בנאים והורשה

- בנאים לא נורשים. רק מתודות ושדות
- אם לא מוגדר super מתווסף אחד דיפולטי
- מה ידפיס הקוד הבא?

```
class B {
    B () {System.out.println("B called");}
    int triple (int n) {return n*3;}
}

class C extends B {}

public class Inh {
    public static void main(String[] args) {
        C c = new C();
        System.out.println(c.triple(4));
     }
}
```

```
B called
12
```

code credit: Xah Lee

בנאים והורשה (המשך)

- ומה עם הקוד הזה?

```
class B {
    B (int n) {
        System.out.println("B's constructor called");
    }
}

class C extends B {
    C (int n) {
        System.out.println("C's constructor called");
    }
}

public class I2 {
    public static void main(String[] args) {
        B b = new B(4);
        C c = new C(2);
     }
}
```

```
I2.java:10: error: constructor B in class B cannot be applied to given types;
    C (int n) {
              ^
  required: int
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

בנאים והורשה (המשך ii)

- ומה עם הקוד הזה?

```
class B {
    B (int n) {
        System.out.println("B's constructor called");
    }
}

class C extends B {
    C (int n) {
        super(n);
        System.out.println("C's constructor called");
    }
}

public class I2 {
    public static void main(String[] args) {
        B b = new B(4);
        C c = new C(2);
    }
}
```

code credit: Xah Lee

בנאים והורשה (המשך iii)

- למה בנאים של מחלקה יורשת חייבים לקרוא ל-super? לא חייבים

8

They don't.

If you don't explicitly call a superconstructor, it's equivalent to calling the parameterless superconstructor.

```
public class Sub
{
    public Sub()
    {
        // Do some stuff
    }
}
```

is equivalent to:

```
public class Sub
{
    public Sub()
    {
        super();
        // Do some stuff
    }
}
```

You *do* explicitly have to call a superconstructor if you want to specify arguments. That's pretty reasonable, IMO - would you really want the compiler *guessing* which arguments you wanted to supply?

בנאים והורשה (המשך iv)

- למה בנאים של מחלקה יורשת חייבים לקרוא ל-super? לא חייבים
  - אם לא קוראים "הקומפיילר מכניס" קריאה ל-super()
  - ניתן במקום לקרוא ל-this() או this(params) ואז הקומפיילר לא יכניס
    (ההנחה שבבנאי שקוראים לו תהייה לבסוף קריאה שכזו)

בנאים והורשה (המשך v)

- למה קריאה ל-super לא יכולה להיות באמצע הבנאי?
  כדי למנוע שימוש באובייקט שלא אותחל

```
public MySubClassB extends MyClass {
        public MySubClassB(Object[] myArray) {
                someMethodOnSuper(); //ERROR super not yet constructed
                super(myArray);
        }
}
```

האם ניתן בנאי PRIVATE?

- כן
- למה זה טוב?

```
public class MyClass {
    private final String value;
    private final String type;

    public MyClass(int x){
        this(Integer.toString(x), "int");
    }

    public MyClass(boolean x){
        this(Boolean.toString(x), "boolean");
    }

    public String toString(){
        return value;
    }

    public String getType(){
        return type;
    }

    private MyClass(String value, String type){
        this.value = value;
        this.type = type;
    }
}
```

```java
public class MyClass {
    private final String value;
    private final String type;

    public MyClass(int x){
        this(Integer.toString(x), "int");
    }

    public MyClass(boolean x){
        this(Boolean.toString(x), "boolean");
    }

    public String toString(){
        return value;
    }

    public String getType(){
        return type;
    }

    private MyClass(String value, String type){
        this.value = value;
        this.type = type;
    }
}
```

האם ניתן בנאי PRIVATE?

- סינגלטונים
- מפעלים
- מחלקה של שרותים סטטיים
- מחלקה של קבועים גלובליים

יש בנאים במחלקות ABSTRACT?

```
abstract class Product {
    int multiplyBy;
    public Product( int multiplyBy ) {
        this.multiplyBy = multiplyBy;
    }

    public int mutiply(int val) {
       return multiplyBy * val;
    }
}

class TimesTwo extends Product {
    public TimesTwo() {
        super(2);
    }
}

class TimesWhat extends Product {
    public TimesWhat(int what) {
        super(what);
    }
}
```

- כן
- שימושים כדי לאכוף התנהגות אחידה
- איתחול משותף של שדות
- דאגה לאינוריאנטות

יש בנאים במחלקות ABSTRACT?

```
abstract class Product {
    int multiplyBy;
    public Product( int multiplyBy ) {
        this.multiplyBy = multiplyBy;
    }

    public int mutiply(int val) {
       return multiplyBy * val;
    }
}

class TimesTwo extends Product {
    public TimesTwo() {
        super(2);
    }
}

class TimesWhat extends Product {
    public TimesWhat(int what) {
        super(what);
    }
}
```

- למה PUBLIC?
- היחידים שיכולים לקרוא זה בכל מקרה המחלקות היורשות
- יכול להיות PROTECTED ללא כל שינוי מעשי

טיפוסי זמן ריצה

- בשל הפולימורפיזם ב Java אנו לא יודעים מה הטיפוס המדויק של עצמים
  - הטיפוס הדינאמי עשוי להיות שונה מהטיפוס הסטטי
- בהינתן הטיפוס הדינאמי עשויות להיות פעולות נוספות שניתן לבצע על העצם המוצבע (פעולות שלא הוגדרו בטיפוס הסטטי)
- כדי להפעיל פעולות אלו עלינו לבצע המרת טיפוסים (Casting) על ההפניה

המרת טיפוסים Cast

- המרת טיפוסים בג'אוה נעשית בעזרת אופרטור אונרי שנקרא Cast ונוצר על ידי כתיבת סוגריים מסביב לשם הטיפוס אליו רוצים להמיר.

```
(Type) <Expression>
```

- (הדיון כאן אינו מתייחס לטיפוסים פרימיטיביים).
- הוא מייצר ייחוס מטיפוס Type עבור העצם שהביטוי **<Expression>** מחשב, אם העצם **מתאים** לטיפוס.

- הפעולה מצליחה אם הייחוס שנוצר מתייחס לעצם **מתאים** לטיפוס Type
  - המרה למטה (downcast): המרה של ייחוס לטיפוס פחות כללי, כלומר הטיפוס Type הוא צאצא של הטיפוס הסטטי של העצם.
  - המרה למעלה (upcast): המרה של ייחוס לטיפוס יותר כללי (מחלקה או מנשק)
  - כל המרה אחרת גוררת שגיאת קומפילציה.

- המרה למעלה תמיד מצליחה, ובדרך כלל לא מצריכה אופרטור מפורש; היא פשוט גורמת לקומפיילר לאבד מידע
- המרה למטה עלולה להיכשל: אם בזמן ריצה טיפוס העצם המוצבע לא תואם לטיפוס Type התוכנית תעוף (ייזרק חריג ClassCastException)

Dynamic dispatch vs. static binding

- הפעלת שרותי מופע ב Java היא דינאמית:
  - הקומפיילר לא מציין ל-JVM איזו פונקציה יש להפעיל (רק את החתימה שלה)
  - בזמן ריצה ה JVM מפעיל את השרות המתאים לפי הטיפוס הדינאמי, כלומר לפי טיפוס העצם המוצבע בפועל
- הפעלה דינאמית מכונה לפעמים **וירטואלית**
- הפעלה דינאמית שכזו **איטית יותר** מתהליך שבו הקומפיילר, כחלק מתהליך הקומפילציה, היה מציין איזו פונקציה יש להפעיל ואז לא היה צורך לברר בזמן ריצה מהו הטיפוס הדינאמי ולהסיק מכך מהי הפונקציה שיש להפעיל
- מקרים שבהם הקומפיילר קובע איזו פונקציה תרוץ נקראים static binding (קישור סטטי)

אופטימיזציה: devirtualization

- במקרים מסוימים, כבר בזמן קומפילציה ברור שהטיפוס הדינאמי של הפנייה זהה לטיפוס הסטאטי שלה, ואז אין צורך בהפעלה וירטואלית

- למשל, בקוד:

```
MyClass o = new MyClass();
o.method1(5); // clearly o is a member of MyClass
```

- ואולם לא את כל המקרים האלה יודע **הקומפיילר** לזהות

- יש מקרים שכן:
  - אם `MyClass` מוגדר `final`
  - או שהשירות `method1` מוגדר במחלקה `final`; זה מונע דריסה שלו
  - הפעלת שרות `private`
  - הפעלת בנאים
  - הפעלת שרות `super`
  - הפעלת שרותי מחלקה (static method, כפי שמרמז שמם...)

- במקרים כאלה, הקומפיילר יכול לבצע devirtualization ולהורות ל JVM איזו פונקציה להפעיל

```java
public class Animal {
    public static void hide() {
        System.out.format("The hide method in Animal.%n");
    }
    public void override() {
        System.out.format("The override method in Animal.%n");
    }
}
```

```java
public class Cat extends Animal {
    public static void hide() {
        System.out.format("The hide method in Cat.%n");
    }
    public void override() {
        System.out.format("The override method in Cat.%n");
    }
 }
```

```java
public class Client{
    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        //myAnimal.hide();    //BAD STYLE
        Animal.hide();        //Better!
        myAnimal.override();
    }
}
```

מה יודפס?

```
The hide method in Animal.
The override method in Cat.
```

```java
public class Base {
    private void priv() { System.out.println("priv in Base"); }
    public void pub() { System.out.println("pub in Base"); }

    public void foo() {
       priv();
       pub();
    }
}
```

```java
public class Sub extends Base {
    private void priv() { System.out.println("priv in Sub"); }
    public void pub() { System.out.println("pub in Sub"); }
}
```

```java
public class Test {

     public static void main(String[] args) {
          Base b = new Sub();
          b.foo();
     }
}
```

מה יודפס?

```
priv in Base
pub in Sub
```

שדות, הורשה וקישור סטטי

- גם קומפילציה של התייחסויות לשדות מתבצעת בצורה סטטית
- מחלקה יורשת יכולה להגדיר שדה גם אם שדה בשם זה היה קיים במחלקת הבסיס (מאותו טיפוס או טיפוס אחר)

```
public class Base {
        public int i = 5;
}
```

```
public class Sub extends Base {
        public String i = "five";
}
```

```
5
five
5
```

```
public class Test {

  public static void main(String[] args) {
     Base bb = new Base();
     Sub ss = new Sub();
     Base bs = new Sub();

     System.out.println(bb.i);
     System.out.println(ss.i);
     System.out.println(bs.i);
  }
}
```

מה יודפס?

העמסה והורשה

- במקרים של העמסה **הקומפיילר** מחליט איזו גרסה תרוץ (יותר נכון: איזו גרסה לא תרוץ)
- זה נראה סביר (הפרוצדורות מתוך `java.lang.String`):

```
static String valueOf(double d)      {…}
static String valueOf(boolean b)     {…}
```

- אבל מה עם זה?

```
overloaded(Rectangle          x) {…}
overloaded(ColoredRectangle x) {…}
```

- לא נורא, הקומפיילר יכול להחליט,

```
Rectangle          r = new ColoredRectangle ();
ColoredRectangle cr = new ColoredRectangle ();
overloaded(r); // we must use the more general method
overloaded(cr); // The more specific method applies
```

העמסה והורשה

- **אבל זה כבר מוגזם:**

```
overTheTop(Rectangle x, ColoredRectangle y) {…}
overTheTop(ColoredRectangle x, Rectangle y) {…}

ColoredRectangle a = new ColoredRectangle ();
ColoredRectangle b = new ColoredRectangle ();
overTheTop(a, b);
```

- ברור שנדרשת המרה (casting)אבל של איזה פרמטר? a או b?
- אין דרך להחליט; הפעלת השגרה לא חוקית בג'אווה

העמסה והורשה - שבריריות

```
overTheTop(Rectangle x, ColoredRectangle y) {…}
overTheTop(ColoredRectangle x, Rectangle y) {…}

ColoredRectangle a = new ColoredRectangle ();
ColoredRectangle b = new ColoredRectangle ();
overTheTop(a, b);
```

- אם הייתה רק הגרסה הירוקה, הקריאה לשגרה הייתה חוקית
- כאשר מוסיפים את הגרסה הסגולה, הקריאה נהפכת ללא חוקית; אבל הקומפיילר לא יגלה את זה אם זה בקובץ אחר, והתוכנית תמשיך לעבוד, ולקרוא לגרסה הירוקה
- לא טוב שקומפילציה רק של קובץ שלא השתנה תשנה את התנהגות התוכנית; זה מצב **שבריר**י

העמסה והורשה - יותר גרוע

```
class B {
  overloaded(Rectangle        x) {…}
}

class S extends B {
   overloaded(Rectangle         x) {…} // override
   overloaded(ColoredRectangle x) {…} // overload but no override!
}

S o = new S();
ColoredRectangle cr = ...
o.overloaded( cr );        // invoke the purple
((B) o).overloaded( cr )   // What to invoke?
```

העמסה והורשה - יותר גרוע

```
class B {
  overloaded(Rectangle       x) {…}
}

class S extends B {
      overloaded(Rectangle        x) {…} // override
      overloaded(ColoredRectangle x) {…} // overload but no override!
}

S o = new S();
ColoredRectangle cr = ...
o.overloaded( cr );       // invoke the purple
((B) o).overloaded( cr )  // What to invoke?
```

- מנגנון ההעמסה הוא סטטי: בוחר את החתימה של השרות (טיפוס העצם, שם השרות, מספר וסוג הפרמטרים), אבל עדיין לא קובע איזה שירות ייקרא.
- עבור הקריאה `((B) o).overloaded( cr )` תיבחר החתימה:

`B.overloaded(Rectangle)`

- בגלל שיעד הקריאה הוא מטיפוס B השרות היחיד הרלבנטי הוא האדום!
- בזמן ריצה מופעל מנגנון השיגור הדינמי, שבוחר בין השרותים בעלי חתימה זאת, את המתאים ביותר, לטיפוס הדינמי של יעד הקריאה. הטיפוס הדינמי הוא S , לכן נבחר השרות הירוק.
- כנ"ל אם הקריאה היא: `B b = new S(); b.overloaded( cr )`

העמסה זה רע

- אם עוד לא השתכנעתם שהעמסה היא רעיון מסוכן, אז עכשיו זה הזמן
- בייחוד כאשר ההעמסה היא ביחס לטיפוסים שמרחיבים זה את זה, לא זרים לחלוטין
- יוצר שבריריות, קוד שמתנהג בצורה לא אינטואיטיבית (השירות שעצם מפעיל תלוי בטיפוס ההתייחסות לעצם ולא רק במחלקה של העצם), וקושי לדעת איזה שירות בדיוק מופעל
- ומכיוון שהתמורה היחידה (אם בכלל) היא אסתטית, לא כדאי