Конспект лекцій з дисципліни „Конструювання програмного забезпечення”

Поняття конструювання

Людство в ході свого технічного і культурного розвитку виробило багато ідей. Ці ідеї використовуються при конструюванні різноманітних виробів: засобів заготівлі продуктів харчування (від натурального виробництва до автоматичних виробничих ліній), будинків (від глиняних хаток з солом’яною стріхою до споруд з сучасних теплоізоляційних матеріалів), транспортних засобів (від фіри до космічного корабля), засобів комунікації (від телеграфу до сучасного смартфона), засобів обчислення (від рахівниці до суперкомп’ютера), творів мистецтва (від театру до сучасних фільмів з комп’ютерною графікою).

Ідеї розвиваються від винаходу (комерційної таємниці) до загальновідомого поняття. Таким чином колишні винаходи стають надбанням всього людства. Завдяки цьому, приступаючи до конструювання якогось виробу, не доводиться наново винаходити колесо чи велосипед.

Загальновідомі поняття описуються за допомогою стандартизованих мов. Наприклад, для опису математичних тотожностей використовується мова математичних символів, для опису електричних кіл використовується мова схем, для опису музичних творів використовується нотний стан.

Подібним чином розвиваються ідеї у конструюванні програмного забезпечення: від винаходу до стандартизованих понять. Для опису стандартизованих понять в галузі конструювання програмного забезпечення використовують мову UML. Загальновідомі поняття в конструюванні програмного забезпечення називаються *зразками проектування* (design patterns). Зі зразками проектування Ви мали змогу ознайомитися підчас вивчення дисципліни „Проектування програмного забезпечення” на другому курсі. При вивченні ж дисципліни „Конструювання програмного забезпечення” ми не будемо наново вивчати зразки проектування, а будемо ними користуватися практично.

Бути кваліфікованим конструктором програмного забезпечення означає, що ви маєте бути обізнані в обєктно-орієнтованих аспектах Конструювання програмного забезпечення. Ви маєте глибоко опанувати ієрархією успадкування, сила поліморфізму має струмувати через вас, зв’язки і розв’язання має стати вашою другою природою, композиція має бути вашим хлібом і маслом. Ця частина підготує вас до всіх обєктно-орієнтованих тем та питань, з якими ви стикнетесь на екзамені. Ми чули про багатьох досвідчених програмістів Конструювання програмного забезпечення, які не були добре обізнані в обєктно-орієтнованих інструментах, які забезпечує Конструювання програмного забезпечення, тож почнемо спочатку.

Інкапсуляція (об’єкт 5.1)

Уявіть, що ви написали код для класу, а інша дюжина програмістів з вашої компанії написали програми для використання цього класу. Тепер уявіть, що

вам не сподобалось, як себе веде клас, оскільки деякі з екземплярів змінних були задані (іншими програмістами їхнім кодом) новими значеннями, які ви не одобряєте. Їх код спричинав помилки в вашому коді (розслабтесь, це тільки припущення). Що ж, це програма на Яві, тож ви можете зробити нову версію класу, яку вони будуть переносити в свої програми без змін в коду.
Цей сценарій висвітлює 2 обіцянки(переваги) об'єктної орієнтації – гнучність і можливість зберігання. Але ці переваги не приходять автоматично. Ви маєте щось зробити. Ви маєте написати ваші класи і код способом, який підтримує гнучкість і можливість зберігання. То як конструювання програмного забезпечення підтримує ОО? Вона не може розробляти код замість вам. Для прикладу, уявіть, що ви зробили клас з публічними змінними екземплярів, а інші програмісти змінили їх напряму, як демонструє наступний код:

```
public class BadOO {
  public int size;
  public int weight;
  ...
}
public class ExploitBadOO {
  public static void main (String [] args) {
    BadOO b = new BadOO();
    b.size = -5; // Legal but bad!!
  }
}
```

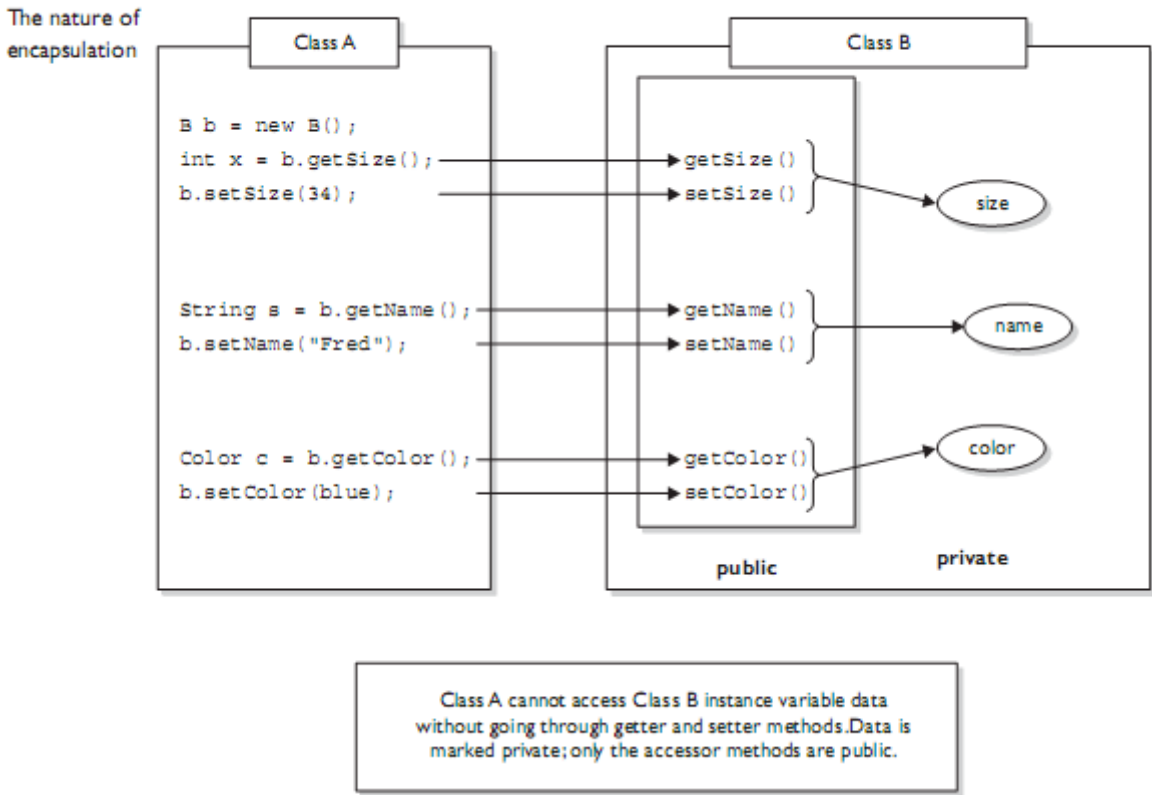
А тепер у вас проблеми. Як ви збираєтесь змінити клас так, щоб ви могли керувати ситуаціями, коли хтось міняє значення змінної size на значення, яке викликає проблеми? Єдиний ваш вибір – повернутись і написати код методу, для задання size (наприклад метод setSize(int a), а потім захистити змінну, скажімо приватним модифікатором доступу. Але як тільки ви зробите такі зміни в коді, ви забороните будь-які інші!
Можливість робити зміни в коді виконання без переривання коду для інших, хто використовує ваш код – це ключова перевага інкапсуляції. Ви хочете заховати деталі виконання за публічним програмним інтерфейсом. Під інтерфейсом ми розуміємо набір доступних методів, які ваш код робить доступними до виклику для інших кодів – іншими словами, API вашого коду. Сковуванням деталей виконання, ви можете переробляти код методу (звичайно ж і зміну шляху використання змінних вашим класом) без примушування до змін коду, що викликає ваш змінений метод.
Якщо ви хочете збереження, гнучкість і можливість розширення (а ви безперечно хочете), ваша розробка має включати інкапсуляцію. Як ви зробите її?

- захистивши змінні екземплярів (модифікаторами доступу)
- Зробивши методи публічного доступу, і примусивши тих, що викликають ваш код, працювати з методами, а не отримувати доступ до змінних напряму.

- Використавши конвенції конструювання програмного забезпеченняbean для методів: set<someProperty> і get<someProperty>

Схема 2-1 ілюструє ідею того, як інкапсуляція примушує тих, що викликають код, слідувати методами, а не міняти змінні напряму.

FIGURE 2-1



Class A cannot access Class B instance variable data without going through getter and setter methods.Data is marked private; only the accessor methods are public.

Ми викликаємо метода читання і запису хоча дехто використовує терміни аксесори і мутатори (Особисто нам не подобається слово mutate)/
Незалежно як ви їх викличете, ц методи, які інші програмісти мають використовувати для доступу до змінних екземплярів. Вони виглядають просто і ви наперно будете завжди їх використовувати:

```
public class Box {
  // protect the instance variable; only an instance
  // of Box can access it
  private int size;
  // Provide public getters and setters
  public int getSize() {
    return size;
  }
  public void setSize(int newSize) {
    size = newSize;
  }
}
```

Почекайте… Наскільки корисним був попередній код? Він навіть не здійснює ніякої обробки. Яку перевагу дадуть нам методи доступу і запису, які не додають функціональності? Ви зміните свою думку пізніше,

коли додасте більше коду в методи без переривання вашого API. Навіть якщо сьогодні ви думаєте, що вам не потрібно здійснювати чи обробляти дані, хороша розробка ОО показує, що у вас є плани на майбутнє. Щоб убезпечитись, заставте код виклику пройти ваші методи заміть прямого доступу до змінних. Завжди. Тоді ви звільнитесь від переробки виконання пізніше, без ризику переслідування дюжини програмістів, які знають де ви живете.

Подивіться на код, який має запитувати про поведінку метод, коли проблема в відсутності інкапсуляції. Подивіться на наступний приклад, поглянемо чи зможете ви визначити, що діється:

```
class Foo {
  public int left = 9;
  public int right = 3;
  public void setLeft(int leftNum) {
    left = leftNum;
    right = leftNum/3;
  }
  // lots of complex test code here
}
```

Тепер задайте собі питання: чи значення right завжди має бути третиною left? Виглядає, що так, доки ви не зрозумієте, що користувачі класу Foo можуть не використовувати метод setleft()! Вони можуть просто отримати доступ до значень змінних і змінити їх на будь-яке цілочислове значення.

Поведінка is-a, has-a (об'єкт 5.5)

Успадкування в конструювання програмного забезпечення можна побачити всюди. Можна сказати, що майже (майже?) неможливо написати навіть найменшу програму без використання успадкування. Щоб розкрити цю тему, ми збираємось використати оператор instanceof, який ми детальніше обговоримо в розділі 4. Зараз просто запам'ятайте, що nstance повертає true, коли змінна, яку ми тестуєм, відповідає типу, з яким ми порівнюємо. Цей код:

```
class Test {
  public static void main(String [] args) {
    Test t1 = new Test();
    Test t2 = new Test();
    if (!t1.equals(t2))
      System.out.println("they're not equal");
    if (t1 instanceof Object)
      System.out.println("t1's an Object");
  }
}
```

Дасть на виході наступне:

```
they're not equal
```

t1's an Object

Звідки ж прийшов цей метод equals? Змінна посилання t1 є типу test, і в класі test немає методу equals. Чи є? тестування другого if питає, чи t1 є екземпляром класу Object, і оскільки воно є (більше про це скоро), тест if дістає ствердну відповідь.

Почекайте, як t1 може бути екземпляром типу Object, ми тільки що сказали, що воно типу test? Я впевнений, ви на шляху перед нами, але виходить, що кожен клас в Конструювання програмного забезпечення є підкласом Object (крім самого класу Object). Іншими словами, всі класи що ви коли-небудь використовували чи писали будуть успадковані класом Object. Ви завжди будете мати метод equals, метод clone, notify, wait та інші готовими до використання. Коли б ви не створили клас, він автоматично успадкує всі методи класу Object.

Чому? Глянемо не метод equals для екземпляру. Творці Ява правильно визначили, що це буде звично для програмістів хотіти перевірити еквівалентність їхніх класів. Якщо клас Object не мав би методу equals, ви би мусили самі його написати. Ви і кожен програміст Конструювання програмного забезпечення. Цей метод був би успадкований мільярди разів. (Чесно кажучи, даний метод був би перевизначений мільярди разів, але не будемо забігати вперед).

Для екзамену вам потрібно знати що ви можете створювати спадкові зв'язки в конструювання програмного забезпечення з допомогою розширення класу. Також важливо зрозуміти, що 2 звичні причини використання успадкування

- забезпечення повторного використання коду
- використання поліморфізму.

Почнемо з повторного використання. Звична розробка – створення загального класу з можливістю створення більш спеціалізованих підкласів, що успадковують з загального класу. Для прикладу:

```
class GameShape {
  public void displayShape() {
    System.out.println("displaying shape");
  }
  // more code
}
class PlayerPiece extends GameShape {
  public void movePiece() {
    System.out.println("moving game piece");
  }
  // more code
}
public class TestShapes {
  public static void main (String[] args) {
    PlayerPiece shape = new PlayerPiece();
```

```
    shape.displayShape();
    shape.movePiece();
  }
}
```
Вивід:
```
displaying shape
moving game piece
```

Зауважте, що клас PlayingPiece успадковує узагальнюючий метод display() з менш спеціалізованого класу GameShape, а також додає власний метод, movePiece(). Повторне використання коду через успадкування означає, що методи з загальною функціональністю, що можуть застосовувати до багатьох різних форм гри – не мають виконуватись повторно. Це означає, що всім спеціалізованим підкласам gameShape гарантується наявність можливостей більш загального надкласу. Вам н потрібно переписувати код display() в кожному спеціалізованому коді он0лайн гри.
Але ви знали це. Ви пізнали біль дублікації коду, коли ви робили зміни в одному місці і мусили прокручувати всі місця, де зустрічалась така ділянка коду.
Друге використання успадкування це дозвіл вашим класам давати доступ поліморфічно – можливість, що забезпечується так само добре інтерфейсами, але ми доберемось до цього за хвилинку. Скажімо ви маєте клас GameLauncher, який хоче повтяти список різних типів об'єктів GameShape і викликати display() на кожен з них. Коли ви напишете цей клас, ви навіть не знаєте можливий вид Gameshape, який кожен може написати пізніше. І ви впевнені, що вам не потрібно міняти код, оскільки хтось захоче побудувати форму Dice шість місяців потому.
Хороша річ в поліморфізмі – ці те, що ви можете задавати будь-який підклас GameShape як GameShape. Іншими словами, ви можете написати код в вашому класі GameLauncher, що каже «мені пофіг якого типу ви об'єкт, допоки ви розширюєте GameShape. Наскільки я обізнаний, якщо ви розширюєте GameShape, вам отримуєте метод display(), тож я можу викликати його».
Уявіть, що зараз ви маєте 2 спеціалізовані підкласи, що розширюють більш загальні GameShape class, PlayerPiece та TilePiece:
```
class GameShape {
  public void displayShape() {
    System.out.println("displaying shape");
  }
  // more code
}
class PlayerPiece extends GameShape {
  public void movePiece() {
    System.out.println("moving game piece");
  }
```

```
  // more code
}
class TilePiece extends GameShape {
   public void getAdjacent() {
     System.out.println("getting adjacent tiles");
   }
   // more code
}
```

Тепер уявіть, що тестовий клас має метод з оголошеним типом аргумента **,** що означає, що він він приймає будь-який вид GameShape. Цей код

```
public class TestShapes {
  public static void main (String[] args) {
     PlayerPiece player = new PlayerPiece();
     TilePiece tile = new TilePiece();
     doShapes(player);
     doShapes(tile);
  }
  public static void doShapes(GameShape shape) {
    shape.displayShape();
  }
}
```

Дає на виході наступне.

```
displaying shape
displaying shape
```

Ключовим моментом тут є те, що метод doShapes(), оголошенй з допомогою аргумента GameShape може бути пройдений будь-яким підтипом (в цьому випадку підкласом) GameShape. Таким чином метод може виликати будь-який метод GameShape без впливу на час виконання типу класу об’єкта, що проходить метод. Хоча є складності.. Метод doShapes() знає тільки, що об’єкти мають тип GameShape, тож так оголошується параметр. А використовуючи змінну посилання, оголошену як GameShape – незалежно від того, чи змінна є параметром методу, локальною змінною чи змінною екземпляру – означає, що методи GameShape можуть викликатись на них. Метод, які ви можете викликати з допомогою посилання, повністю залежать від оголошеного типу змінної, неважливо що є об’єктом, на який посилається змінна. Це означає, що ви не можете використовувати змінну GameShape для виклику, скажімо методу getAdjacent() навіть, якщо обєкт має тип TilePiece. (Ми побачимо це знову коли поглянемо на інтерфейси).

Зв’язки IS-A та HAS-A

Для екзамену вам потрібно вміти глянути на код і визначити, коли він демонструє зв’язки IS-A та HAS-A. Правила прості, тож це буде одна з частин, де відповіді будуть прості і лаконічні майже без мозкового навантаження.

IS-A

В ОО, концепція IS-A базована на успадкування класів або виконання інтерфейсів. IS-A – це шлях сказати «ця штука є типом тої штуки». Для прикладу мустанг тип коня, тож в ОО ми можемо сказати Мустанг IS-A Кінь». Subaru IS-A Car. Broccoli IS-A Vegetable (не смішно, але для кількості нормально). Ви виражаєте зв'язок IS-A з допомогою ключового слова extends (для успадкування класів) і implements (для виконання інтерфейсів).

```
public class Car {
  // Cool Car code goes here
}
public class Subaru extends Car {
   // Important Subaru-specific stuff goes here
   // Don't forget Subaru inherits accessible Car members which
   // can include both methods and variables.
}
```

Car це тип Vehicle, тож дерево успадкування стартує з Vehicle як показано тут:

```
public class Vehicle { ... }
public class Car extends Vehicle { ... }
public class Subaru extends Car { ... }
```

В термінології ОО ви говорите наступне

Vehicle є надкласом Car.
Car є надкласом Vehicle.
Car є надкласом Subaru.
Subaru є надкласом Vehicle.
Car спадковий для Vehicle.
Subaru спадковий для Vehicle і Car.
Subaru походить з Car.
Car походить з Vehicle.
Subaru походить з Vehicle.
Subaru це підтип Vehicle і Car.

Повертаючись до наших зв’язків, наступні твердження правильні:

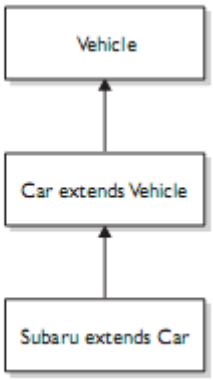
"Car extends Vehicle" означає "Car IS-A Vehicle."
"Subaru extends Car" означає "Subaru IS-A Car."

І ми також кажемо:

"Subaru IS-A Vehicle" тому, що клас має бути типом всього, що стоїть вище в дереві спадковості. Якщо вираз (Foo instanceof Bar) є true, тоді клас Foo IS-A Bar, навіть якщо він напряму не розширює Bar, але замість цього розширює якийсь інший клас що є підкласом bar. Схема 2-2 ілюструє дерево успадкування для Vehicle, Car, і Subaru. Стрілки йдуть від підкласу до надкласу. Іншими словами, стрілка класу рухається до класу, який він розширює.

FIGURE 2-2

Inheritance tree for Vehicle, Car, Subaru



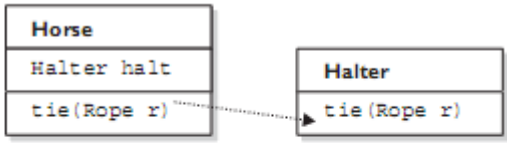
Зв'язки HAS-A базовані на використанні більш ніж на спадковості. Іншими словами клас A HAS-AB якщо код в класі I має послиаання на екземпляр класу B. Для прикладу ви можете сказати наступне.
A Horse IS-A Animal. A Horse HAS-A Halter.
Код буде виглядати наступним чином

```
public class Animal { }
public class Horse extends Animal {
  private Halter myHalter;
}
```

В коді вище, клас horse може мати змінну екземпляру типу Halter, тож ви можете сказати, що "Horse HAS-A Halter." Іншими словами, Horse має посилання на Halter. Код Horse використовує посилання на Halter, щоб викликати методи на Halter, і отримувати поведінку Halter без наявності методів, що належать Halter, в класі Horse. Схема 2-3 ілюструє зв'язки HAS-A між Horse і Halter.

FIGURE 2-3

HAS-A relationship between Horse and Halter



Horse class has a Halter,because Horse declares an instance variable of type Halter. When code invokes `tie()` on a Horse instance, the Horse invokes `tie()` on the Horse object's Halter instance variable.

Зв'язки HAS-A дозволяє вам розробляти класи які відповідають хорошому тону ОО, не маючи монолітних класів, що роблять зовсім різні речі. Класи (та результати їх роботи) мають бути спеціалістами. Як каже наш друг Ендрю, «спеціалізовані класи дозволяють видаляти баги». Чим більше спеціалізований клас, тим більша ймовірність, що ви використаєте його в інших додатках. Якщо ви вставите весь код, що посилається на halter прямо в клас Horse, ви закінчите дублювати код в класі Cow, класі UnpaidIntern та в інший класах, що потребують поведінки Halter. Тримаючи код Halter в окремому класі, ви даєте шанс використанню класу Halter в інших додатках.

З заняття в класі.
Обєктно – орієнтовані розробки

IS-A і HAS-A зв’язки та інкапсуляція це тільки вершина айсберга ОО. Багато книжок і наукових робіт присвячені цій темі. Причина проста: гроші. Розробка софту оцінюється в 10 разів дорожче для програм з бідним дизайнос. Я бачив наслідки бідного дизайну, і можу завірити, що ця оцінка не дуже перебільшена.
Навіть найкращі розробники ОО роблять помилки. Це важк візуалізувати зв’язки між сотнями чи навіть тисячами класів. Коли помилки знаходяться в процесі виконання фази проекту, багато коду має бути переписано, що може навіть призвести до початку писання з нуля. Індустрія софту вирішила допомогти розробнику. Мови візуального моделювання об’єктів, як наприклад (UML), дозволяє розробникам розробити і легко модифікувати класи без написання коду, тому що компоненти представляються графічно. Це дозволяє розробнику створити карту зв’язків класів і дозволяє розпізнавати помилки перед початком кодування. Інша інновація в розробці ОО – це розробка шаблонів.
Дизайнери зауважили, що багато дизайнів ОО повторюються від проекту до проекту, і було в корисно використовувати ті ж дизайни, щоб уникнути потенційно можливі нові помилки. Обєктно-орієнтовані дизайнери почали ділитись цими дизайнами, тепер є багато шаблонів як в Інтернеті, так і в книжках.
Хоча складання екзамену сертифікації не вимагає від вас розуміти розробку ОО, ця інформація допоможе вам розуміти, чому автори тестів захотіли включити інкапсуляцію, IS-A, HAS-A зв’язки на екзамені.
—Jonathan Meeks, Sun Certified Конструювання програмного забезпечення Programmer

Користувачі класу Horse (код, що викликає методи на екземплярі Horse), думають що клас Horse має поведінку halter. Клас Horse має мати метод tie(LeadRope rope), для прикладу. Користувачі Horse не мають знати, що вони викликають метод tie(), об’єкт horse обертається і звертається до класу Halter з допомогою виклику myHalter.tie(rope). Сценарій, що описувався, виглядає так:

```
public class Horse extends Animal {
  private Halter myHalter = new Halter();
  public void tie(LeadRope rope) {
    myHalter.tie(rope);  // Delegate tie behavior to the
                         // Halter object
  }
}
public class Halter {
  public void tie(LeadRope aRope) {
    // Do the actual tie work here
  }
}
```

В ОО, ми не хочемо, щоб ті, хто викликають, хвилювались в якому об'єкті якого класу виконується робота. Щоб це здійснити, клас Horse ховає виконання від користувачів. Користувачі звертаються до об'єкту Horse, щоб виконати щось, і Horse зробить це чи попросить когось зробити це. Для викликача, це виглядає, ніби Horse все робить сам. Користувачі навіть не мають знати про такий клас як halter.

Поліморфізм (Об'єкт 5.2)

Пам'ятайте, кожен об'єкт Конструювання програмного забезпечення, що пройшов більше від одного тесту IS-A може вважатись поліморфним. Оскільки об'єкти, всі об'єкти є поліморфними, якщо пройшли один тест IS-A, для власного типу і для класу Object.

Пам'ятайте, що є тільки один шлях доступу до об'єкта – через змінну посилання, і є декілька речей, які треба запам'ятати щодо посилання.

- змінна посилання може бути тільки одного типу, і цей тип не може мінятись (хоча об'єкт посилання може).
- Посилання це змінна, тож вона може перевизначатись іншими об'єктами (якщо вона не є кінцевою).
- Тип змінної посилання визначає методи, які можуть викликатись на об'єкті, на який посилається змінна.
- Змінна посилання може посилатись на будь-який об'єкт того ж типу, що й змінна, або – що важливо – будь-якого підтипу оголошеного типу!
- Змінна посилання може бути оголошена як тип класу чи тип інтерфейсу. Якщо змінна оголошується як тип інтерфейсу, вона посилається на будь-який об'єкт будь-якого класу, що виконує інтерфейс.

Раніше ми створили клас GameShape що був розширений двома іншими класами, PlayerPiece та TilePiece. Тепер уявіть, що вм треба анімувати деякі з форм в грі. Але не всі з них можуть анімуватись, то що ж вам робити з успадкуванням класів.

Чи можемо ми створити метод animate, і вибрати тільки деякі підкласи надкласу GameShape? Якщо можемо, то PlayerPiece, наприклад розширить 2 класи GameShape Animatable, тоді як TilePiece тільки GameShape. Але ні, це не буде працювати! Конструювання програмного забезпечення підтримує тільки одне успадкування! Це означає, що клас може мати тільки один безпосередній надклас. Іншими словами, якщо PlayerPiece – це клас, то не можна зробити щось подібне:

```
class PlayerPiece extends GameShape, Animatable { // NO!
  // more code
}
```

Клас не може розширювати більше від одного класу. Це означає один зв'язок на клас. Клас може мати багато предків ,наприклад клас B може

розширювати клас А, Клас С – Клас В і так далі. Тож кожен клас може мати багато класів вище по спадковому дереву, але це не те саме, що розширення класом двох надкласів прямим зв'язком.

Деякі мови (як C++) дозволяють класу розширювати більше одного класу. Ця властивість знана як «множинне успадкування». Причина чому творці Ява вирішили не дозволяти множинне успадкування тому, що воно може бути дуже безладним. Проблема коли клас розширив два інші класи, і обидва цих надкласи, скажемо, мають метод doStuff(), котрий із них успадкується? Це може перейти в сценарій, відомий як «смертельний діамант смерті», через форму діаграми класів, що може бути створена при множинному успадкуванні. Діамант формується, кол класи В і С розширюють А, і також В і С успадковують метод з А. Якщо клас Д розширює В і С, і обидва вони перевизначать метод в А, клас Д, теоретично успадкує 2 різні виконання одного методу. Намальована як діаграма класу, форма чотирьох класів виглядає як діамант.

Тож якщо це не працює, що ще можна зробити? Ви можете просто вставити код animate() в GameShape, і потім закрити метод в класі, що не можуть бути анімовані. Але це поганий вибір розробки через багато причин, включаючи те, що в ньому більше загроз помилки, вони роблять клас GameShape менш здатним до зв'язку(про це за хвилину), і це означає, що GameShape API «рекламує», що всі підкласи можуть бути анімовані, тоді як тільки деякі підкласи зможуть запускати метод animate().
То що ж вам ще робити? Ви знаєте відповідь – створіть інтерфейс Animatable, і майте тільки підкласи GameShape, які можуть бути анімовані, що виконають цей інтерфейс. Ось інтерфейс:

```
public interface Animatable {
  public void animate();
}
```

А ось публічний клас Playerpiece, модифікований для виконання інтерфейсу:

```
class PlayerPiece extends GameShape implements Animatable {
  public void movePiece() {
    System.out.println("moving game piece");
  }
  public void animate() {
    System.out.println("animating...");
  }
  // more code
}
```

Тож тепер ми маємо PlayerPiece Що проходить тест IS-A для класу GameShape та інтерфейсу Animatable. Це означає, що PlayerPiece може

бути поліморфічно однією з чотирьох речей в будь-який час, в залежності від типу змінної посилання:

- Object (успадкування від Object)
- GameShape (оскільки PlayerPiece розширює GameShape)
- PlayerPiece (оскільки це він і є)
- Animatable (оскільки PlayerPiece виконує Animatable)

Наступний код – правильний. Подивіться на нього уважно.

```
PlayerPiece player = new PlayerPiece();
Object o = player;
GameShape shape = player;
Animatable mover = player;
```

Є тільки один об'єкт – екземпляр типу playerPiece – та я 4 різних типи змінних посилання, всі посилаються на один об'єкт в памяті. Питання: котрі з попередніх змінних посилання можуть викликати метод displayShape(). Відповідь 2 з 4 оголошень.

Пам'ятайте, що виклики методів дозволені компілятором базовані на оголошеному типу посилань, незалежно від типу об'єкту. Тож дивлячись на 4 типи посилань - Object, GameShape, PlayerPiece, and Animatable,котрі з цих типів знають про метод displayShape().

Ви вгадали - класи GameShape та PlayerPiece мають метод displayShape() (що відомо компілятору), тож кожен з цих типів посилання можуть використовуватись для виклику displayShape(). Пам'ятайте, що дя компілятора PlayerPiece IS-A GameShape, тож компілятор каже "Я бачу, що оголошений тип PlayerPiece, і оскільки PlayerPiece розширює GameShape, це означає PlayerPiece успадкував метод displayShape(). Тому PlayerPiece можуть використовуватись для виклику методу displayShape()."

Які методи можуть бути викликані коли об'єкт PlayerPiece посилається на використання посилання оголошеного як тип Animatable? Тільки метод animate(). Звичайно круто, що кожен клас з будь-якого дерева успадкування може використовувати Animatable, тож це означає, що якщо ви маєте метод з аргументом, оголошеним як тип Animatable, ви можете виконувати об'єкти PlayerPiece SpinningLogo та інші, які екземпляром класу, що виконує Animatable. І ви можете використати цей параметр (типу Animatable) для виклику метод animate(), але не метод displayShape() (який його може навіть не мати), або будь-який інший, що базується на типі посилання. Компілятор завжди знає, що ви можете викликати методи класу Object на будь-якому об'єкті, тож вони безпечно для виклику незалежно від посилання – клас інтерфейсу – використовуються для посилання на об'єкт.

Вагомим зауваженням буде те, що компілятор знає тільки про оголошений тип посилання, Конструювання програмного забезпечення Virtual Machine (JVM) під час виконання знає, де знаходиться об'єкт. І це означає, що навіть якщо метод displayShape() об'єкта PlayerPiece викликається з використанням

змінної посилання GameShape, якщо PlayerPiece пере визначає метод DisplayShape(), JVM викличе версію PlayerPiece! JVM дивиться на реальний об'єкт в іншому кінці посилання, бачить що він перевизначив метод оголошеного типу змінно посилання і викликає метод на дійсному класі об'єкта. Але ще одна річ для запам'ятовування:

Виклик поліморфічного методу застосовуються тільки до методів екземплярів. Ви можете завжди посилатись на об'єкт з більш загальним типом змінною посилання (надкласу чи інтерфейсу), але в час запуску, єдині речі, які вибираються динамічно побудовані на дійсному об'єкті (а не на типі посилання) є методами екземплярів. Не статичні методи. Не змінні. Тільки перевизначені методи екземплярів базовані на динамічному виклику на реальному типі об'єкта.

Оскільки це визначення залежить від чіткого розуміння пере визначення, і знання різниці між статичними методами і методами екземплярів, ви розглянемо їх далі.

Перевизначення/перезавантаження

Перевизначені методи

Кожен раз, коли ви маєте клас, що успадковує метод з надкласу, ви маєте перевизначити метод (якщо він не позначений final). Ключова перевага перевизначення це можливість визначати поведінку специфічну для конкретного типу підкласу. Наступний приклад демонструє підклас Horse класу Animal, що пере визначає версію Animal методу eat():

```
public class Animal {
  public void eat() {
    System.out.println("Generic Animal Eating Generically");
  }
}
class Horse extends Animal {
  public void eat() {
     System.out.println("Horse eating hay, oats, "
                  + "and horse treats");
  }
}
```

Для абстрактних методів, які ви успадкували з надкласу, ви не маєте вибору. Ви маєте виконати метод в підкласу якщо він теж не абстрактний. Абстрактні метод мають виконуватись дійсним підкласом, але це виглядає ніби дійсний клас пере визначає абстрактні методи надкласу. Тож ви можете думати про абстрактні методи, які ви заставили перевизначитись.

Автор класу Animal може вирішити що для потреб поліморфізму, всі підтипи Animal можуть мати метод eat(), визначений в унікальний, специфічний спосіб. Пліморфічно, коли один має посилання та Animal що посилається не на екземпляр Animal, а на екземпляр підкласу Animal,

викликач не зможе викликати eat() на посиланні Animal, але дійсний обжкт запуску (скажімо екземпляр Horse) запустить власний специфічний метод eat(). Позначання методу eat() як абстрактного це спосіб програміста Animal сказати всім розробникам підкласів «Немає сенсу для вашого підтипу використовувати загальний метод eat(), тож прийдіть з своїм виконанням методу eat()!» Не абстрактний приклад використання поліморфізму виглядає так:

```
public class TestAnimals {
  public static void main (String [] args) {
    Animal a = new Animal();
    Animal b = new Horse();  //Animal ref, but a Horse object
    a.eat(); // Runs the Animal version of eat()
    b.eat(); // Runs the Horse version of eat()
  }
}
class Animal {
  public void eat() {
  System.out.println("Generic Animal Eating Generically");
  }
}
class Horse extends Animal {
  public void eat() {
    System.out.println("Horse eating hay, oats, "
                  + "and horse treats");
   }
   public void buck() { }
}
```

В попередньому коді, тестовий клас використовує посилання для виклику методу на об’єкті Horse. Пам’ятайте, що компілятор дозволить тільки методи в класі Animal для виклику при посиланні на Animal. Наступне не буде правильним:

```
Animal c = new Horse();
c.buck();  // Can't invoke buck();
        // Animal class doesn't have that method
```

Для реітерації, компілятор дивиться тільки на тип посилання, а не на тип екземпляру. Поліморфізм дозволяє вам використовувати більш абстрактний надтип (включаючи інтерфейс) для посилання на один з підтипів (включаючи виконувачі інтерфейсу).

Перевизначення методу не може мати більш строгого модифікатора доступу, ніж є в метода, що перевизначається (для прикладу, не можна публічний метод зробити захищеним.) Подумайте про це: якщо клас Animal рекламує публічний метод eat() і вхтось з посиланням Animal (іншими слвами, посилання з типом animal), цей хтось вирішить, що безпечно

викликати eat() на посиланні незалежно на що посилається змінна посилання фактично. Якщо підкласу дозволено було зайти в метод і змінити модифікатор доступу при перевизначенні методу, то скоро під час виконання – коли JVM викликає справжні версії методу об'єкту, а не версії типу посилання – програма помре наглою смертю. Давайте модифікуємо поліморфічний приклад, який ми бачили раніше:

```
public class TestAnimals {
  public static void main (String [] args) {
    Animal a = new Animal();
    Animal b = new Horse();  //Animal ref, but a Horse object
    a.eat(); // Runs the Animal version of eat()
    b.eat(); // Runs the Horse version of eat()
  }
}
class Animal {
  public void eat() {
    System.out.println("Generic Animal Eating Generically");
  }
}
class Horse extends Animal {
  private void eat() {  // whoa! - it's private!
    System.out.println("Horse eating hay, oats, "
                       + "and horse treats");
  }
}
```

Якщо код скомплілюється (що не станеться), провал наступить під час запуску:

```
Animal b = new Horse();  // Animal ref, but a Horse
                         // object , so far so good
b.eat();                 // Meltdown at runtime!
```

Змінна b типу Animal, який має публічний метод eat(). Але пам'ятаєте – під час виконання конструювання програмного забезпечення використовує виклик методу для динамічного вибору дійсної версії методу, що запуститься, базований на актуальному екземплярі. Посилання Animal може завжди посилатись на екземпляр Horse, оскільки Horse IS-A(n) Animal. Що робить посилання надкласу на екземпляр підкласу можливим це те що підклас гарантовано матиме можливість робити все, робить надклас. Оскільки екземпляр Horse пере визначає успадковаі методи Animal, або просто успадковує їх, кожен з посиланням Animal на екземпляр horse може вільно викликати всі доступні методи. З цієї причини, метод, що пере визначає, має відповідати контракту надкласу.

Правила перевизначення методу наступні:

- Список аргументів мусить точно співпадати з перевизначеним методом. Якщо вони не співпадають, ви можете закінчити перевантаженим методом, який ви не включили.

- Тик повернення має бути той же, що, або підтип типу повернення, оголошеного в оригінальному перевизначеному методі надкласу (більше про це – ількома сторінками пізніше.)
- Рівень доступу не може бути більш строгим ніж до перевизначення.
- Рівень доступу Може бути менш строгим.
- Методи екземплярів можуть бути перевизначені тільки якщо вони були успадковані підкласом. Підклас в одному пакеті, що і надклас екземпляру можу перевизначати всі методи надклас, що не позначені private чи final. Підклас в різних пакетах може перевизначати тільки некінцеві методи, позначені public чи protected (оскільки захищені методи успадковуються підкласом).
- Перевизначення методу може давати будь-яке виключення виконання, незалежно від того, чи оголошує це виключення перевизначений метод. (більше в розділі 5).
- Метод, що пере визначає не має видавати виключення, що є нові чи сторонні відносно тих, які оголошені перевизначеним методом. Для прикладу, метод оголошує FileNotFoundException, то не може бути перевизначеним методом, що оголошує SQLException, Exception, чи інші виключення, якщо це не підклас FileNotFoundException.
- Метод, що пере визначає може видавати вузькі виключення. Хоч перевизначений метод «ризикує», це не означає, що метод, який пере визначає, мА такий же ризик. Він не моє оголошувати виключення, які ніколи не видадуться, незалежно від того, що оголошує перевизначений метод.
- Ви не можете перевизначати метод, позначений final. Ви не можете перевизначати статичні методи. Ми подивимось на приклад за декілька сторінок, коли обговоримо статичні методи більш детально.
- Якщо метод не може бути успадкований, ми не можете перевизначити його. Пам’ятайте, що превизначення відбувається при перевиконанні методу, який ви успадкували. Для прикладу, наступний код неправильний, і навіть якщо ви додали метод eat() в Horse, це не буде перевизначенням методу eat() класу Animal/

```
public class TestAnimals {
  public static void main (String [] args) {
    Horse h =  new Horse();
    h.eat(); // Not legal because Horse didn't inherit eat()
  }
}
class Animal {
  private void eat() {
    System.out.println("Generic Animal Eating Generically");
  }
}
class Horse extends Animal { }
```

Виклик версії надкласу перевизначеного методу.
Часто вам потрібно надати перевагу якомусь коду в версії надкласу методу, і перевизначити його, щоб спеціалізувати його поведінку. Це як сказати «запустіть версію надкласу методу, потім поверніться сюди і завершіть кодом додаткового методу підкласу.» (Зауважте, що немає вимоги, щоб версія надкласу запускалась перед кдом підкласу). Це просто зробити в коді з використанням ключового слова super, як тут:

```
public class Animal {
  public void eat() { }
  public void printYourself() {
    // Useful printing code goes here
  }
}
class Horse extends Animal {
  public void printYourself() {
    // Take advantage of Animal code, then add some more
    super.printYourself();  // Invoke the superclass
                            // (Animal) code
                            // Then do Horse-specific
                            // print work here
  }
}
```

Зауваження: використання super для вилику перевизначеного методу застовується тільки для методів екземплярів. (Пам’ятайте, статичні методи не можуть бути перевизначені).

Зауваження.
Якщо метод перевизначений, але ви використовуєте поліморфічне посилання для посилання на об’єкт з методом, що пере визначає, компілятор вирішить, що ви викликаєте версію над типу методу. Якщо версія над типу оголошує відмічене виключення, а метод, що пере визначає, компілятор все ж думає, що ви викликаєте метод, що оголошує виключення (більше в частині 5).
Подивимось на приклад:

```
class Animal {
 public void eat() throws Exception {
   // throws an Exception
 }
}
class Dog2 extends Animal {
 public void eat() { /* no Exceptions */}
 public static void main(String [] args) {
   Animal a = new Dog2();
   Dog2 d = new Dog2();
   d.eat();          // ok
```

```
    a.eat();          // compiler error -
                      // unreported exception
  }
}
```

Цей код не скомпілюється через те, що Exception оголошений на методі eat() . Це стається навіть тоді, коли використовуваний метод eat() буде версії Dog, що не передбачає виключення.

Приклади правильних неправильних перевизначень

Подивимось на перевизначення методу eat(), дане в попередній версії класу Animal.

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

Таблиця 2-1 перелічує приклади легального перевизначення методу eat() , дані в попередній версії класу Animal/

Приклад неправильного коду	Проблема з кодом
private void eat() { }	Модифікатор доступу надто строгий
public void eat() throws IOException { }	Оголошує відмічене виключення не визначене версією надкласу
public void eat(String food) { }	Правильне перевантаження, а не перевизначення, оскільки змінився список аргументі
public String eat() { }	Не перевизначення через тип повернення, не перевантаження, оскільки немає змін в списку аргументів.

Перезавантажені методи.

Ви усвідомлюєте, що перезавантажені методи роблять в розділі про ОО, але ми включили їх через одну річ, якою стурбовані нові розробники Конструювання програмного забезпечення – різницю між перезавантаженими та перевизначеними методами.

Перезавантажені методи дозволяють вам повторно використовувати імя методу в класі, але з іншими аргументами (і, можливо з іншим типом повернення). Перезавантаження методу часто означає, що ви будете більш добрішим для того, хто виликає ваші методи, оскільки ваш код дістає можливість роботи з різними типами аргументів замість примушування того хто викликає до конвертування перед викликом методу. Правила прості:

- Перезавантажені методи МУСЯТЬ міняти список аргументів.
- Перезавантажені методи МОЖУТЬ міняти тип повернення.
- Перезавантажені методи МОЖУТЬ міняти модифікатор доступу.
- Перезавантажені методи МОЖУТЬ оголошувати нові або ширші відмічені виключення.
- Метод може бути перезавантажений в тому ж класі або в підкласі. Іншими словами, якщо клас А визначає метод doStuff(int i), підклас В

може визначити метод doStuff(String s) без перевизначення версії надкласу що використовує int. Тож 2 методи з одним імям але в різних класа можуть вже ж вважатись перезавантаженими, якщо підклас успадковує одну версію методу і потім оголошує іншу перезавантажену версію в визначенні свого класу.

Увага
Будьте обережні з відрізненням перезавантаженого методу від перевизначеного. Ви можете побачити метод що виглядає як викривлений приклад перевизначення, але це насправді буде легальне перезавантаження, як в наступному прикладі:

```
public class Foo {
  public void doStuff(int y, String s) { }
  public void moreThings(int x) { }
}
class Bar extends Foo {
  public void doStuff(int y, long s) throws IOException { }
}
```

Так і хочеться визначити IOException як проблему, оскільки перевизначений метод doStuff() не оголошує виключення, а IOException відзначене компілятором. Але метод doStuff() не перевизначений! Підклас Bar перезавантажує даний метод, змінюючи список аргументів, тож з виключенням все в порядку.
Легальні перезавантаження.
Подивимось на метод, який ми хочемо перезавантажити:

```
public void changeSize(int size, String name, float pattern) { }
```

Наступні методи є правильними перезавантаженнями методу changeSize():

```
public void changeSize(int size, String name) { }
public int changeSize(int size, float pattern) { }
public void changeSize(float pattern, String name)
                throws IOException { }
```

Виклик перезавантажених методів.
Зауважте, що набагато більше ми будемо обговорювати те, як компілятор знає який метод викликати, але решта буде розглянуто в розділі 3, де ми поглянемо на боксінг та var-arg – з якими часто стикаємось в перезавантаженні. (вам все ж треба приділити увагу цій частині).
Коли метод викликається, для одного типу об’єкту, на якому ви його викликаєте, може існувати більш ніж один метод з однаковим іменем. Для прикладу клас Horse може мати 3 методи з однаковим іменем, але різними списками аргументів, що означає, що метод перезавантажений.
Вибір серед методів що співпали базується на аргументах. Якщо ви викликаєте метод з рядковим аргументом, викликається перезавантажена версія, що вибирає змінну типу String. Якщо ви хочете викликати метод з

тим же імям, але використати аргумент типу float, буде запущена версія, яка вибирає float. Якщо ви викликаєте метод з тим же іменем але запускаєте його на об'єкті Foo, і немає перезавантаженої версії, що вибирає Foo, тоді компілятор буде жалітись нам на те, що він не може знайти співпадінь. Наступний код є прикладом виклику перезавантажених методів:

```
class Adder {
  public int addThem(int x, int y) {
    return x + y;
  }
  // Overload the addThem method to add doubles instead of ints
  public double addThem(double x, double y) {
    return x + y;
  }
}
// From another class, invoke the addThem() method
public class TestAdder {
  public static void main (String [] args) {
    Adder a = new Adder();
    int b = 27;
    int c = 3;
    int result = a.addThem(b,c); // Which addThem is invoked?
    double doubleResult = a.addThem(22.5,9.3); // Which addThem?
   }
}
```

В попередньому коді TestAdder, першим викликається a.addThem(b,c) і вибирає два цілі аргументи, тож викликається перша версія addThem() – перезавантажена версія, що вибирає два цілі аргументи. Наступний виклик йде до a.addThem(22.5, 9.3), що обробляє два аргументи типу double, тож викликається друга версія addThem() – перезавантажена версія, що працює з двома аргументами типу double.

Виклики перезавантажених методів, що використовують посилання на об'єкт, а не примітиви, є трошки цікавішими. Скажімо ви маєте перезавантажений метод, в якому одна версія використовує Animal, а інша Horse (підклас Animal). Якщо ви використовуєте об'єкт Horse в виклику методу, ви будете викликати перезавантажену версію, що використовує Horse. Подивимось на перший приклад:

```
class Animal { }
class Horse extends Animal { }
class UseAnimals {
  public void doStuff(Animal a) {
    System.out.println("In the Animal version");
  }
  public void doStuff(Horse h) {
    System.out.println("In the Horse version");
```

```
  }
  public static void main (String [] args) {
    UseAnimals ua = new UseAnimals();
    Animal animalObj = new Animal();
    Horse horseObj = new Horse();
    ua.doStuff(animalObj);
    ua.doStuff(horseObj);
  }
}
```

Вихід очікуваний:
in the Animal version
in the Horse version

А що якщо ви захочете використати посилання Animal на об'єкт Horse?

```
Animal animalRefToHorse = new Horse();
 ua.doStuff(animalRefToHorse);
```

Яка з перезавантажених версій буде виклиатись? Ви можете захотіти сказати «Та, яка використовує horse, оскільки зараз для методу використовується об'єкт Horse». Але працює вно не так. Попередній код при виконанні видасть:
in the Animal version
Навіть якщо під час запуску актуальним об'єктом є Horse а не Animal вибір перезавантаженго методу для виклику (іншими словами, підпису методу) не вирішується динамічно під час виконання. Просто запам'ятайте, що те, який метод викликається залежить від типу посилання (а не типу об'єкта)! Для підсумків, яку перевизначену версію методу викликати (іншими словами, з якого класу в дереві успадкування) вирішується в процесі виконання і базоване на типі об'єкта, а для перезавантаженої версії методу виклик базується на типі посилання аргумента, що використовувався на час компіляції. Якщо ви викликаєте метод, що використовує посилання Animal на об'єкт Horse, компілятор буде знати тільки про Animal, тож він вибере перезавантажену версію метода, що вибирає Animal. Немає різниці, що на час виконання використовується Horse.

Поліморфізм в перезавантажених та перевизначених методах
Як поліморфізм працює для перезавантажених методів? З того, що ми побачили щойно, не видно, щоб поліморфізм мав якесь значення при перезавантаженні метода. Якщо ви використовуєте посилання Animal, буде викликатись перезавантажений метод, що використовує Animal, навіть якщо на даний момент використовується Horse. В метод потрапляє об'єкт Horse, замаскований під Animal, хоча даний об'єкт і надалі залишається horse всупереч тому, що він використовується в методі як Animal. Тож це правда, поліморфізм не визначає, яку перезавантажену версію виклиати, поліморфізм грає роль якщо йде мова про виклик

перевизначеного методу. Але деколи метод як перезавантажується, так і перевизначається. Уявіть, що класи Horse та Animal виглядають так:

```
public class Animal {
  public void eat() {
    System.out.println("Generic Animal Eating Generically");
  }
}
public class Horse extends Animal {
  public void eat() {
     System.out.println("Horse eating hay ");
  }
  public void eat(String s) {
    System.out.println("Horse eating " + s);
  }
}
```

Зауважте, що клас Horse має як перезавантажений, так і перевизначений метод eat(). Таблиця 2-2 показує яка з версій трьох методів eat() буде запускатись в залежності від того, як вони викликаються.

Код виклику методу	Результат
Animal a = new Animal(); a.eat();	Generic Animal Eating Generically
Horse h = new Horse(); h.eat();	Horse eating hay
Animal ah = new Horse(); ah.eat(	Horse eating hay Поліморфізм працює – актуальний тип об'єкту (Horse), а не тип змінної (Animal) використовується для визначення, який з eat() буде викликатись
Horse he = new Horse(); he.eat("Apples");	Horse eating Apples Викликається перезавантажений метод eat(String s)
Animal a2 = new Animal(); a2.eat("treats");	Помилка компілятора! Компілятор бачить, що клас Animal не має методу eat(), що використовує аргумент типу String
Animal ah2 = new Horse(); ah2.eat("Carrots");	Помилка компілятора! Компілятор дивиться тільки на посилання, і бачить, що Animal не має методу eat(), що використовує string. Компілятора не хвилює те, що на даний момент актуальним об'єктом може бути Horse.

Увага

Не дайте обманути себе методами, що є перезавантаженими, але не перевизначеними підкласом. Наступне буде абсолютно правильним:

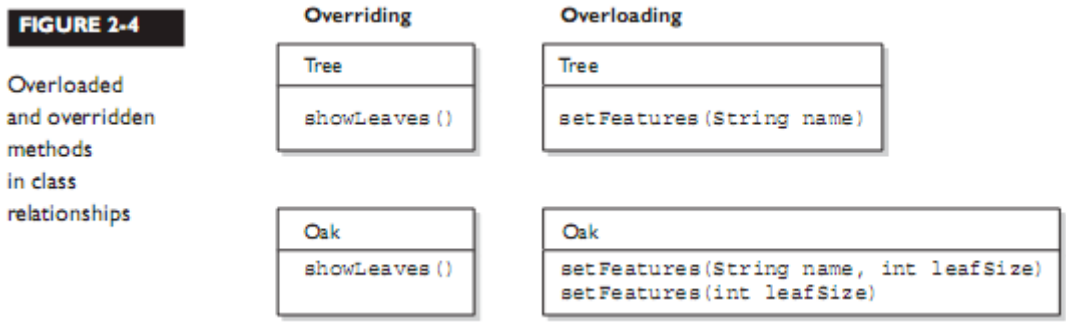
```
public class Foo {
  void doStuff() { }
}
class Bar extends Foo {
  void doStuff(String s) { }
}
```

Клас Bar має 2 методи doStuff() – безаргументна версія, яку він успадковує з Foo (і не пере визначає) та перезавантажена dostuff(String s), визначена в класі Bar. Код з посиланням на Foo може викликати тільки без аргументну версію, але код з посиланням на Bar може викликати обидві з перезавантажених версій.

Таблиця 2-3 підсумовує різницю між перезавантаженими та перевизначеними методами.

	Перезавантажені методи	Перевизначені методи
Аргумент(и)	Мусить змінюватись	Не має змінюванить
Тип повернення	Можуть змінюватись	Не можуть змінюватись, за виключенням коваріантних повернень
Виключення	Можуть змінюватися	Можуть видалятись чи усуватись. Не можна додавати нові чи ширше відмічені виключення
Доступ	Може мінятись	Не має ставати більш строгим (може бути менш строгим)
Виклик	Тип посилання визначає, котра з перевизначени версій (базована на оголошених типах аргументів) вибрана. Визначається під час компіляції. Актуальний метод, що викликається є все ж віртуальним викликом методу, що використовується на даний момент, але компілятор вже буде знати підпис метод для виклику. Тож на час запуску, спів падіння аргументів буде прив’язане, але не до класу, в якому живе метод.	Тип об’єкту (іншими словами, тип актуального екземпляру в памяті) визначає котрий з методів буде вибраний. Визначається під час виконання.

Поточна тема (5.4) покриває перезавантаження і методів і конструкторів, але ми розглянемо перезавантаження конструкторів трохи пізніше, де ми також покриємо інші теми, пов’язані з конструктором, що будуть на екзамені. Схема 2-4 демострує те, як перезавантажені та перевизначені методи використаються в зв’язках класів.



Вибір змінної посилання (тема 5.2)

За поданим сценарієм, розробіть од, що демонструє використання поліморфізму. Далі, визначте коли буде потрібен вибів і розпізнайте помилки компіляції та виконання, пов'язані з вибором посилання об'єкта.

Ми побачили, як можливо використовувати типи загальних змінних посилання для посилання на більш специфічні типи об'єктів. Це в серці поліморфізму. Для прикладу цей рядок коду був дотепер другою природою:

Animal animal = new Dog();

Але що станеться, коли ви хочете використати змінну посилання animal для виклику методу, який міститься тільки в класі Dog? Ви знаєте, що воно посилається на Dog, і ви хочете зробити річ, специфічну для Dog? В наступному коді, ми маємо масив Animal, в коли ми знайдемо в ньому Dog, ми захочемо виконати специфічну операцію. Давайте припустимо поки що, що з кодом все гаразд, крім того, що ми не впевнені в рядку, що викликає метод playDead.

```
class Animal {
  void makeNoise() {System.out.println("generic noise"); }
}
class Dog extends Animal {
  void makeNoise() {System.out.println("bark"); }
  void playDead() { System.out.println("roll over"); }
}
class CastTest2 {
  public static void main(String [] args) {
    Animal [] a = {new Animal(), new Dog(), new Animal() };
    for(Animal animal : a) {
      animal.makeNoise();
      if(animal instanceof Dog) {
        animal.playDead();       // try to do a Dog behavior ?
      }
    }
  }
}
```

Коли ми спробуємо скомпілювати даний код, компілятор скаже щось типу цього:

cannot find symbol
Компілятор каже: «Альо, клас Animal не має методу playDead()»
Давайте змінимо блок коду if:

```
if(animal instanceof Dog) {
  Dog d = (Dog) animal;      // casting the ref. var.
  d.playDead();
 }
```

Новий удосконалений блок коду містить вибір, який в цьому випадку деколи називають downcast, оскільки ми робимо вибір вниз по дереву успадкування до більш специфічних класів. Тепер компілятор щасливий. Перед тим, як ми спробуємо викликати playDead, ми вибрали для змінної animal тип Dog. Цим ми кажемо компілятору «Ми знаємо, що воно дійсно посилається на об'єкт Dog, тож це нормально – зробити нову змінну посилання Dog для посилання на цей об'єкт.» В цьому випадку ми в безпеці, оскільки будь-коли коли ми спробуємо зробити вибір, ми робимо тест instanceof для впевненості.
Важливо знати, що компілятор примушений довіряти нам, коли ми робимо downcast, навіть якщо ми лажаємо:

```
class Animal { }
class Dog extends Animal { }
class DogTest {
  public static void main(String [] args) {
    Animal animal = new Animal();
    Dog d = (Dog) animal;          // compiles but fails later
  }
}
```

Це капець! Цей код компілюється! Коли ми спробуємо запустити його на виконання, отримаємо наступне виключення:
конструювання програмного забезпечення.lang.ClassCastException

Чому ми не можемо довірити компілятору допомогу нам в цьому випадку? Чи він не бачить, що animal має тип Animal& Все що може зробити компілятор – це впевнитись, що два типи знаходяться в одному дереві успадкування, тож все залежить від коду перед вибором, можливо, що animal має тип Dog. Компілятор має дозволити речі, які можливо будуть працювати під час виконання. Хоча якщо компілятор точно знає, коли вибір не буде працювати, то компіляція не вдасться. Наступні зміни в блоці коду if не скомпілюються:

```
Animal animal = new Animal();
Dog d = (Dog) animal;
String s = (String) animal;  // animal can't EVER be a String
```

В цьому випадку, ви отримуєте помилку типу цієї:
inconvertible types
На відміну від downcasting, upcasting (вибір вгору по дереву успадкування до більш загальни типів) працює беззаперечно (так як вам не треба

вводити тип в виборі) Оскільки коли ви виконуєте upcasting, ви строго обмежуєте число методів, яке ви можете викликати, на противагу downcasting, який значить, що ви будете викликати більш специфічні методи. Для прикладу:

```
class Animal { }
class Dog extends Animal { }
class DogTest {
  public static void main(String [] args) {
    Dog d = new Dog();
    Animal a1 = d;           // upcast ok with no explicit cast
    Animal a2 = (Animal) d;  // upcast ok with an explicit cast
  }
}
```

Обидва з upcasting, подані вище, будуть компілюватись і запускатись без виключень, оскільки Dog IS-A Animal, що означає, що все що може робити Animal, може робити і Dog. Звичайно, Dog може робити більше, але суть в тому, що кожен з посиланням Animal може безпечно викликати методи Animal на екземплярі Dog. Методи Animal можуть перевизначатись в класі Dog, але ми зараз хвилюємось за те, щоб Dog міг робити як мінімум все, що може зробити Animal. Компілятор і JVM знають це теж, тож безперечний upcast завжди легальний для призначення об'єкту підтипу для посилання на один з класів (або інтерфейсів) над типу. Якщо Dog виконує Pet, і Pet визначає beFriendly(), тоді Dog може вибирати Pet, але єдиний метод Dog, який ви можете викликати, буде beFriendly(), який Dog буде змушений виконувати, оскільки Dog виконує інтерфейс Pet.

Ще одне… якщо Dog виконує Pet, тоді якщо Beagle розширює Dog, але не оголошує, що виконує Pet, Beagle все ж буде Pet! Beagle буде Pet тільки тому, що він розширює Dog, а на рахунок Dog і його дочірніх класів вже попіклувались щодо виконання Pet. Клас Beagle може завжди перевизначати будь-які методи, успадковані від Dog, включаючи ті, які Dog виконує для дотримання контракту інтерфейсу.

І ще одне… Якщо Beagle оголошує, що він виконує Pet, так щоб всі, поглянувши на API класу Beagle могли легко побачити, що Beagle IS-A Pet, без перегляду надкласів Beagle, він все одно не має виконувати метод beFriendly(), якщо клас Dog (надклас класу Beagle) попіклувався про це. Іншими словами, якщо Beagle IS-A Dog, і Dog IS-A Pet, тоді Beagle IS-A Pet, і все отримав зобов'язання Pet для виконання beFriendly(), оскільки він успадкував метод beFriendly(). Компіляторові стане розуму сказати «Я знаю, що Beagle вже IS a Dog, але ви можете зробити це більш очевидним.»

Тож не дайте себе обдурити кодом, що показує дійсний клас, що оголошує, що він виконує інтерфейс, але не виконує методи інтерфейсу. Перед тим, як ви скажете, чи код є правильним, ви маєте знати що оголосили надкласи виконуючого класу. Якщо якийсь надклас в його

дереві успадкування вже забезпечив дійсне (не абстрактне) виконання методу, тоді незалежно від того, чи надклас оголошує, що він виконує інтерфейс, підклас не має зобов'язання перевиконувати (перевизначати) ці метод.

Увага!
Творці екзамену скажуть що вони змушені ліпити тонни коду в малий простір «через движок екзамену». Це частково правда, але вони також хочуть і запутати. Наступний код:
Animal a = new Dog();
Dog d = (Dog) a;
d.doDogStuff();
Може бути замінений таким легко читабельним приколом:
Animal a = new Dog();
((Dog)a).doDogStuff();
В цьому випадку компілятору потрібні ці всі дужки, інакше він побачить в ньому незакінчений вираз.

Виконання інтерфейсу (тема 1.2)
1.2 розробіть код що оголошує інтерфейс…
Коли ви виконуєте інтерфейс, ви погоджуєтесь виконувати контракт, визначений в інтерфейсі. Це означає, що ви погоджуєтесь забезпечувати правильне виконання для кожного методу, визначеного в інтерфейс, і той, хто знає як виглядають методи інтерфейсів (не як вони виконуються, а як вони можуть називатись і що вони повертають) може полегшено зітхнути, і бути впевненим, що вони можуть викликати ці методи на екземплярі вашого класу, що виконується.
Для прикладу, якщо ви створите клас, що виконує інтерфейс Runnable (ваш код може виконуватись специфічним потоком), ви мусите забезпечити метод public void run(). В іншому випадку, бідному потоку може бути наказано виконувати код об'єкту Runnable і – сюрпрайз!!! – потік бачить, що об'єкт не має методу Run()! (І тут потік виносить і JVM розривається жахливим вибухом). Дякувати Богу, Конструювання програмного забезпечення не допускає виникнення такого колапсу, використовуючи запуск перевірки компілятором кожного класу, який хоче виконати інтерфейс. Якщо клас каже, що він виконує інтерфейс, він має мати виконання кожного методу інтерфейсу (є кілька винятків, на них глянемо за хвилину).
Маємо інтерфейс Bounceable з двома методами: bounce() та setBounceFactor(), наступний клас буде компілюватись:

```
public class Ball implements Bounceable {  // Keyword
                                 // 'implements'
  public void bounce() { }
  public void setBounceFactor(int bf) { }
}
```

ОК, ми знаємо що ви зараз думаєте: «це найгірший клас виконання в історії класів виконання». Але він компілюється. І запускається. Контракт інтерфейсу гарантує що клас буде мати метод)іншими словами, інші можуть викликати суб'єкт методу до контролю доступу), але він не гарантує правильне виконання – чи навіть жодного коду виконання в тілі методу. Компілятор ніколи не скаже вам «Ем, вибачте, але ви дійсно нічого не хотіли ставити між фігурними дужками? Ульо!!! Це ж метод все-таки, хіба він нічого не має робити?»
Класи виконання мусять слідувати виконанню метод як клас, що розширює абстрактний клас. Для того, щоб бути легальним класом виконання, неабстрактний клас виконання має робити наступне:
Забезпечувати дійсні (неабстрактні) виконання для всіх методів з оголошеного інтерфесу.
Слідувати всім правилам легальних перевизначень.
Не оголошувати відмічених виключень на методи виконання, крім тих, що були оголошені методом інтерфейсу, чи підкласів тих, що були оголошені методом інтерфейсу.
Підтримувати підпис методу інтерфейсу і підтримувати той же тип (або підтип) повернення). (але може не оголошувати виключення, оголошені в оголошенні методу інтерфейсу.)

Але почекайте, ще одне! Клас виконання сам по собі може бути абстрактним! Для прикладу наступне буде правильним для класу Ball, що виконує Bounceable:
abstract class Ball implements Bounceable { }
Нічого не пропустили? Ми ніде не забезпечували методи виконання. І це нічого. Якщо клас виконання є абстрактним, він може передати виконання першому ж дійсному підкласу. Для прикладу, якщо клас BeachBall розширює Ball, і BeachBall не є абстрактним, то BeachBall буде змушений забезпечувати всі методи з Bounceable:

```
class BeachBall extends Ball {
  // Even though we don't say it in the class declaration above,
  // BeachBall implements Bounceable, since BeachBall's abstract
  // superclass (Ball) implements Bounceable
   public void bounce() {
      // interesting BeachBall-specific bounce code
   }
   public void setBounceFactor(int bf) {
      // clever BeachBall-specific code for setting
      // a bounce factor
   }
   // if class Ball defined any abstract methods,
   // they'll have to be
   // implemented here as well.
}
```

Пошукайте класи, що вимагають виконання інтерфейсу але не забезпечують коректне виконання методу. Хоча виконуваний клас є абстрактним, все ж виконуваний клас має забезпечувати виконання всіх методів, визначених в інтерфейсі.

Ще два правила, які вам потрібно знати перед тим як відправити цю тему відпочивати (чи відправити вас відпочивати, яка різниця):

1. Клас може виконувати більше одного інтерфейсу. Правильно буде сказати, для прикладу, наступне:

```
public class Ball implements Bounceable, Serializable, Runnable
{ ... }
```

Ви можете розширювати тільки один клас, але виконувати багато інтерфейсів. Але запам'ятайте, що підкасування визначає хто ви і що ви, тоді як виконання визначає, яку роль ви можете грати чи яку маску одягнути, показуючи наскільки різним може бути виконання іншим класом того ж інтерфейсу (але з іншого дерева успадкування). Для прикладу, Person розширює HumanBeing (хоча для когось це спірне питання). Але Person може також виконувати Programmer, Snowboarder, Employee, Parent чи PersonCrazyEnoughToTakeThisExam.

2. Інтерфейс може собою розширювати інший інтерфейс, але ніколи – виконувати щось. Наступний код є абсолютно правильним:

```
public interface Bounceable extends Moveable { }   // ok!
```

Що це означає? Перший дійсний(неабстрактний) клас виконання з Bounceable має виконувати всі методи Bounceable, плюс всі методи з Moveable! Підінтерфейс, як ми його назвемо, просто додає більше вимог до контракту надінтерфейсу. Ви побачите, як ця концепція використовується в багатьох областях Конструювання програмного забезпечення, особливо J2EE, де ви часто будете змушені будувати власний інтерфейс, що розширює один з інтерфейсів J2EE.

Залишайтесь з нами, далі – ще дивніше. Інтерфейс може розширювати більше ніж один інтерфейс! Задумайтесь над цим. Ви знаєте, що якщо ми говоримо про класи, наступне є нелегальним:

```
public class Programmer extends Employee, Geek { } // Illegal!
```

Як ми згадували раніше, класу не дозволяється розширювати декілька класів Конструювання програмного забезпечення. А інтерфейсу дозволено розширювати декілька інтерфейсів.

```
interface Bounceable extends Moveable, Spherical {   // ok!
  void bounce();
  void setBounceFactor(int bf);
}
interface Moveable {
  void moveIt();
}
interface Spherical {
  void doSphericalThing();
}
```

В наступному прикладі, від Ball вимагається виконувати Bounceable, плюс всі методи з інтерфейсів, які розширює Bounceable (включаючи всі інтерфейси, які розширюють дані інтерфейси, і так далі доки ви не досягнете вершини стеку – чи може це дно стеку?). Тож ball мусить виглядати на зразок наступного:

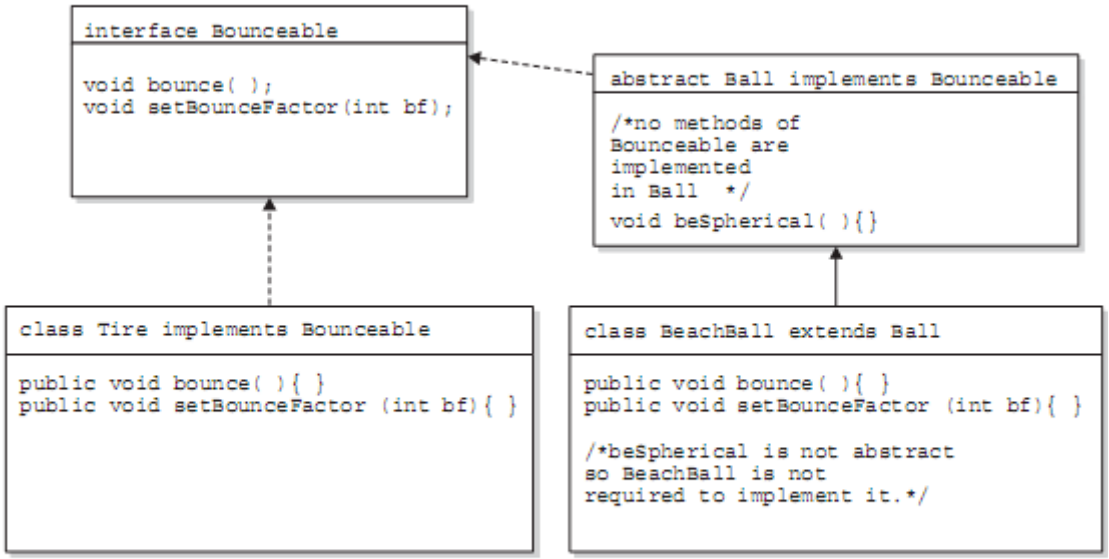
```
class Ball implements Bounceable {

  public void bounce() { }   // Implement Bounceable's methods
  public void setBounceFactor(int bf) { }
  public void moveIt() { }    // Implement Moveable's method

  public void doSphericalThing() { }   // Implement Spherical
}
```

Якщо клас Ball не може виконати хоч один з методів з Bounceable, Moveable чи Spherical, компілятор буде скакати в безумстві, доки Ball не виконає цей метод. Якщо звичайно, клас ball не позначений як sbstract. В цьому випадку, Ball може вибрати для виконання будь-якого, усіх, чи жодного методу з будь-якого інтерфейсу, залишаючи решту дійсному підкласу Ball, як показано нижче:

```
abstract class Ball implements Bounceable {
  public void bounce() { ... }  // Define bounce behavior
  public void setBounceFactor(int bf) { ... }
  // Don't implement the rest; leave it for a subclass
}
class SoccerBall extends Ball {  // class SoccerBall must
          // implement the interface methods that Ball didn't
  public void moveIt() { ... }
  public void doSphericalThing() { ... }
  // SoccerBall can choose to override the Bounceable methods
  // implemented by Ball
  public void bounce() { ... }
}
```

Схема 2-5 порівнює дійсні та абстрактні приклади розширення та виконання для класів та інтерфейсів.

FIGURE 2-5 Comparing concrete and abstract examples of extends and implements



Because BeachBall is the first concrete class to implement Bounceable, it must provide implementations for all methods of Bounceable ,except those defined in the abstract class Ball. Because Ball did not provide implementations of Bounceable methods, BeachBall was required to implement all of them.

Оскільки Beachball є першим дійсним класом для Виконання Bounceable, він має забезпечувати вионання всіх Методів з Bounceable, крім тих, які визначені в абстрактному Класі Ball. Оскільки Ball не забезпечує виконання методів Bounceable, BeachBall мусить виконувати їх всіх.

Увага

Подивіться на неправильні використання розширень і виконань. Наступні приклади демонструють правильні та неправильні оголошення класів та інтерфейсів.

```
class Foo { }                   // OK
class Bar implements Foo  { }      // No! Can't implement a class
interface Baz { }                 // OK
interface Fi { }                  // OK
interface Fee implements Baz { }   // No! Interface can't
                                   // implement an interface
interface Zee implements Foo { }   // No! Interface can't
                                   // implement a class
interface Zoo extends Foo { }      // No! Interface can't
                                   // extend a class
interface Boo extends Fi { }       // OK. Interface can extend
                                   // an interface
class Toon extends Foo, Button { } // No! Class can't extend
                                   // multiple classes
class Zoom implements Fi, Baz { }  // OK. class can implement
                                   // multiple interfaces
interface Vroom extends Fi, Baz { }  // OK. interface can extend
```

```
                                    // multiple interfaces
class Yow extends Foo implements Fi { } // OK. Class can do both
                                    // (extends must be 1st)
```

Зарубайте собі це на носі і зважайте на це в питаннях, що попадуться на екзамені. Незалежно від того, яке питання задається на тестуванні, проблема може бути в оголошенні інтерфейсу чи класу. Перед тим як приступити, скажімо до трасування поточного виконання, перевірте хоча б чи цей код взагалі компілюється. (Ця маленька річ, яку ми вам сказали, може бути вартою уваги!) (Ви будете здивовані спробами розробників екзамену відвести вас від реальної проблеми.) (І як тільки люди могли писати перед тим, як винайшли дужки?)

Легальні типи повернення (Тема 1.5)

1.5. За поданим прикладом коду визначте чи метод правильно перезавантажує та пере визначає інший метод, і визначте легальні значення повернення (включаючи коваріантні повернення), для методу.

Ця тема покриває два аспекти типів повернення: що ви можете оголосити як тип повернення і що ви можете повертати як значення. Що ви можете оголошувати а що ні, досить прямолінійно, але все залежить від того чи ви пере визначаєте успадкований метод чи просто оголошуєте новий метод (що включає перезавантажені методи). Ви тільки подивимось на різницею в правилах типів повернення для перезавантажених та перевизначених методів, оскільки ми вже розглядали їх в цій частині. Ми покриємо малу частку нового, коли ми подивимось на поліморфічні типи повернення і правила що правильно а що ні для повернення.

Оголошення типів повернення

Ця підтема розглядає, що вам дозволено оголошувати типом повернення, що в першу чергу залежить від того, чи ви перезавантажуєте, перевизначаєте чи оголошуєте новий метод.

Типи повернення для перезавантажених методів.

Пам’ятайте, перезавантаження методів це не більше ніж пере використання імені. Перезавантажений метод це повністю інший метод з тим же іменем. Тож якщо ви успадковуєте метод, але перезавантажуєте його в підкласі, вас не будуть стосуватись обмеження перевизначення, що означає, що ви можете задавати будь-який тип повернення. Що ви не можете зробити – так це змінити ТІЛЬКИ тип повернення. Згадайте, для перезавантаження методів вам слід змінити список аргументів. Наступний код показує перезаватажений метод:

```
public class Foo{
  void go() { }
}
public class Bar extends Foo {
  String go(int x) {
```

```
    return null;
  }
}
```
Зауважте, що версія методу bar, використовує інший тип повернення. Це класно. Оскільки ви змінили список аргументів, ви перезавантажили метод, тож тип повернення не має відповідати типу повернення версії надкласу. Але вам НЕ дозволено робити наступне:
```
public class Foo{
  void go() { }
}
public class Bar extends Foo {
  String go() { // Not legal! Can't change only the return type
    return null;
  }
}
```
Перевизначення, типи повернення та коваріантні повернення

Коли підклас хоче змінити виконання успадкованого методу (і перевизначити), підклас мусить визначити метод, що відповідає успадкованій версії. Або, як в конструювання програмного забезпечення 5, вам дозволено міняти тип повернення в методі, що перевизначається тоді, коли новий тип повернення є підтипом оголошеного типу повернення перевизначеного (суперкласового) методу.

Подивимось на коваріантне повернення в дії:
```
class Alpha {
  Alpha doStuff(char c) {
    return new Alpha();
  }
}
class Beta extends Alpha {
   Beta doStuff(char c) {       // legal override in Конструювання програмного забезпечення 1.5
    return new Beta();
  }
}
```
Згідно з конструювання програмного забезпечення5, код буде компілюватись. Якщо ви спробуєте скомпілювати цей код компілятором 1.4 або прапорцем джерела як нижче:

конструювання програмного забезпеченняс -source 1.4 Beta.конструювання програмного забезпечення

Ви отримаєте помилку компілятора типу цієї:

attempting to use incompatible return type

(Ми поговоримо про прапорці компіляторів детальніше в десятому розділі).

Інші правила застосування перевизначення включають правила для модифікаторів доступу та оголошених виключень, але вони не відповідають темі обговорення типів повернення.

Для екзамену будьте впевнені, що ви знаєте, що перезавантажені методи можуть міняти тип повернення, але перевизначення методів може робити це теж, в межах оварінтних повернень. Навіть це єдине знання допоможе в широкому спектрі питань на екзамені.

Повернення значення

Вам потрібно запам'ятати тільки 6 правил для повернення значень

1. Ви можете повертати null в методі з типом повернення посилання об'єкта.

```
public Button doStuff() {
   return null;
 }
```

2. Масив може бути типом повернення.

```
public String[] go() {
  return new String[] {"Fred", "Barney", "Wilma"};
}
```

3. В методі з типом повернення у вигляді примітиву, ви можете повертати будь-яке значення чи змінну, що може бути конвертованою в оголошений тип повернення.

```
public int foo() {
  char c = 'c';
  return c;  // char is compatible with int
}
```

4. в методі з типом повернення у вигляду примітиву, ви можете повертати будь-яке значення чи змінну, що може вибиратись згідно оголошеного типу повернення.

```
public int foo () {
  float f = 32.5f;
  return (int) f;
}
```

5. Ви не маєте нічого повертати з методу з типом повернення void.

```
public void bar() {
  return "this is it";  // Not legal!!
}
```

6. В методі з типом повернення посилання на об'єкт, ви можете повернути будь-який тип об'єкту що може бути вибраний згідно оголошеного типу повернення.

```
public Animal getAnimal() {
  return new Horse();  // Assume Horse extends Animal
}
public Object getObject() {
  int[] nums = {1,2,3};
  return nums;  // Return an int array,
          // which is still an object
}
```

```
public interface Chewable { }
public class Gum implements Chewable { }
public class TestChewable {
   // Method with an interface return type
   public Chewable getChewable() {
     return new Gum();  // Return interface implementer
   }
}
```

Увага
Зважайте на методи, що оголошують тип повернення абстрактного класу чи інтерфейсу, і знайте, що кожен об'єкт, що проходить тест IS-A (іншими словами, перевіряє правильність твердження, використовуючи оператор instanceof) може повертатись з цього методу. Для прикладу:

```
public abstract class Animal { }
public class Bear extends Animal { }
public class Test {
  public Animal go() {
    return new Bear();  // OK, Bear "is-a" Animal
  }
}
```

Цей код буде компілюватись, значенням повернення є підтип.

Конструктори та екземпляризація (Теми 1.6, 5.3, 5.4)

1.6. За поданим набором класів та накласів, розробіть конструктори для одного чи більше класів. За поданим оголошенням класу, визначте чи буде створено конструктор по замовчуванню, і якщо так, визначте поведінку конструктора. За поданим списком класів, напишіть код для екземпляризації класу.
5.3. Поясніть дію модифікаторів на успадкування, зважаючи на конструктори, екземпляри чи статичні змінні, та екземпляри чи статичні методи.
5.4. За поданим сценарієм, розробіть код що оголошує і/чи викликає перевизначений чи перезавантажений метод, та код що що оголошує і/чи викликає надклас, перевизначені чи перезавантажені конструктори.

Об'єкти є сконструйованими. Ви не можете створити новий об'єкт без виклику конструктора. Фактично, ви не можете зробити новий об'єкт без виклику не тільки конструктора типу актуального об'єкта, а й конструктора кожного з його надкласів! Конструктори – це код, який запускається будь-коли, коли ви використовуєте слово new. Гаразд, щоб бути більш точними, ще мають бути блоки ініціалізації, о запускаються, коли ви кажете new, але ми розглянемо їх та тему статичної ініціалізації в наступному розділі. Ми й так маємо про що тут поговорити – ми подивимось, як кодуються конструктори, хто її кодує і як вони працюють під час виконання. Тож беріть каску і молот, позаймаємось будівництвом об'єктів.

Основи конструкторів. Кожен клас, включаючи абстрактні, мусить мати конструктор. Запам'ятайте це. Але якщо клас його має мати, то це ще не

означає, що програміст має друкувати його. Конструктор виглядає наступним чином:

```
class Foo {
  Foo() { } // The constructor for the Foo class
}
```

Помітили чого не вистачає? Немає типу повернення! Два ключові моменти для запам'ятовування щодо конструкторів це те, що вони не мають типу повернення і їхні імена мають точно відповідати іменам відповідних класів. Зазвичай, конструктори використовуються для ініціалізації стану змінної екземпляру, як показано нижче:

```
 class Foo {
   int size;
   String name;
   Foo(String name, int size) {
     this.name = name;
     this.size = size;
   }
}
```

В попередньому прикладі коду, клас Foo не має безаргументного конструктора. Це означає, що компіляція наступного не відбудеться:

```
Foo f = new Foo();  // Won't compile, no matching constructor
```

Але наступне скомпілюється:

```
Foo f = new Foo("Fred", 43);  // No problem. Arguments match
                              // the Foo constructor.
```

Тож це звично (і бажано) для класу мати безаргументний конструктор, незважаючи на те, скільки інших перезавантажених конструкторів є в класі (так, конструктори можуть перезавантажуватись). Ви не завжди можете робити це класів, може трапитись ситуація, коли немає сенсу створювати екземпляр без забезпечення конструктора інформацією. Об'єкт конструювання програмного забезпечення.awt.Color, для прикладу, не можу створюватись викликом безаргументного конструктора, оскільки це буде все одно що сказати до JVM «Зробіть новий об'єкт Color, і мені по барабану що це буде за колір… на ваш смак». Ви дійсно хочете, щоб JVM сама робила рішення в вашому стилі?

Прив'язка конструктора

Ми знаємо, що конструктори викликаються під час виконання, коли ви кажете new перед певним типом класу, як показано нижче:

```
Horse h = new Horse();
```

Але що стається насправді, коли ви кажете new Horse()?

(Припустимо, що horse розширює Animal, а Animal розширює Object).

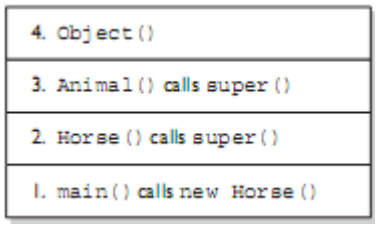
1. Викликається конструктор Horse. Кожен конструктор викликає конструктор свого надкласу з допомогою безумовного виклику super(),

якщо тільки конструктор не викликає перезавантажений конструктор такого ж класу (більше про це за хвилину).
2. Викликається конструктор Animal (Animal є надкласом Horse).
3. Викликається кнструктор Objaect (це загальний надклас усіх класів, тож Animal розширює Object навіть якщо ви очевидно не введете “extends Object” в оголошенні класу Animal. Це безумовно.) Тепер ми на вершині стеку.
4. Змінні екземпляру об’єкта отримують задані значення. Під заданими ми розуміємо значення, які були визначені на час оголошення змінних, наприклад "int x = 27", де 27 буде заданим значенням (на противагу значенню по замовчуванню) змінної екземпляру.
5. конструктор Object завершує роботу.
6. змінні екземпляру Animal отримують свої задані значення (якщо такі є).
7. конструктор Animal завершує роботу.
8. Змінні екземпляру Horse отримують свої значення (якщо такі є).
9. конструктор Horse завершує роботу.

Схема 2-6 показує як працює конструктор на стеку виклику.

FIGURE 2-6

Constructors on the call stack



Правила для конструкторів
Наступний список підсумовує правила які вам слід знати для екзамену (і щоб зрозуміти решту цієї підтеми). Ви МУСИТЕ запам’ятати це, тож вивчіть і декілька разів повторіть це.

- Конструктори можуть використовувати будь-які модифікатори доступу, включаючи private. (Приватний конструктор означає, що тільки код всередині класу може екземпляризувати об’єкт даного типу, тож якщо приватний клас конструктора хоче дозволити використовувати екземпляр класу, лас має забезпечити статичний метод чи змінну що дозволяє доступ до екземпляру, створеного з класу.)
- Імя конструктора має відповідати імені класу.
- Конструктори не мають типу повернення.
- Це легально (але по-дурному) мати метод з тим же іменем, що клас, але це не робть даний метод конструктором. Якщо ви бачите тип повернення, це метод а не конструктр. Фактично, ви можете мати і метод, і конструктор з однаковим іменем – іменем класу – в тому ж класі, для Конструювання програмного забезпечення це не проблема.

Будьте уважні у різниці між методом і конструктором – дивіться на тип повернення.

- Якщо ви не написали конструктор в вашому коді класу, то конструктор по замовчуванню буде згенерований компілятором.
- Конструктор по замовчуванню завжди є безаргументним.
- Якщо вам потрібен безаргументний конструктор і ви ввели якісь інші конструктори в вашому коді, компілятор не забезпечить безаргументний конструктор (чи будь-який інший конструктор) для вас. Іншими словами, якщо ви ввели конструктор з аргументами, ви не будете мати безаргументного конструктора, поки самі його не зробите!
- Кожен конструктор містить в першому виразі виклик перезавантаженого конструктора (this()) або виклик конструктора накласу (super()), хоча пам'ятайте, що цей виклик може вставлятись компілятором.
- Якщо ви ввели код конструктора (замість того, щоб покластися на створений компілятором по замовчуванню) і не ввели виклик super() чи виклик this(), компілятор вставить без аргументний виклик super() замість вас, поставивши його першим виразом в конструкторі.
- Виклик super() може бути без аргументним чи містити аргументи, які застосовує над конструктор.
- Без аргументний конструктор не обов'язково є по замовчування, хоча конструктор по замовчуванню є завжди без аргументним. Його може забезпечити компілятор, але ви й самі можете вставити свій без аргументний конструктор.
- Ви не можете зробити виклик до екземпляру методу, чи отримати доступ до змінної екземпляру, якщо не запустився над конструктор.
- Тільки статичні змінні та методи можуть бути доступні як частина виклику super() чи this(). (Приклад: super(Animal.NAME) є правильним, оскільки NAME оголошена як статична змінна.
- Абстрактні класи мають конструктори, і ці конструктори викликаються завжди, коли екземпляризується дійсний клас.
- Інтерфейси не мають конструкторів. Інтерфейси не є частиною дерева успадкування об'єкта.
- Єдиний спосіб виклику конструктора – з допомогою іншого конструктора. Іншими словами, ви не можете написати код, що явно викликає конструктор, як нижче:

```
class Horse {
 Horse() { } // constructor
 void doStuff() {
   Horse();  // calling the constructor - illegal!
  }
}
```

Визначення, коли буде створений конструктор по замовчуванню.
Наступний приклад показує клас Horse з двома конструкторами:

```
class Horse {
  Horse() { }
  Horse(String name) { }
 }
```

Чи буде компілятор вставляти конструктор по замовчуванню в клас вище? Ні! А як на рахунок наступної комбінації:

```
class Horse {
  Horse(String name) { }
}
```

А тепер, буде компілятор вставляти конструктор по замовчуванню чи ні? Ні!

А як на рахунок цього класу:

```
class Horse { }
```

А тепер поговоримо. Компілятор згенерує конструктор по замовчуванню для попереднього класу, оскільки він не має жодного визначеного конструктора. Добре, а як на рахунок цього класу?

```
class Horse {
  void Horse() { }
}
```

Виглядає так, ніби компілятор не буде створювати нічого, оскільки вже є конструктор в класі Horse. Але чи є? Подивимось ще раз на попередній клас.

Що неправильного в конструкторі Horse()? Це взагалі не конструктор! Це просто метод який має те ж ім'я, що й клас. Пам'ятайте, тип повернення – вірний знак того, що ми бачимо перед собою метод, а не конструктор.

Як ми можемо бути впевнені в тому, що буде створений конструктор по замовчуванню?

Якщо ми не написали жодних конструкторів для класу.

Як знати, що перед тобою конструктор по замовчуванню?

- конструктор по замовчуванню має той же модифікатор доступу, що і клас.
- Конструктор по замовчуванню не має аргументів.
- Конструктор по замовчуванню включає без аргументний виклик надконструктора

(super()).

Таблиця 2-4 показує що компілятор згенерує (або не згенерує) для вашого класу.

Що стається, коли надконструктор має аргументи.

Конструктори можуть мати аргументи, як і методи, і якщо ви захочете викликати метод, що використовує int, але ви нічого не надасте методу, компілятор буде жалітись, як в наступному прикладі:

```
class Bar {
   void takeInt(int x) { }
}
class UseBar {
  public static void main (String [] args) {
    Bar b = new Bar();
    b.takeInt();  // Try to invoke a no-arg takeInt() method
  }
}
```

Компілятор пожаліється на те, що ви не можете викликати takeInt() без наявності int. Звичайно, компілятор обожнює випадкові загадки, тож повідомлення, яке він видасть в деяких версіях JVM буде більш ніж очевидним:

```
UseBar.конструювання програмного забезпечення:7: takeInt(int) in Bar cannot be applied to ()
    b.takeInt();
     ^
```

Але ви маєте ідею. Нижній рядок означає, що має бути відповідність методу. А під відповідністю ми маємо на увазі, що типи аргументи мусять мати можливість для прийому значень чи змінних, які ви використовуєте, в порядку, в якому ви їх використовуєте. Що повертає нас до конструкторів (зараз ви подумаєте, що ми ніколи туди не доберемось), які працюють в той же спосіб.

TABLE 2-4 Compiler-Generated Constructor Code

Class Code (WhatYouType)	Compiler Generated Constructor Code (in Bold)
`class Foo { }`	`class Foo {` `  **Foo() {**` `  **super();**` `  **}**` `}`
`class Foo {` `  Foo() { }` `}`	`class Foo {` `  Foo() {` `  **super();**` `  }` `}`
`public class Foo { }`	`public class Foo {` `  **public Foo() {**` `  **super();**` `  **}**` `}`
`class Foo {` `  Foo(String s) { }` `}`	`class Foo {` `  Foo(String s) {` `  **super();**` `  }` `}`
`class Foo {` `  Foo(String s) {` `  super();` `  }` `}`	*Nothing, compiler doesn't need to insert anything.*
`class Foo {` `  void Foo() { }` `}`	`class Foo {` `  void Foo() { }` `  **Foo() {**` `  **super();**` `  }` `}` *(void Foo() is a method, not a constructor.)*

Тож якщо ваш надконструктор (тобто конструктор вашого безпосереднього надкласу) має аргументи, ви маєте ввести виклик до super(), перед тим визначивши відповідні аргументи. Важливий момент: якщо ваш надклас не має безаргументного конструктора, ви мусите ввести конструктор у вашому класі (підкласі) оскільки вам потрібне місце для вставки виклику super з відповідними аргументами.

Наступний код є прикладом даної проблеми:

```
class Animal {
  Animal(String name) { }
}
class Horse extends Animal {
  Horse() {
    super();  // Problem!
  }
}
```

І знов компілятор називає нас негарними словами:

Horse.конструювання програмного забезпечення:7: cannot resolve symbol
symbol : constructor Animal ()

```
location: class Animal
    super();  // Problem!
    ^
```

Якщо ви щасливчик (і зараз повний місяць), ваш компілятор буде більш однозначним. Але знову ж, проблема в тому, що просто не має відповідності в тому, що ми збираємось викликати з допомогою super() – без аргументний конструктор Animal.

Інший спосіб вставити це – це якщо надклас не має без аргументного конструктора, тоді в вашому підкласі ви не зможете використати конструктор по замовчуванню, забезпечуваний компілятором. Це просто. Оскільки компілятор може тільки вставляти виклик без аргументного supre(), ви не зможете зробити щось подібне до наступного:

```
class Clothing {
  Clothing(String s) { }
}
class TShirt extends Clothing { }
```

Спроба скомпілювати цей код дає нам таку ж помилку, яку ми отримали коли вставили конструктор в підкласі з викликом без аргументної версії super():

```
Clothing.конструювання програмного забезпечення:4: cannot resolve symbol
symbol  : constructor Clothing  ()
location: class Clothing
class TShirt extends Clothing { }
^
```

В дійсності, попередній код Clothing та TShirt є такий же, як наступний код, де ми забезпечили конструктор для Tshirt, що є ідентичним до конструктора по замовчуванню, який нам забезпечує компілятор:

```
class Clothing {
  Clothing(String s) { }
}
class TShirt extends Clothing {
            // Constructor identical to compiler-supplied
            // default constructor
  TShirt() {
    super(); // Won't work!
  }     // Invokes a no-arg Clothing() constructor,
}       // but there isn't one!
```

Ще одне ключове питання щодо конструкторів по замовчуванню (яке здається досить очевидним, але ми маємо про нього згадати, а то нас совість загризе насмерть), конструктори ніколи не успадковуються. Це не методи. Вони не можуть перевизначатись (оскільки вони не методи, а тільки екземпляри методів можуть перевизначатись). Тож тип конструкторів, які має ваш надклас, не визначає тип конструктора, який ви отримуєте. Деякі

чуваки помилково вірять, що конструктор по замовчуванню якось співпадає з над конструктором і в аргументах, які має конструктор по замовчуванню (пам'ятаєте, він завжди без аргументний), чи в аргументах які використовуються в підтримуваному компілятором виклику super().
Тож, хоча конструктори не можуть бути перевизначеними, ви побачили, що вони можуть перезавантажуватись, і часто так і роблять.

Перезавантажені конструктори
Перезавантаження конструктора означає ввід декількох версій конструктора, кожна з яких має свій список аргументів, як у наступних прикладах:

```
class Foo {
  Foo() { }
  Foo(String s) { }
}
```

Попередній клас Foo має два перезавантажені конструктори, один використовує рядкову величину, а інший не має аргументів. Оскільки код в без аргументній версії відсутній, вона є ідентичною до конструктора по замовчуванню, спроектованого компілятором, але пам'ятайте, якщо в класі вже є конструктор (той, що використовує рядкову величину), компілятор не забезпечить конструктор по замовчуванню. Якщо вам потрібен без аргументний конструктор для перезавантаження аргументної версії, яка у вас вже є, ви мусите самі його ввести, як було показано в попередньому прикладі.
Перезавантаження конструктора зазвичай використовується для забезпечення клієнтів альтернативними шляхами екземпляризації об'єктів вашого класу. Для прикладу, якщо клієнт знає імя тварии, він може задіяти конструктор Animal, що використовує рядкову величину. Але якщо імя невідоме, клієнт може викликати без аргументний конструктор і цей конструктор надасть імя по замовчуванню. Ось так це виглядає:

```
1. public class Animal {
2.   String name;
3.   Animal(String name) {
4.     this.name = name;
5.   }
6.
7.   Animal() {
8.     this(makeRandomName());
9.   }
10.
11.   static String makeRandomName() {
12.     int x = (int) (Math.random() * 5);
13.     String name = new String[] {"Fluffy", "Fido",
                                    "Rover", "Spike",
                                   "Gigi"}[x];
14.     return name;
```

```
15.   }
16.
17.   public static void main (String [] args) {
18.     Animal a = new Animal();
19.     System.out.println(a.name);
20.     Animal b = new Animal("Zeus");
21.     System.out.println(b.name);
22.   }
23. }
```

Запуск коду 4 рази дасть наступний вихід:
% конструювання програмного забезпечення Animal
Gigi
Zeus

% конструювання програмного забезпечення Animal
Fluffy
Zeus

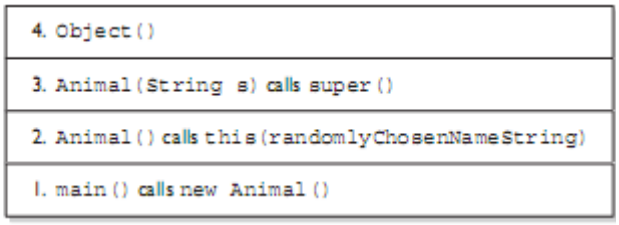
% конструювання програмного забезпечення Animal
Rover
Zeus

% конструювання програмного забезпечення Animal
Fluffy
Zeus

В попередньому коді проходить багато операцій. Схема 2-7 показує стек виклику для викликів конструктора при перезавантаженні конструктора. Подивіться на стек виклику і давайте пройдемось по коду з самого початку.

FIGURE 2-7

Overloaded constructors on the call stack



- Рядок 2. Оголошення імені змінної екземпляру String.
- Рядки 3-5. Конструктор, що використовує String і задає це значення змінній екземпляру.
- Рядок 7. Отут стає цікаво. Уявіть, що кожній тварині потрібне ім'я, але клієнт (код, що викликає) не завжди може знати яке ім'я має бути, тож ви отримаєте довільне ім'я. Без аргументний конструктор згенерує ім'я з допомогою виклику методу makeRandomName().

- Рядок 8. Без аргументний конструктор викликає власний перезавантажений конструктор, що використовує String, в дійсності викликаючи його в той же спосіб, яким воно буде викликане, якщо код клієнта використає new для екземпляризації об'єкта, використовуючи String для імені. Перезавантажений виклик використовує ключове слово this, але використовує його під вглядом імені методу, this(). Тож рядок 8 є просто викликом конструктора з рядка 3, який буде використовувати довільно вибраний String, а не імя, вибране клієнтом.
- Рядок 11. Зауважте, що метод makeRandomName() позначений static!. Це тому, що ви не можете викликати метод екземпляру (іншими словами, нестатичний) чи отримувати доступ до змінної екземпляру, доки не виконає свою роботу надконструктор. А оскільки він викликається конструктором з рядка 3, а не з 7-го, 8-й рядок може використати тільки статичний метод для генерації імені. Якщо б ми хотіли, щоб всі тварини, які не мають специфічного імені, заданого тим, хто викликає, мали просто однакове імя по замовчуванню, скажімо, Fred, тоді рядок 8 міг би прочитати this("Fred"); а не викликати метод, що повертає рядкову змінну з довільно обраним іменем.
- Рядок 12. Він не має нічого спільного з конструкторами, але так як всі ми тут для того щоб вчитись… він генерує випадкове ціле число від 0 до 4.
- Рядок 13. Дивний синтакс, ми знаємо. Ми створюємо новий об'єкт String (просто єдиний екземпляр String), але ми хочемо, щоб рядкова величина вибиралась довільно зі списку. Ми не маємо списку, тож слід його зробити. Тож тут ми:

1. Оголошуємо змінну String, імя.
2. Створюємо масив String (анонімно – ми не присвоюємо йому ніяких значень).
3. знаходимо рядок з індексом [x] (x – довільний номер, згенерований в рядку 12) нового масиву String.
4. Присвоюємо рядкову величину з масиву оголошеному імені змінної екземпляру. Ми можемо зробити це простішим для читання, якщо напишемо так

```
String[] nameList = {"Fluffy", "Fido", "Rover", "Spike",
                     "Gigi"};
String name = nameList[x];
```

Але в чому ж сіль? Вставка незвичного синтаксису (особливо в коді, який непов'язаний з реальним питанням) – це в дусі екзамену. Не будьте вразливими! (Ну добре, будьте, але потім скажіть собі «Хух!» і продовжуйте).

- Рядок 18. Ми викликаємо без аргументну версію конструктора (створюючи довільне імя для використання іншим конструктором).

- Рядок 20. Ми викликаємо перезавантажений конструктор, що вибирає рядкову величину, яка представляє імя.

Ключовим моментом в цьому прикладі коду є рядок 8. Замість того щоб викликати super(), ми викликаємо This(), а this() завжди означає виклик іншого конструктора в тому ж класі. Добре, а що станеться після виклику this()? Рано чи пізно конструктор super() все одно буде викликатись, так? Точно. Виклик this() тільк вікладе незворотне. Якийсь конструктор все ж мусить десь зробити виклик super().

Ключове правило: Першим рядом в конструкторі мусить бути виклик super() або виклик this().

Без винятків. Якщо ви не маєте жодного з них, компілятор вставить свій без аргументний виклик до super(). Іншими словами, якщо конструктор A() має виклик this(), компілятор знає, що конструктор A() не буде викликати super().
Попереднє правило означає, що конструктор ніколи не зможе викликати разом і this(), і super(). Оскільки обидваз цих викликів мають бути першим виразом в конструкторі, ви не можете використовувати їх обох в одному конструкторі. Це також означає, що компілятор не вставить виклик super() в будь-якому конструкторі, де вже є виклик this().
Питання: Що на вашу думку трапиться, якщо ми спробужмо скомпілювати цей код?

```
class A {
  A() {
    this("foo");
  }
  A(String s) {
     this();
  }
}
```

Ваш компілятор не побачить жодних проблем (все залежить від компілятора, але більшість не зауважать). Припускається, що ви знаєте, що робите. Можете вирішити проблему? Оскільки над конструктор має викликатись завжди, куди піде виклик super()? Пам’ятайте, компілятор не буде вставляти конструкто по замовчуванню, якщо ви вже маєте один чи більше конструкторів в класі. І коли компілятор не вставляє конструктор по замовчуванню, він все ж вставляє виклик super() у конструктори, де немає безпосереднього виклику над конструктора, якщо звичайно конструктор вже не має виклику до this(). Тож в попередньому коді, де буде super(). Обидва конструктора в класі мають виклики this() і фактично ви отримуєте те, що мали б, якби ввели наступний код:

```
public void go() {
  doStuff();
```

```
}
public void doStuff() {
  go();
}
```

А тепер ви бачите проблему? Звичайно. Стек вибухає! Він піднімається вище і вище і вище, доки не розривається, код методу падає і вилітаючи з JVM розбивається об підлогу. Два перезавантажені конструктори, які викликають this() – це конструктори, які викликають один одного. Знову, знову й знову, в результаті:

```
% конструювання програмного забезпечення A
Exception in thread "main" конструювання програмного забезпечення.lang.StackOverflowError
```

Перевага наявності перезавантажених конструкторів це те, що ви отримуєте гнучкі шляхи для екземпляризації обжктів з вашого класу. Перевага можливості конструктора викликати інший перезавантажений конструктор в тому, що ви уникаєте дублікації коду. В прикладі Animal, є тільки код задання імені, але уявіть якби після четвертого рядка було ще більше роботи для виконання в конструкторі. Вкладанням одних конструкторів в інші і викликом з допомогою конструктора ви позбавляєте себе необхідності писати і виконувати декілька версій коду конструктора. Простіше, кожен з перезавантажених конструкторів буде викликати інший конструктор, надаючи йому всі потрібні дані (дані, які не забезпечує код клієнта).

Конструктори та екземпляризація стають навіть більш хвилюючими (коли ви думаєте, що вона є безпечною), коли ви добираєтесь до внутрішніх класів, але ми знаємо, що вам досить цікавого в цій частині, тож прибережемо решту обговорення екземпляризації внутрішніх класів до розділу 8.

Статика (Тема 1.3)

1.3. Розробіть код, що оголошує, ініціалізує, і використовує примітиви, масиви, списки, а також об'єкти як статичні, екземпляри та локальні змінні. Використовуйте легальні ідентифікатори для імен змінних.

Статичні змінні та методи

Модифікатор static має настільки глибокі нюанси в поведінці методів та змінних, що ми обговорюємо його як концепцію, що абсолютно відмінна від інших модифікаторів.

Для розуміння того, як працює статичний член, ми спочатку подивимось на причини його використання. Уявіть, що ви маєте клас з методом, який завжди запускається однаково, він обчислює функцію для повернення, скажімо, випадкового числа. Незалежно від того, який екземпляр класу викликає метод, він завжди виконується одним і тим самим чином.

Іншими словами, поведіна методу не залежить від стану (значень змінних екземпляру) об'єкта. Тоді навіщо вам об'єкт, коли метод ніколи не буде специфічним для екземпляра? Чому просто не попросити клас, щоб він сам запустив метод?
Давайте уявимо інший сценарій: Припустимо, ви хочете утримувати лічильник всіх екземплярів з окремого класу. Де ви будете зберігати цю змінну? Вона не буде працювати, якщо ви будете зберігати її як змінну екземпляру всередині класу, екземпляри якого ви рахуєте, оскільки рахунок буе збиватись на значення по замовчуванню з кожним новим екземпляром. Рішення для методу що завжди запускає однаковий сценарій та збереження загального числа екземплярів – використати модифікатор static. Статичні методи та Змінні належать класу, а не частковому екземпляру. Фактично ви використовуєте статичний метод чи змінну без наявності екземплярів класу взагалі. Вам потрібен тільки клас, який може викликати статичний метод чи отримати доступ до статичної змінної. Статичні змінні теж можуть бути доступні без наявності екземпляру класу. Але якщо такі є, статична змінна буде доступна для всіх екземплярів цього класу; буде тільки одна копія. Наступний код оголошує та використовує статичну змінну лічильника:

```
class Frog {
  static int frogCount = 0;  // Declare and initialize
                             // static variable
  public Frog() {
    frogCount += 1;  // Modify the value in the constructor
  }
  public static void main (String [] args) {
    new Frog();
    new Frog();
    new Frog();
    System.out.println("Frog count is now " + frogCount);
  }
}
```

В попередньому коді, статична змінна frogCount встановлюється на нуль, коли клас Frog вперше завантажується JVM, перед тим, як створюються екземпляри Frog. (до речі, вам не треба ініціалізувати статичну змінну як нуль;, статичні змінн отримують ті ж значення по замовчуванню, що й змінні екземплярів). Коли створюється екземпляр Frog, конструктор запускається і збільшує значення frogCount на одиницю. Коли цей код виконується, створюється 3 екземпляри frog в main(), і в результаті маємо:

```
Frog count is now 3
```

Зараз уявіть що станеться, коли frogCount буде змінною екземпляру (іншими словами, нестатичною):

```
class Frog {
```

```
  int frogCount = 0;  // Declare and initialize
                      // instance variable
  public Frog() {
    frogCount += 1;  // Modify the value in the constructor
  }
  public static void main (String [] args) {
    new Frog();
    new Frog();
    new Frog();
    System.out.println("Frog count is now " + frogCount);
  }
}
```

При виконанні коду знову ж створиться 3 екземпляри Frog, але результатом буде… помилка компілятора! Ми не зможемо скомпілювати цей код.

```
Frog.конструювання програмного забезпечення:11: nonstatic variable
frogCount cannot be referenced
from a static context
   System.out.println("Frog count is " + frogCount);
                                        ^
   1 error
```

JVM не знає, frogCount котрого з об'єктів Frog ви хочете отримати. Проблема в тому, що main є сам по собі статичним методом, і це є не запуском будь-якого окремого екземпляру, а запуском на класі як такому. Статичний метод не може мати доступ до нестатичних змінних, оскільки немає екземпляру! Ми не говоримо про відсутність екземплярів класу, але навіть якщо такі є, статичний метод нічого про них не знає. Те саме стосується методів екземплярів, статичний метод не може напряму викликати нестатичний метод. Вважайте, що статичний=клас, нестатичний= екземпляр. Коли ми викликаємо метод з допомогою JVM (main()), статичний метод означає, що JVM не має створювати екземпляр класу для запуску коду.

Увага

Одною з помилкок, які найчастіше роблять початківці в програмуванні Конструювання програмного забезпечення – це спроба доступу до змінної екземпляру (нестатичної змінної) з статичного методу main() (який не знає нічого про екземпляри, тож не може отримати доступ). Наступний код є прикладом неправильного доступу до нестатичної змінної зі статичного методу:

```
class Foo {
   int x = 3;
   public static void main (String [] args) {
     System.out.println("x is " + x);
   }
```

```
}
```

Зрозумійте, цей код не скомпілюється, оскільки ви не можете отримати доступ до нестатичної змінної зі статичного методу. Подумайте про компілятор, що каже «Агов, я не маю зеленого поняття, змінну з якого об'єкта Foo ти хочеш вивести!». Пам'ятайте, це клас, що запускає метод main(), а не екземпляр класу.
Звичайно весь прикол екзамену в тому, що питання не буде виглядати настільки очевидно як в попередньому прикладі. Проблема, на яку вас будуть перевіряти – доступ до нестатичної змінної зі статичного методу – буде схована глибоко в коді, який буде виглядати так, ніби перевіряє щось інше. Для прикладу, попередній код швидше появиться в такому вигляді:

```
class Foo {
   int x = 3;
   float y = 4.3f;
   public static void main (String [] args) {
     for (int z = x; z < ++x; z--, y = y + z)
       // complicated looping and branching code
   }
}
```

Тож поки ви пробуєте слідувати логіці, справжня проблема в тому, що x і y не можуть використовуватись з main(), оскільки вони є нестатичними змінними, змінними екземпляру! Те саме стосується доступу нестатичних методів зі статичного. Правило таке: статичний метод класу не може мати доступ до нестатичного методу чи змінної власного класу.

Доступ до статичних методів та змінних.
Оскільки вам не потрібно екземпляру для виклку статичного методу чи доступу до статичної зміннох, як же ви викличете чи використаєте статичний член? Який синтакс? Ми знаємо, що з регулярним старим методом екземпляру ви використовуєте оператор-крапку для посилання на екземпляр:

```
class Frog {
  int frogSize = 0;
  public int getFrogSize() {
    return frogSize;
  }
  public Frog(int s) {
    frogSize = s;
  }
  public static void main (String [] args) {
    Frog f = new Frog(25);
    System.out.println(f.getFrogSize()); // Access instance
                                         // method using f
```

```
  }
}
```

В попередньому коді ми екземпляризуємо frog, прив'язуючи його до змінної посилання f, а потім використовуємо посилання f для виклику методу на екземплярі Frog, який ми щойно створили. Іншими словами, метод getFrogSize() буде викликаний на специфічному об'єкті Frog в памяті.
Але це наближення (використання посилання на об'єкт) не є призначенням для доступу до статичного методу, оскільки там може не бути екземплярів класу взагалі! Тож, шлях, яким ми отримуємо доступ до статичного методу (чи статичної змінної) – це використання оператора-крапки на імені класу, на противагу використанню його на посилання на екземпляр, як нижче:

```
class Frog {
  static int frogCount = 0;  // Declare and initialize
                             // static variable
  public Frog() {
    frogCount += 1;  // Modify the value in the constructor
  }
}
class TestFrog {
  public static void main (String [] args) {
    new Frog();
    new Frog();
    new Frog();
    System.out.print("frogCount:"+Frog.frogCount); //Access
  }
}
```

Але щоб зробити це ще більш запутаним, Конструювання програмного забезпечення дозволяє вам використовувати змінну посилання обжкта для доступу до статичного члена:

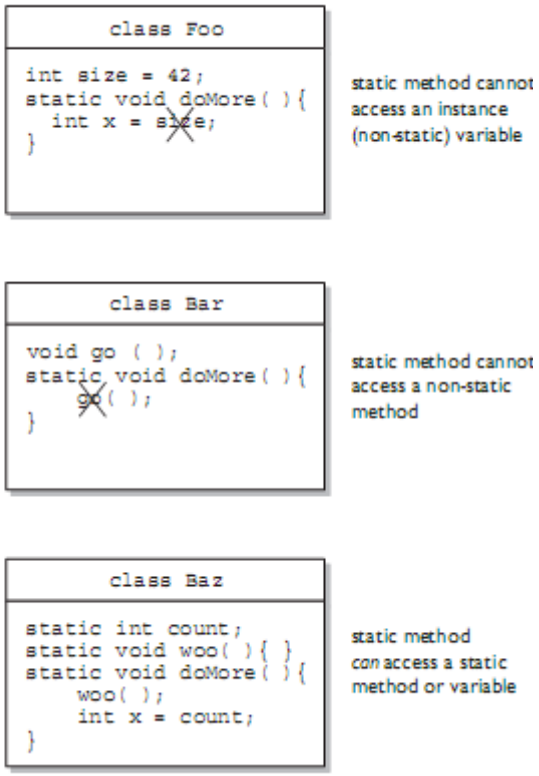
```
Frog f = new Frog();
int frogs = f.frogCount; // Access static variable
                         // FrogCount using f
```

В попередньому коду ми екземпляризуємо Frog, прив'язуємо новий об'єкт Frog до змінної посилання f, а потім використовуємо дане посилання для виклику статичного методу! Але навіть коли ми використовуємо специфічний екземпляр для доступу до статичного методу, правила не змінились. Це тільки фокус з синтаксисом щоб дозволити фам використовувати змінну посилання обжкта (але не обжкт, на який вона посилається) дя доступу до статичного методу чи змінної, але статичний член все ж недоступний окремому екземпляру, що використовувався для

його виклику. В прикладі Frog, компілятор знає що змінна послання f має тип frog, тож статичний метод класу Frog запускається без відома екземпляру Frog в іншому кінці посилання f. Іншими словами, компілятор піклується тільки про те, щоб змінна посилання була оголошена з типом Frog.
Схема 2-8 ілюструє дію статичного модифікатора на методи та змінні.

FIGURE 2-8

The effects of static on methods and variables



Нарешті, запам’ятайте, що статичні методи не можуть бути перевизначені! Це означає, що вони не можуть отримувати нове визначення в підкласах, але це не те саме, що перевизначення. Давайте подивимось на статичний метод, що отримав нове визначення (не перевизначений):

```
class Animal {
  static void doStuff() {
    System.out.print("a ");
  }
}
class Dog extends Animal {
  static void doStuff() {         // it's a redefinition,
                                  // not an override
    System.out.print("d ");
  }
  public static void main(String [] args) {
    Animal [] a = {new Animal(), new Dog(), new Animal()};
    for(int x = 0; x < a.length; x++)
      a[x].doStuff();             // invoke the static method
  }
```

}
Запуск коду дасть нам наступний вихід:
a a a

Пам'ятайте, синтакс a[x].doStuff() є просто скороченням (прийомом синтаксису)… компілятор замінить його на щось типу Animal.doStuff(). Зауважте, що ми не використовували цикл for з Конструювання програмного забезпечення 1.5 (розглянутий в частині 5), навіть коли ми могли. Очікуйте побачити мікс стилів кодування Конструювання програмного забезпечення 1.4 та Конструювання програмного забезпечення 5 на екзамені.

Спряження та привязка
5.1 Розробіть код, що виконує компактну інкапсуляцію, широке спряження та високу прив'язку в класах, та опишіть переваги.

Ми збираємося розглянути це далі. Визначення екзамену для спряження та сполучення є трохи суб'єктивними, тож ми обговоримо їх в цій частині з точки зору екзамену без конкретних визначень для цих двох принципів розробки ОО. Це не буде вивчення вами теми, це буде те, що вам буде потрібне для відповідей на запитання. Ви будете мати дуже мало питань по привязці та спряженню на екзамені.
Ці дві теми, привязка та сполучення мають розглядатись з точки зору розробки ОО. В загальному, хороша розробка ОО включає широке спряження і уникає вузького, а також включає високу привязку і уникає низької. І в більшості дискусій щодо розробки ОО, цілями додатку є
- простота створення
- простота виконання
- простота вдосконалення

Спряження.
Почнемо зі спроби дати визначення спряженню. Спряження – це величина, за якою клас знає про інший клас. Якщо єдина інформація, яку клас А має про клас В – це те, що клас В показується через свій інтерфейс, тоді про класи А та В говорять, що вони широко спряжені… це добре. Якщо навпаки, клас А покладається на частини класу В, що не є частинами інтерфейсу класу В, тоді спряження між ними є вузьким… це погано. Іншими словами, якщо клас І знає більше, ніж він має знати про шлях виконання В, тоді А і В є вузько спряженими.
Для другого сценарію уявіть, що стається коли клас В є посиленим. Це досить ймовірно, що розробник, який підсилив клас В не знає про клас А, навіщо це йому? Розробник класу В має відчувати, що будь-які посилення що не переривають інтерфейс класу мають бути безпечні, тож він може змінити деякі неінтерфейсні частини класу, що спричинить переривання класу А.

В далекому кінці спектру спряження видніється жахлива ситуація, в якій клас А знає деякі речі, що не відносяться до API щодо класу В, А клас В знає щось подібне на рахунок класу А… це ДІЙСНО ПОГАНО. Якщо один з класів зміниться, є ймовірність, що перерветься інший клас. Подивимось на очевидний приклад вузького спряження, який був виключений через бідну інкапсуляцію:

```
class DoTaxes {
  float rate;
  float doColorado() {
    SalesTaxRates str = new SalesTaxRates();
    rate = str.salesRate;     // ouch
                              // this should be a method call:
          // rate = str.getSalesRate("CO");
    // do stuff with rate
  }
}
class SalesTaxRates {
  public float salesRate;            // should be private
  public float adjustedSalesRate;    // should be private
  public float getSalesRate(String region) {
    salesRate = new DoTaxes().doColorado();    // ouch again!
    // do region-based calculations
    return adjustedSalesRate;
  }
}
```

Нетривіальні додатки ОО є міксом багатьох класів та інтерфейсів, які працюють разом. В ідеалі, всі зв’язки між об’єктами в такій системі мають використовувати API, іншими словами, контракти класів, що зважають на об’єкти. Теоретично, якщо всі класи в додатку мають добре розроблені API, то можливо таке, що для всіх між класових операцій будуть використовуватись тільки API. Як ми говорили раніше в цій частині, головна запорука хорошої розробки класу та API – добра інкапсуляція.
Наприкінці скажемо, що спряження є суб’єктивним поняттям. Тому екзамен буде тестувати вас на очевидних прикладах вузького спряження, вас не будуть змішувати приймати спірні рішення.

Прив’язка.
Якщо спряження визначає зв'язок класів між собою, прив’язка характеризує розробку одного класу. Термін прив’язка використовується для визначення ступеня того, наскільки клас має власне, сфокусоване призначення. Пам’ятайте, що прив’язка – суб’єктивне поняття. Більш сфокусований клас – відповідно вища прив’язка – це добре. Ключова перевага високої прив’язки це те, що такі класи зазвичай легші для виконання (і рідше міняються), ніж класи з низькою прив’язкою. Ще одна

перевага це те, що класи з високою прив'язкою можуть бути більш корисними в повторному використанні, ніж інші ласи. Давайте подивимось на приклад псевдо-коду:

```
class BudgetReport {
  void connectToRDBMS(){ }
  void generateBudgetReport() { }
  void saveToFile() { }
  void print() { }
}
```

А тепер уявіть, що ваш начальник підходить і каже: «Ти знаєш той додаток роботи з рахунками? Клієнти тільки що вирішили, що вони хочуть генерувати звіт по прибутках та звіт по інвентарю. Вони також хочуть розширити можливості звітів, впевнитись, що всі звіти дозволятимуть вибирати базу даних, принтер, та зберігати звіти в файли даних…» Холера!

Замість того, щоб вставляти весь код в один файл звіту, нам краще почати розробку спочатку:

```
class BudgetReport {
  Options getReportingOptions() { }
  void generateBudgetReport(Options o) { }
}
class ConnectToRDBMS {
  DBconnection getRDBMS() { }
}
class PrintStuff {
  PrintOptions getPrintOptions() { }
}
class FileSaver {
  SaveOptions getFileSaveOptions() { }
}
```

Цей дизайн є більш прив'язаним. Замість одного класу, що робить все, ми розбили систему на 4 головні класи, кожен з особливою, прив'язаною, роллю. Оскільки ми побудували спеціалізовані, готові для пере використання, класи, буде набагато простіше написати новий звіт, оскільки ми фже маємо клас зєднання з базою даних, клас друку, клас зберігання файлу і це означає, що вони можуть використовуватись іншими класами, що захочуть надрукувати звіт.

Підсумки

Ми почали розділ обговоренням важливості інкапсуляції в хорошому дизайні ОО, а потім ми поговорили про виконання хорошої інкапсуляції. З приватними змінними екземплярів та публічними отримувачами до задавачами.

Потім ми розглянули важливість успадкування, тож ви можете сприймати перевизначення, перезавантаження, поліморфізм, вибір посилання, типи повернення та конструктори.
Ми покрили IS-A та HAS-A. IS-A виконується з використанням успадкування, а HAS-A виконується використанням змінних екземплярів, що посилаються на інші обжкти.
Далі був поліморфізм. Хоча тип змінної посилання не може мінятись, він може використовуватись для посилання на обєкт, тип якого є його власним підтипом. Ми вивчили як визначити які методи є можливими для виклику для даної змінної посилання.
Ми побачили різницю між перевизначеними та перезавантаженими методами, вивчивши, що перевизначений метод виникає, коли підклас успадковує метод з надкласу, і потім перевиконує метод для додання більш спеціалізованої поведінки. Ми вивчили, що під час виконання, Jvm буде викликати версію підкласу на екземплярі класу, а версію надкласу на екземплярі надкласу. Абстрактні методи мусять бути перевизначеними (технічно, вони мають бути виконані, на відміну від перевизначення, оскільки перевизначати нема що).
Ми побачили що перевизначені методи мають оголошувати той же список аргументів та тип повернення (або як в конструювання програмного забезпечення5 вони можуть повертати підтип оголошеного типу повернення перевизначеного методу надкласу), і що модифікатор доступу не може бути більш строгим. Перевизначений метод також не може давати жодних розширених виключень, які не були оголошені в перевизначеному методі. Ви також вивчили, що перевизначений метод може викликатись синтаксисом super.doSomething();.
Перезавантажені методи дозволяють вам перевикористовувати те саме імя методу в класі, але з іншими аргументами (і, можливо, іншим типом повернення). Тоді як перевизначені методи не можуть міняти список аргументів, перезавантажені мусять це робити. Але на відміну від перевизначених, перезавантажені методи можуть варіювати тип повернення, модифікатор доступу та оголошені виключення як собі захочуть.
Ми вивчили механізм вибору (переважно даункастингу), змінних посилань, коли це потрібно та те, як використовувати оператор instanceof.
Далі було виконання інтерфейсів. Інтерфейс описує контракт, якому має слідувати виконуючий клас. Правила виконання є аналогічними до правил розширення абстрактного класу. Також пам’ятайте, що клас може виконувати декілька інтерфейсів, а інтерфейс може розширювати інший інтерфейс. Ми також розглянули типи повернення методу і побачили, що ви можете оголошувати будь-який тип повернення (припускається, що ви маєте доступ до класу для типу повернення посилання об’єкту), якщо ви не пере визначаєте метод. Виключаючи коваріантне повернення, перевизначений метод надкласу має мати той же тип повернення що й метод, що пере визначає. Ми побачили, що перевизначені методи не

можуть міняти тип повернення, тоді як перезавантажені можуть (оскільки вони мінюють список аргументів).
Нарешті, ви вивчили, що можна повертати будь-яке значення чи змінну, яке може бути конвертоване до заданого типу повернення. Тож, для прикладу, може повертатись short, коли оглошеним типом є int. І (припустимо, що horse розширює Animal) посилання horse може повертатись, коли оголошеним типом є Animal.
Ми детально розглянули конструктори, вивчили, що коли ви не забезпечите конструктор класу, то компілятор зробить це за вас. Згенерований таким чином конструктор називається конструктором по замовчуванню, і він є завжди безаргументним конструктором з безаргументним викликом super(). Конструктор по замовчуванню не буде генеруватись, якщо у вашому класі є хоча б один конструктор (незалежно від його аргументів), тож якщо вам потрібен більш ніж один конструктор в класі, ви маєте написати його самі. Ми також побачили, що конструктори не успадковуються і ви можете бути розгублені перед методом, що має те саме імя, що клас (що є легальним). Ознакою методу є тип повернення, тоді як конструктори його не мають. Ми побачили як викликаються конструктори в дереві успадкування обжктів, коли об’єкт екземпляризується використанням new. Ми побачили також, що конструктори можуть бути перевизначені, що означає визначення конструктора з іншим списком аргументів. Конструктор може викликати інший конструктор того ж класу, використанням ключового слова this(), що робить конструктор подібним до методу з іменем this(). Ми бачили, що кожен конструктор мусить мати або this() або super() у першому своєму виразі (хоча компілятор може вставити його замість вас).
Ми розглянули статичні методи та змінні. Статичні члени прив’язані до класу, а не до екземпляру, тож є тільки одна копія одного статичного члена. Звичною помилкою є спроба посилання на змінну екземпляру зі статичного методу. Використовуйте імя класу та оператор-крапку для доступу до статичних членів.
Ми обговорили концепції ОО для спряження ти прив’язки. Широке спряження є бажаним для двох чи більше класів, що взаємодіють між собою через API. Вузье спряження є небажаним станом для двох чи більше класі, які знають внутрішні деталі інших класів, які не входять в API класу. Висока прив’язка є бажаною для окремого класу, використання та спеціалізація якого є обмеженими та сфокусованими.
І, знову ж таки, ви вивчили, що екзамен включає закручені питання, розроблені в основному для того, щоб перевірити ваше вміння розпізнати, наскільки закрученими можуть бути питання.