
- native - użycie konstrukcji z innego języka (np. C, Assemblera)

Modyfikatory cd.

- `abstract` - odpowiednio dla:
  - klasy - nieinstancjonowalna
  - metody - niezdefiniowana, wymusza abstrakcyjność klasy, w nieabstrakcyjnej podklasie musi być zdefiniowana
- synchronized - używany dla oznaczenia sekcji krytycznej
- native - użycie konstrukcji z innego języka (np. C, Assemblera)
- strictfp - zapewnia identyczną (dla wszelkich maszyn wirtualnych) dokładność obliczeń zmiennoprzecinkowych

Modyfikatory cd.

- `abstract` - odpowiednio dla:
  - klasy - nieinstancjonowalna
  - metody - niezdefiniowana, wymusza abstrakcyjność klasy, w nieabstrakcyjnej podklasie musi być zdefiniowana
- synchronized - używany dla oznaczenia sekcji krytycznej
- native - użycie konstrukcji z innego języka (np. C, Assemblera)
- strictfp - zapewnia identyczną (dla wszelkich maszyn wirtualnych) dokładność obliczeń zmiennoprzecinkowych
- transient - pole nie jest zachowywane przy serializacji obiektu

Modyfikatory cd.

- `abstract` - odpowiednio dla:
  - klasy - nieinstancjonowalna
  - metody - niezdefiniowana, wymusza abstrakcyjność klasy, w nieabstrakcyjnej podklasie musi być zdefiniowana
- synchronized - używany dla oznaczenia sekcji krytycznej
- native - użycie konstrukcji z innego języka (np. C, Assemblera)
- strictfp - zapewnia identyczną (dla wszelkich maszyn wirtualnych) dokładność obliczeń zmiennoprzecinkowych
- transient - pole nie jest zachowywane przy serializacji obiektu
- volatile - pole może być modyfikowane asynchronicznie, przez konkurencyjne wątki

Przeciążanie metod

- Metodę rozróżniamy po jej nazwie oraz liście (uporządkowanej) jej argumentów.

Przeciążanie metod

- Metodę rozróżniamy po jej nazwie oraz liście (uporządkowanej) jej argumentów.
- Wybierana jest metoda o szerszym typie.

Przeciążanie metod

- Metodę rozróżniamy po jej nazwie oraz liście (uporządkowanej) jej argumentów.
- Wybierana jest metoda o szerszym typie.
- `char` jest promowany do `int`.

Przeciążanie metod

- Metodę rozróżniamy po jej nazwie oraz liście (uporządkowanej) jej argumentów.
- Wybierana jest metoda o szerszym typie.
- `char` jest promowany do `int`.
- Przy zwężaniu, potrzebne jest rzutowanie.

Przeciążanie konstruktorów

- Może być kilka konstruktorów różniących się listą parametrów.

Przeciążanie konstruktorów

- Może być kilka konstruktorów różniących się listą parametrów.
- Przeciążając konstruktor, można posłużyć się innym konstruktorem, ale tylko jednym i musi być wywołany w pierwszej linijce definiowanego konstruktora.

Typy Generyczne

Czym są typy generyczne?

Typy Generyczne

Czym są typy generyczne?

- Składnią przypominają szablony z języka C.

Typy Generyczne

Czym są typy generyczne?

- Składnią przypominają szablony z języka C.
- Służą do parametryzowania obiektów oraz metod klasami.

Typy Generyczne

Czym są typy generyczne?

- Składnią przypominają szablony z języka C.
- Służą do parametryzowania obiektów oraz metod klasami.
- Istnieje specjalny znak ? pasujący do dowolnego typu.

Typy Generyczne

Czym są typy generyczne?

- Składnią przypominają szablony z języka C.
- Służą do parametryzowania obiektów oraz metod klasami.
- Istnieje specjalny znak ? pasujący do dowolnego typu.
-

```
[ modyfikatory ] class Klasa < T > {
...
T atrybut;
...
T metoda(...,T x, ...);
...
}
```

Typy Generyczne

Od kiedy typy generyczne?

Typy generyczne pojawiły się w Javie od wersji 1.5.
Pomimo wprowadzenia nowej konstrukcji do języka, zachowano zgodność wstecz. Niestety, skutkiem ubocznym zasady kompatybilności jest możliwość „oszukania" typów generycznych.

Typy Generyczne

Od kiedy typy generyczne?

Typy generyczne pojawiły się w Javie od wersji 1.5.
Pomimo wprowadzenia nowej konstrukcji do języka, zachowano zgodność wstecz. Niestety, skutkiem ubocznym zasady kompatybilności jest możliwość „oszukania" typów generycznych.

Po co są typy generyczne?

- Większa kontrola programisty nad programem.

Typy Generyczne

Od kiedy typy generyczne?

Typy generyczne pojawiły się w Javie od wersji 1.5.
Pomimo wprowadzenia nowej konstrukcji do języka, zachowano zgodność wstecz. Niestety, skutkiem ubocznym zasady kompatybilności jest możliwość „oszukania" typów generycznych.

Po co są typy generyczne?

- Większa kontrola programisty nad programem.
- Mniejsze ryzyko popełnienia błędu - ściślejsza kontrola typów.

Typy Generyczne

Od kiedy typy generyczne?

Typy generyczne pojawiły się w Javie od wersji 1.5.
Pomimo wprowadzenia nowej konstrukcji do języka, zachowano zgodność wstecz. Niestety, skutkiem ubocznym zasady kompatybilności jest możliwość „oszukania" typów generycznych.

Po co są typy generyczne?

- Większa kontrola programisty nad programem.
- Mniejsze ryzyko popełnienia błędu - ściślejsza kontrola typów.
- Przydatne np. przy wielu kolekcjach obiektów jednego typu (ale każda z nich innego).

Dziedziczenie

Definicja

Przyjęta definicja mówi, że klasa SYN dziedziczy (rozszerza) po klasie OJCIEC wtedy, kiedy obiektu klasy SYN można użyć wszędzie tam, gdzie można było użyć obiektu klasy OJCIEC.

Dziedziczenie

Zasady

Dziedziczenie

Zasady

- Zasada dziedziczenia zgodna z tym co znamy z innych języków obiektowych (przesłanianie i rozszerzanie).

Dziedziczenie

Zasady

- Zasada dziedziczenia zgodna z tym co znamy z innych języków obiektowych (przesłanianie i rozszerzanie).
- Nie ma wielodziedziczenia klas (jest wielodziedziczenie interfejsów).

Dziedziczenie

Zasady

- Zasada dziedziczenia zgodna z tym co znamy z innych języków obiektowych (przesłanianie i rozszerzanie).
- Nie ma wielodziedziczenia klas (jest wielodziedziczenie interfejsów).
- Dynamiczne wiązanie dla metod, ale nie dla zmiennych!

Dziedziczenie

Zasady

- Zasada dziedziczenia zgodna z tym co znamy z innych języków obiektowych (przesłanianie i rozszerzanie).
- Nie ma wielodziedziczenia klas (jest wielodziedziczenie interfejsów).
- Dynamiczne wiązanie dla metod, ale nie dla zmiennych!
- Składnia jest postaci:

```
[ modyfikatory ] class Syn extends Ojciec {
...
}
```

Dziedziczenie

Kolejność inicjacji przy dziedziczeniu

Dziedziczenie

Kolejność inicjacji przy dziedziczeniu

1. Statyczna część nadklasy (tylko za pierwszym razem).

Dziedziczenie

Kolejność inicjacji przy dziedziczeniu

1. Statyczna część nadklasy (tylko za pierwszym razem).
2. Statyczna część podklasy (tylko za pierwszym razem).

Dziedziczenie

Kolejność inicjacji przy dziedziczeniu

1. Statyczna część nadklasy (tylko za pierwszym razem).
2. Statyczna część podklasy (tylko za pierwszym razem).
3. Niestatyczna część nadklasy.

Dziedziczenie

Kolejność inicjacji przy dziedziczeniu

1. Statyczna część nadklasy (tylko za pierwszym razem).
2. Statyczna część podklasy (tylko za pierwszym razem).
3. Niestatyczna część nadklasy.
4. Konstruktor nadklasy.

Dziedziczenie

Kolejność inicjacji przy dziedziczeniu

1. Statyczna część nadklasy (tylko za pierwszym razem).
2. Statyczna część podklasy (tylko za pierwszym razem).
3. Niestatyczna część nadklasy.
4. Konstruktor nadklasy.
5. Niestatyczna część podklasy.

Dziedziczenie

Kolejność inicjacji przy dziedziczeniu

1. Statyczna część nadklasy (tylko za pierwszym razem).
2. Statyczna część podklasy (tylko za pierwszym razem).
3. Niestatyczna część nadklasy.
4. Konstruktor nadklasy.
5. Niestatyczna część podklasy.
6. Konstruktor podklasy.

Dziedziczenie

Literał super

W podklasie, podobnie jak w Smalltalku, do nadklasy możemy odwoływać się poprzez literał **super**.

Literał this

Do obiektu, który aktualnie wykonuje daną część kodu możemy dostać się poprzez się poprzez literał **this**.

Dziedziczenie

Literał super

W podklasie, podobnie jak w Smalltalku, do nadklasy możemy odwoływać się poprzez literał **super**.

- Pola: super.pole

Literał this

Do obiektu, który aktualnie wykonuje daną część kodu możemy dostać się poprzez się poprzez literał **this**.

Dziedziczenie

Literał super

W podklasie, podobnie jak w Smalltalku, do nadklasy możemy odwoływać się poprzez literał **super**.

- Pola: super.pole
- Metody: super.metoda([parametry])

Literał this

Do obiektu, który aktualnie wykonuje daną część kodu możemy dostać się poprzez się poprzez literał **this**.

Dziedziczenie

Literał super

W podklasie, podobnie jak w Smalltalku, do nadklasy możemy odwoływać się poprzez literał **super**.

- Pola: super.pole
- Metody: super.metoda([parametry])
- Konstruktory: super([parametry])
  Tak możemy jawnie wywołać konstruktor nadklasy, ale pod warunkiem, że umieścimy to w pierwszej linijce konstruktora podklasy.

Literał this

Do obiektu, który aktualnie wykonuje daną część kodu możemy dostać się poprzez się poprzez literał **this**.

Uszczegóławianie wartości zwracanych przez metody

Co to jest?

Zgodnie z definicją dziedziczenia, kiedy uszczegóławiamy jakąś klasę powinniśmy móc również uszczegółowić typy zwracane przez rozszerzane metody.

Uszczegóławianie wartości zwracanych przez metody

Jak to wygląda w Javie?

Uszczegóławianie wartości zwracanych przez metody

Jak to wygląda w Javie?

- Java ver. $<$ 1.5
- Java ver. $>=$ 1.5

Uszczegóławianie wartości zwracanych przez metody

Jak to wygląda w Javie?

- Java ver. < 1.5
  1. Nie ma uszczegóławiania typów zwracanych przez metody.
- Java ver. $>= 1.5$
  1. Wprowadzono uszczegóławiania typów zwracanych przez metody.

Uszczegóławianie wartości zwracanych przez metody

Jak to wygląda w Javie?

- Java ver. < 1.5
  1. Nie ma uszczegóławiania typów zwracanych przez metody.
  2. Nie możemy uszczegółowić wartości zwracanych przez metody nadklasy.
- Java ver. $>= 1.5$
  1. Wprowadzono uszczegóławiania typów zwracanych przez metody.
  2. Możemy uszczegółowić wartości zwracane przez metody nadklasy.

Uszczegóławianie wartości zwracanych przez metody

Jak to wygląda w Javie?

- Java ver. < 1.5
  1. Nie ma uszczegóławiania typów zwracanych przez metody.
  2. Nie możemy uszczegółowić wartości zwracanych przez metody nadklasy.
  3. Jesteśmy zmuszeni w tej sytuacji do rzutowania.
- Java ver. $>= 1.5$
  1. Wprowadzono uszczegóławiania typów zwracanych przez metody.
  2. Możemy uszczegółowić wartości zwracane przez metody nadklasy.
  3. Nie musimy w tej sytuacji używać rzutowania.

Interfejsy

Interfejsy

- Interfejsy to elementy Języka, które zawierają definicje metod i atrybutów.

Interfejsy

- Interfejsy to elementy Języka, które zawierają definicje metod i atrybutów.
- Interfejsy nie zawierają implementacji.

Interfejsy

- Interfejsy to elementy Języka, które zawierają definicje metod i atrybutów.
- Interfejsy nie zawierają implementacji.
- Interfejsy mogą po sobie dziedziczyć tak jak klasy.

Interfejsy

- Interfejsy to elementy Języka, które zawierają definicje metod i atrybutów.
- Interfejsy nie zawierają implementacji.
- Interfejsy mogą po sobie dziedziczyć tak jak klasy.
- Wielodziedziczenie interfejsów jest dozwolone, ponieważ interfejsy nie zawierają implementacji metod, a zatem nie ma niejednoznaczności (wielu ścieżek w grafie dziedziczenia).

Interfejsy

- Interfejsy to elementy Języka, które zawierają definicje metod i atrybutów.
- Interfejsy nie zawierają implementacji.
- Interfejsy mogą po sobie dziedziczyć tak jak klasy.
- Wielodziedziczenie interfejsów jest dozwolone, ponieważ interfejsy nie zawierają implementacji metod, a zatem nie ma niejednoznaczności (wielu ścieżek w grafie dziedziczenia).
- Ponieważ atrybuty klas są wiązane statycznie, nie ma obawy o utracenie informacji przy implementacji kilku interfejsów z takimi samymi zmiennymi (choć należy tego unikać).

Interfejsy

- Interfejsy to elementy Języka, które zawierają definicje metod i atrybutów.
- Interfejsy nie zawierają implementacji.
- Interfejsy mogą po sobie dziedziczyć tak jak klasy.
- Wielodziedziczenie interfejsów jest dozwolone, ponieważ interfejsy nie zawierają implementacji metod, a zatem nie ma niejednoznaczności (wielu ścieżek w grafie dziedziczenia).
- Ponieważ atrybuty klas są wiązane statycznie, nie ma obawy o utracenie informacji przy implementacji kilku interfejsów z takimi samymi zmiennymi (choć należy tego unikać).
- Interfejs nie może rozszerzać dwóch interfejsów mających metody o tej samej nazwie i parametrach, a różnych typach wyniku.

Interfejsy

- Interfejsy to elementy Języka, które zawierają definicje metod i atrybutów.
- Interfejsy nie zawierają implementacji.
- Interfejsy mogą po sobie dziedziczyć tak jak klasy.
- Wielodziedziczenie interfejsów jest dozwolone, ponieważ interfejsy nie zawierają implementacji metod, a zatem nie ma niejednoznaczności (wielu ścieżek w grafie dziedziczenia).
- Ponieważ atrybuty klas są wiązane statycznie, nie ma obawy o utracenie informacji przy implementacji kilku interfejsów z takimi samymi zmiennymi (choć należy tego unikać).
- Interfejs nie może rozszerzać dwóch interfejsów mających metody o tej samej nazwie i parametrach, a różnych typach wyniku.
- Przy uogólnianiu wybierany jest typ najbardziej szczegółowy.

Interfejsy

Interfejsy jako sposób na wielodziedziczenie klas

Ponieważ interfejsy mogą po sobie wielodziedziczyć, to klasy je implementujące posiadają część związanej z tym funkcjonalności. Klasa implementuje interfejsy w następujący sposób:

```
class Syn extends Ojciec implements Interfejs1, Interfejs2
{
...
przedefiniowanie metod zadeklarowanych w interfejsach
...
}
```

Mechanizm wyjątków

- sytuacje wyjątkowe zgłaszamy za pomocą `throw`

Mechanizm wyjątków

- sytuacje wyjątkowe zgłaszamy za pomocą `throw`
- jeśli wyjątek może być zgłoszony na zewnątrz deklarujemy za pomocą `throws`

Mechanizm wyjątków

- sytuacje wyjątkowe zgłaszamy za pomocą `throw`
- jeśli wyjątek może być zgłoszony na zewnątrz deklarujemy za pomocą `throws`
- nieobsłużone wyjątki, wydostając się poza main, powodują przerwanie programu

Obsługa wyjątków

- za pomocą bloku `try` otaczamy kod, który może zwrócić wyjątek

```
try {
//...
} catch (TypWyjątku wyjątek) {
//...
}
```

Obsługa wyjątków

- za pomocą bloku `try` otaczamy kod, który może zwrócić wyjątek
- za pomocą bloku `catch` możemy obsłużyć wyjątek

```
try {
//...
} catch (TypWyjątku wyjątek) {
//...
}
```

Obsługa wyjątków
catch

Jeśli TypWyjątku nie pasuje, nie jest obsługiwany w bloku catch, tylko przekazywany dalej.

Wyjątki jako obiekty

- wyjątki dziedziczą po klasie `Throwable`, która ma dwie podklasy

Wyjątki jako obiekty

- wyjątki dziedziczą po klasie `Throwable`, która ma dwie podklasy
  - `Error` - poważne błędy niezwiązane z działaniem aplikacji; nie trzeba ich oczywiście deklarować w metodach

Wyjątki jako obiekty

- wyjątki dziedziczą po klasie `Throwable`, która ma dwie podklasy
  - `Error` - poważne błędy niezwiązane z działaniem aplikacji; nie trzeba ich oczywiście deklarować w metodach
  - `Exception` - błędy, na które aplikacja powinna reagować

Wyjątki jako obiekty

- wyjątki dziedziczą po klasie `Throwable`, która ma dwie podklasy
  - `Error` - poważne błędy niezwiązane z działaniem aplikacji; nie trzeba ich oczywiście deklarować w metodach
  - `Exception` - błędy, na które aplikacja powinna reagować
- `RuntimeException` to podklasa `Exception` reprezentująca błędy podczas działania maszyny wirtualnej (np. dzielenie przez zero); tych wyjątków także nie trzeba deklarować

Kilka klauzul catch

- Do jednego bloku `try` można dodać kilka klauzul `catch`.

Kilka klauzul catch

- Do jednego bloku `try` można dodać kilka klauzul `catch`.
- Dopasowana jest pierwsza klauzula `catch`, do której pasuje wyjątek. Pozostałe są ignorowane.

Kilka klauzul catch

- Do jednego bloku `try` można dodać kilka klauzul `catch`.
- Dopasowana jest pierwsza klauzula `catch`, do której pasuje wyjątek. Pozostałe są ignorowane.
- Kompilator nie pozwala, by któraś klauzula przesłoniła następujące po niej.

Klauzula finally

Zamiast klauzul `catch` lub po ostatniej z nich można dodać klauzulę `finally`, której kod będzie wykonany niezależnie od zgłoszenia wyjątku.

Wyjątki i dziedziczenie

Obiektów podklasy można używać wszędzie tam, gdzie obiektów nadklasy, zatem:

Wyjątki i dziedziczenie

Obiektów podklasy można używać wszędzie tam, gdzie obiektów nadklasy, zatem:

- Rozszerzane metody nie mogą zgłaszać wyjątków, których dotąd nie trzeba było obsługiwać.

Wyjątki i dziedziczenie

Obiektów podklasy można używać wszędzie tam, gdzie obiektów nadklasy, zatem:

- Rozszerzane metody nie mogą zgłaszać wyjątków, których dotąd nie trzeba było obsługiwać.
- Rozszerzane metody mogą ograniczyć zgłaszane wyjątki, np. je uogólnić.

Wyjątki i dziedziczenie

Obiektów podklasy można używać wszędzie tam, gdzie obiektów nadklasy, zatem:

- Rozszerzane metody nie mogą zgłaszać wyjątków, których dotąd nie trzeba było obsługiwać.
- Rozszerzane metody mogą ograniczyć zgłaszane wyjątki, np. je uogólnić.

Konstruktory podklasy nie mają możliwości obsługi wyjątków zgłaszanych przez konstruktor nadklasy.

Wątki
Założenia i korzyści

- Realizacja koncepcji wielozadaniowości

Wątki
Założenia i korzyści

- Realizacja koncepcji wielozadaniowości
  - Wiele zadań jest wykonywanych w jednym momencie

Wątki
Założenia i korzyści

- Realizacja koncepcji wielozadaniowości
  - Wiele zadań jest wykonywanych w jednym momencie
  - Każdy wątek uruchamia własny fragment kodu

Wątki
Założenia i korzyści

- Realizacja koncepcji wielozadaniowości
  - Wiele zadań jest wykonywanych w jednym momencie
  - Każdy wątek uruchamia własny fragment kodu
- Tworzenie mechanizmu dla programowania współbieżnego

Wątki
Założenia i korzyści

- Realizacja koncepcji wielozadaniowości
  - Wiele zadań jest wykonywanych w jednym momencie
  - Każdy wątek uruchamia własny fragment kodu
- Tworzenie mechanizmu dla programowania współbieżnego
  - Szeregowanie zadań(thread scheduler)

Wątki
Założenia i korzyści

- Realizacja koncepcji wielozadaniowości
  - Wiele zadań jest wykonywanych w jednym momencie
  - Każdy wątek uruchamia własny fragment kodu
- Tworzenie mechanizmu dla programowania współbieżnego
  - Szeregowanie zadań(thread scheduler)
  - Mechanizm sekcji krytycznej

Wątki
Definicja

1. Podklasa klasy Thread

Wątki
Definicja

1. Podklasa klasy Thread

```
public class Watek extends Thread{
  public void run(){
  /* ... */
  }
}
```

Wątki
Definicja

1. Podklasa klasy Thread

```
public class Watek extends Thread{
  public void run(){
  /* ... */
  }
}
```

2. Implementacja interfejsu Runnable

Wątki
Definicja

1. Podklasa klasy Thread

```
public class Watek extends Thread{
  public void run(){
  /* ... */
  }
}
```

2. Implementacja interfejsu Runnable

```
public class Watek implements Runnable{
  Thread T;
  public void run(){
  /* ... */
  }
}
```

Wątki
Wykonywanie

Główną metodą klasy wątku jest metoda run.
Całe zadanie wątku polega na wykonywaniu zadanego kodu.
Wątek jest traktowany jako "żywy" jeśli znajduje się w trakcie wykonywania metody run.

Wątki
Wykonywanie

```
public void run()
{
 while(true)
 {
  Count++;
  System.out.println(Count);
  try {
   t.sleep(10);
  }catch (InterruptedException e) {}
 }
}
```

Wątki
Uruchamianie

1. start()
   Metoda ta służy do uruchomienia wątku.

Wątki
Uruchamianie

1. start()
   Metoda ta służy do uruchomienia wątku.
2. stop()
   Metoda ta kończy działanie wątku.

Wątki
Uruchamianie

1. start()
   Metoda ta służy do uruchomienia wątku.
2. stop()
   Metoda ta kończy działanie wątku.
3. suspend()
   Metoda ta usypia wątek. Należy jednak do metod niebezpiecznych gdyż może wywołać zakleszczenie.

Wątki
Uruchamianie

1. start()
   Metoda ta służy do uruchomienia wątku.
2. stop()
   Metoda ta kończy działanie wątku.
3. suspend()
   Metoda ta usypia wątek. Należy jednak do metod niebezpiecznych gdyż może wywołać zakleszczenie.
4. resume()
   Metoda ta budzi wątek. Należy jednak do metod niebezpiecznych gdyż może wywołać zakleszczenie.

Watki
Mechanizm synchronizacji

- sekcja krytyczna

Watki
Mechanizm synchronizacji

- sekcja krytyczna

```
synchronized ( identyfikator_obiektu )
{
 //kod sekcji krytycznej
}
```

Watki
Mechanizm synchronizacji

- sekcja krytyczna

```
synchronized ( identyfikator_obiektu )
{
 //kod sekcji krytycznej
}
```

- metody krytyczne

Watki
Mechanizm synchronizacji

- sekcja krytyczna

```
synchronized ( identyfikator_obiektu )
{
 //kod sekcji krytycznej
}
```

- metody krytyczne

```
class Obiekt
{
 String nazwa;
 synchronized void zmien(String nowanazwa)
 {
  nazwa=nowanazwa
 }
}
```

Wątki
Mechanizm synchronizacji

- wait()
  Zawieszenie wątku na obiekcie w oczekiwaniu na sygnał budzący. Wywołując tę metodę wątek zwalnia wszystkie blokady.

Wątki
Mechanizm synchronizacji

- wait()
  Zawieszenie wątku na obiekcie w oczekiwaniu na sygnał budzący. Wywołując tę metodę wątek zwalnia wszystkie blokady.
- notify()
  Wysłanie sygnału budzącego. Jeśli jakiś wątek oczekuje na sygnał na danym obiekcie to zostanie obudzony.

Wątki
Nowe mechanizmy synchronizacji

- Semafory

Wątki
Nowe mechanizmy synchronizacji

- Semafory
- Bariery

Wątki
Nowe mechanizmy synchronizacji

- Semafory
- Bariery
- Spotkania

Wątki
Nowe mechanizmy synchronizacji

- Semafory
- Bariery
- Spotkania
- Blokady czytania i pisania

Kolekcje
Założenia

1. Oddzielenie interfejsu od implementacji:

Kolekcje
Założenia

1. Oddzielenie interfejsu od implementacji:
   - Odwołujemy się do kolekcji poprzez interfejsy.

Kolekcje
Założenia

1. Oddzielenie interfejsu od implementacji:
   - Odwołujemy się do kolekcji poprzez interfejsy.
   - Instancje tworzymy za pomocą konkretnych kolekcji.

Kolekcje
Założenia

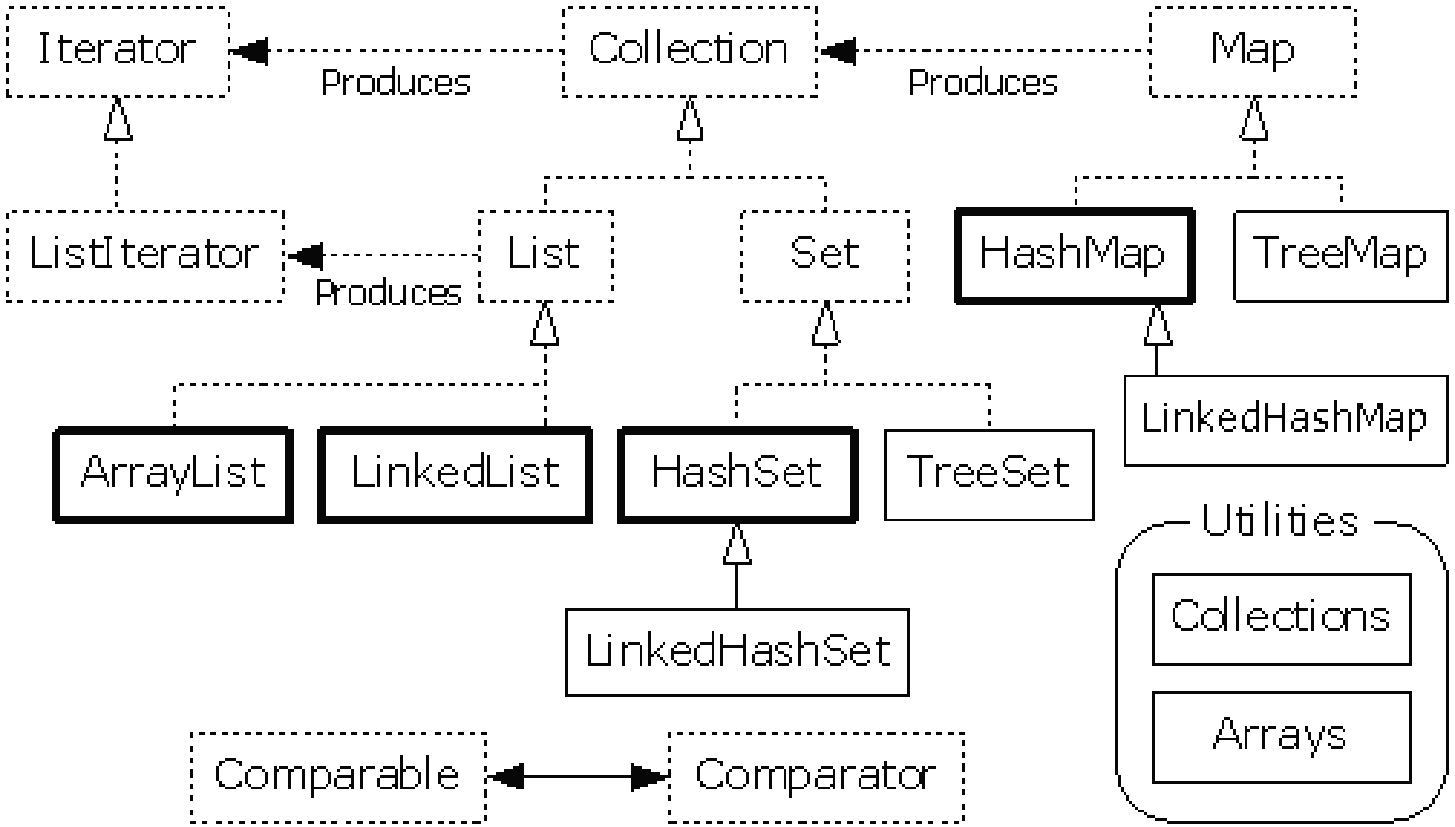
1. Oddzielenie interfejsu od implementacji:
   - Odwołujemy się do kolekcji poprzez interfejsy.
   - Instancje tworzymy za pomocą konkretnych kolekcji.
2. Maksymalne uproszczenie hierarchii interfejsów.

Kolekcje
Założenia

1. Oddzielenie interfejsu od implementacji:
   - Odwołujemy się do kolekcji poprzez interfejsy.
   - Instancje tworzymy za pomocą konkretnych kolekcji.
2. Maksymalne uproszczenie hierarchii interfejsów.
3. Wykorzystanie typów generycznych do otypowania kolekcji.

Kolekcje

Główna część API kolekcji zawartego w pakiecie `java.util`:



Kolekcje

```
public interface Collection<E> extends Iterable<E> {
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                    //z Iterable<E>
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                    //z Iterable<E>
  Object[] toArray();
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                    //z Iterable<E>
  Object[] toArray();
  <T> T[] toArray(T[] a);
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                    //z Iterable<E>
  Object[] toArray();
  <T> T[] toArray(T[] a);
  boolean add(E);                           //optional
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                    //z Iterable<E>
  Object[] toArray();
  <T> T[] toArray(T[] a);
  boolean add(E);                           //optional
  boolean remove(Object);                   //optional
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                    //z Iterable<E>
  Object[] toArray();
  <T> T[] toArray(T[] a);
  boolean add(E);                           //optional
  boolean remove(Object);                   //optional
  boolean containsAll(Collection<?>);
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                          //z Iterable<E>
  Object[] toArray();
  <T> T[] toArray(T[] a);
  boolean add(E);                                 //optional
  boolean remove(Object);                         //optional
  boolean containsAll(Collection<?>);
  boolean addAll(Collection<? extends E>);        //optional
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                          //z Iterable<E>
  Object[] toArray();
  <T> T[] toArray(T[] a);
  boolean add(E);                                 //optional
  boolean remove(Object);                         //optional
  boolean containsAll(Collection<?>);
  boolean addAll(Collection<? extends E>);        //optional
  boolean removeAll(Collection<?>;                //optional
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                          //z Iterable<E>
  Object[] toArray();
  <T> T[] toArray(T[] a);
  boolean add(E);                                 //optional
  boolean remove(Object);                         //optional
  boolean containsAll(Collection<?>);
  boolean addAll(Collection<? extends E>);        //optional
  boolean removeAll(Collection<?>;                //optional
  boolean retainAll(Collection<?>);               //optional
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                          //z Iterable<E>
  Object[] toArray();
  <T> T[] toArray(T[] a);
  boolean add(E);                                 //optional
  boolean remove(Object);                         //optional
  boolean containsAll(Collection<?>);
  boolean addAll(Collection<? extends E>);        //optional
  boolean removeAll(Collection<?>;                //optional
  boolean retainAll(Collection<?>);               //optional
  void clear();                                   //optional
```

Kolekcje

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object);
  Iterator iterator();                          //z Iterable<E>
  Object[] toArray();
  <T> T[] toArray(T[] a);
  boolean add(E);                                 //optional
  boolean remove(Object);                         //optional
  boolean containsAll(Collection<?>);
  boolean addAll(Collection<? extends E>);        //optional
  boolean removeAll(Collection<?>;                //optional
  boolean retainAll(Collection<?>);               //optional
  void clear();                                   //optional
  /* ... */
}
```

Iteratory

```
public interface Iterator<E> {
```

Iteratory

```
public interface Iterator<E> {
  boolean hasNext();
```

Iteratory

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
```

Iteratory

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();                                    //optional
```

Iteratory

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();                                    //optional
}
```

Iteratory

```
public interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();
  E next();
  void remove();                                    //optional

  boolean hasPrevious();
```

Iteratory

```
public interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();
  E next();
  void remove();                                    //optional

  boolean hasPrevious();
  E previous();
```

Iteratory

```
public interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();
  E next();
  void remove();                                    //optional

  boolean hasPrevious();
  E previous();
  int nextIndex();
```

Iteratory

```
public interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();
  E next();
  void remove();                                    //optional

  boolean hasPrevious();
  E previous();
  int nextIndex();
  int previousIndex();
```

Iteratory

```
public interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();
  E next();
  void remove();                                    //optional

  boolean hasPrevious();
  E previous();
  int nextIndex();
  int previousIndex();
  void set(E o);                                    //optional
```

Iteratory

```
public interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();
  E next();
  void remove();                                    //optional

  boolean hasPrevious();
  E previous();
  int nextIndex();
  int previousIndex();
  void set(E o);                                    //optional
  void add(E o);                                    //optional
}
```

Przykład

```
Collection<Integer> intKolekcja = new HashSet<Integer>();
intKolekcja.add(1); // auto-boxing (od 1.5)
intKolekcja.add(2);
intKolekcja.add(3);
```

Przykład

```
Collection<Integer> intKolekcja = new HashSet<Integer>();
intKolekcja.add(1); // auto-boxing (od 1.5)
intKolekcja.add(2);
intKolekcja.add(3);

Iterator<Integer> i = intKolekcja.iterator();
while (i.hasNext())
  System.out.println(i.next());
```

Przykład

```
Collection<Integer> intKolekcja = new HashSet<Integer>();
intKolekcja.add(1); // auto-boxing (od 1.5)
intKolekcja.add(2);
intKolekcja.add(3);

Iterator<Integer> i = intKolekcja.iterator();
while (i.hasNext())
  System.out.println(i.next());

// nowa (1.5) składnia for'a:
for (Integer n : intKolekcja)
  System.out.println(n);
```

Klasa Collections

Klasa zawiera sporo użytecznych funkcji działających na kolekcjach, jak:

```
//sortowanie listy przy pomocy podanego komparatora
<T> void sort(List<T> list, Comparator<? super T> c);
```

Klasa Collections

Klasa zawiera sporo użytecznych funkcji działających na kolekcjach, jak:

```
//sortowanie listy przy pomocy podanego komparatora
<T> void sort(List<T> list, Comparator<? super T> c);

//wyszukiwanie binarne listy przy pomocy podanego komparatora
<T> int binarySearch(List<? extends T> list, T key, Comparator<? super
T> c);
```

Klasa Collections

Klasa zawiera sporo użytecznych funkcji działających na kolekcjach, jak:

```
//sortowanie listy przy pomocy podanego komparatora
<T> void sort(List<T> list, Comparator<? super T> c);

//wyszukiwanie binarne listy przy pomocy podanego komparatora
<T> int binarySearch(List<? extends T> list, T key, Comparator<? super
T> c);

//wypełnianie listy zadanym obiektem
<T> void fill(List<? super T> list, T obj);
```

Klasa Collections

Klasa zawiera sporo użytecznych funkcji działających na kolekcjach, jak:

```
//sortowanie listy przy pomocy podanego komparatora
<T> void sort(List<T> list, Comparator<? super T> c);

//wyszukiwanie binarne listy przy pomocy podanego komparatora
<T> int binarySearch(List<? extends T> list, T key, Comparator<? super
T> c);

//wypełnianie listy zadanym obiektem
<T> void fill(List<? super T> list, T obj);

<T> T max(Collection<? extends T> coll, Comparator<? super T> comp);
```

Klasa Collections

Klasa zawiera sporo użytecznych funkcji działających na kolekcjach, jak:

```
//sortowanie listy przy pomocy podanego komparatora
<T> void sort(List<T> list, Comparator<? super T> c);

//wyszukiwanie binarne listy przy pomocy podanego komparatora
<T> int binarySearch(List<? extends T> list, T key, Comparator<? super
T> c);

//wypełnianie listy zadanym obiektem
<T> void fill(List<? super T> list, T obj);

<T> T max(Collection<? extends T> coll, Comparator<? super T> comp);
<T> T min(Collection<? extends T> coll, Comparator<? super T> comp);
```

Klasa Collections

Klasa zawiera sporo użytecznych funkcji działających na kolekcjach, jak:

```
//sortowanie listy przy pomocy podanego komparatora
<T> void sort(List<T> list, Comparator<? super T> c);

//wyszukiwanie binarne listy przy pomocy podanego komparatora
<T> int binarySearch(List<? extends T> list, T key, Comparator<? super
T> c);

//wypełnianie listy zadanym obiektem
<T> void fill(List<? super T> list, T obj);

<T> T max(Collection<? extends T> coll, Comparator<? super T> comp);
<T> T min(Collection<? extends T> coll, Comparator<? super T> comp);

//owijacz kolekcji zwracający jej niemodyfikowalny "view"
<T> Collection<T> unmodifiableCollection(Collection<? extends T> c);
```

Klasa Arrays

Podobną funkcję, ale w odniesieniu do tablic pełni klasa `Arrays`. Również zawiera funkcje sortujące, np.

```
<T> void sort(T[] a, Comparator<? super T> c);
```

Klasa Arrays

Podobną funkcję, ale w odniesieniu do tablic pełni klasa `Arrays`. Również zawiera funkcje sortujące, np.

```
<T> void sort(T[] a, Comparator<? super T> c);
```

Interfejs Comparator

```
public interface Comparator<T> {
```

Klasa Arrays

Podobną funkcję, ale w odniesieniu do tablic pełni klasa `Arrays`. Również zawiera funkcje sortujące, np.

```
<T> void sort(T[] a, Comparator<? super T> c);
```

Interfejs Comparator

```
public interface Comparator<T> {
  int compare(T o1, T o2);
```

Klasa Arrays

Podobną funkcję, ale w odniesieniu do tablic pełni klasa `Arrays`. Również zawiera funkcje sortujące, np.

```
<T> void sort(T[] a, Comparator<? super T> c);
```

Interfejs Comparator

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  boolean equals(Object obj);
//uwaga: ta metoda jest już zaimplementowana w klasie Object
```

Klasa Arrays

Podobną funkcję, ale w odniesieniu do tablic pełni klasa `Arrays`. Również zawiera funkcje sortujące, np.

```
<T> void sort(T[] a, Comparator<? super T> c);
```

Interfejs Comparator

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  boolean equals(Object obj);
//uwaga: ta metoda jest już zaimplementowana w klasie Object
}
```

Kompilacja

- Kod źródłowy programu znajduje się w plikach `*.java`

Kompilacja

- Kod źródłowy programu znajduje się w plikach `*.java`
- Kompilujemy poleceniem `javac NazwaKlasy.java`

Kompilacja

- Kod źródłowy programu znajduje się w plikach `*.java`
- Kompilujemy poleceniem `javac NazwaKlasy.java`
- Powstaje zbiór plików `*.class`, po jednym dla klasy.

Kompilacja

- Kod źródłowy programu znajduje się w plikach `*.java`
- Kompilujemy poleceniem `javac NazwaKlasy.java`
- Powstaje zbiór plików `*.class`, po jednym dla klasy.

Kompilacja

- Kod źródłowy programu znajduje się w plikach `*.java`
- Kompilujemy poleceniem `javac NazwaKlasy.java`
- Powstaje zbiór plików `*.class`, po jednym dla klasy.

Jednostki kompilacji

Jeden plik `.java` nazywamy jednostką kompilacji.
Wewnątrz takiej jednostki może znajdować się co najwyżej jedna klasa publiczna (której nazwa musi być taka sama jak nazwa pliku `.java`).

Zarządzanie klasami

Pakiety

Wiele jednostek kompilacji możemy połączyć w pakiet.
Robimy to, wstawiając na początku pliku instrukcję:

```
package nazwapakietu
```

Zarządzanie klasami

Pakiety

Wiele jednostek kompilacji możemy połączyć w pakiet.

Robimy to, wstawiając na początku pliku instrukcję:

```
package nazwapakietu
```

Inne pakiety możemy importować za pomocą instrukcji:

```
import nazwapakietu.*
```

Zarządzanie klasami

Pakiety

Wiele jednostek kompilacji możemy połączyć w pakiet.
Robimy to, wstawiając na początku pliku instrukcję:

```
package nazwapakietu
```

Inne pakiety możemy importować za pomocą instrukcji:

```
import nazwapakietu.*
```

Pojedyncze klasy z innych pakietów importujemy poleceniem

```
import nazwapakietu.NazwaKlasy
```

Zarządzanie klasami

Pakiety

Wiele jednostek kompilacji możemy połączyć w pakiet.
Robimy to, wstawiając na początku pliku instrukcję:

```
package nazwapakietu
```

Inne pakiety możemy importować za pomocą instrukcji:

```
import nazwapakietu.*
```

Pojedyncze klasy z innych pakietów importujemy poleceniem

```
import nazwapakietu.NazwaKlasy
```

Pliki Jar

Nasz program to jeden lub więcej plików `.class`.
Możemy je spakować za pomocą Java'owego archiwizera `jar` do pliku o roszerzeniu `.jar` i używać zamiast plików `.class`.

Uruchamianie

Uruchamianie

Aby uruchomić program należy użyć polecenia:

```
java [pakiet.]NazwaKlasy
```

Uruchamianie

Uruchamianie

Aby uruchomić program należy użyć polecenia:

```
java [pakiet.]NazwaKlasy
```

gdzie `pakiet` określa ścieżkę (rozdzieloną kropkami) do pliku `NazwaKlasy.class` względem bieżącego katalogu lub innego, określonego przez zmienną środowiskową `CLASSPATH`.

Uruchamianie

Uruchamianie

Aby uruchomić program należy użyć polecenia:

```
java [pakiet.]NazwaKlasy
```

gdzie `pakiet` określa ścieżkę (rozdzieloną kropkami) do pliku `NazwaKlasy.class` względem bieżącego katalogu lub innego, określonego przez zmienną środowiskową `CLASSPATH`.

main( )

Uruchamiana klasa musi zawierać publiczną i statyczną metodę main:

```
public static void main(String[] arg);
```

- http://java.sun.com/docs/books/tutorial/index.html - Rozbudowany tutorial.
- http://java.sun.com/docs/books/tutorial/uiswing/ - Część poświęcona Swing.
- http://java.sun.com/reference/api/index.html - specyfikacje API
- http://java.sun.com/docs/books/jls/index.html - The Java Language Specification, Third Edition (html, pdf)
- pl.comp.lang.java - polska grupa dyskusyjna poświęcona Javie
- http://www.eclipse.org - dobre środowisko do Javy.