Inheritance, Polymorphism and Abstract Classes

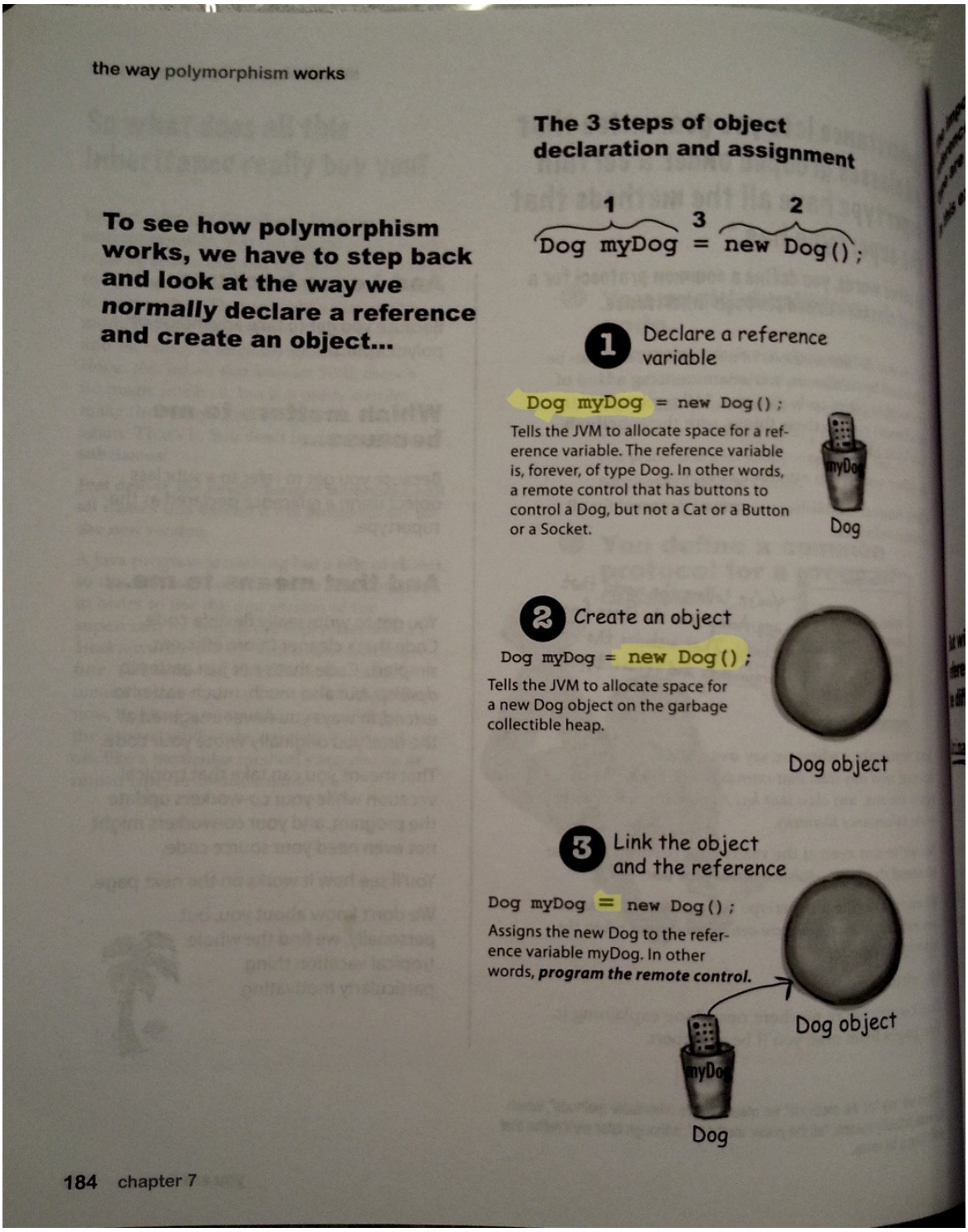
As you know, when one class extends another, it inherits all of the parent class' methods and variables.

** The original variables should not be recreated in the child class as this leads to shadow variables.*

** Methods are overridden, not variables. * "Fields" is the term for both methods and variables.*

First we need to review what happens when we create an object.

These photos are of the excellent book "Head First Java".



the way polymorphism works

To see how polymorphism works, we have to step back and look at the way we *normally* declare a reference and create an object...

The 3 steps of object declaration and assignment

1 3 2
`Dog myDog = new Dog();`

1 Declare a reference variable

`Dog myDog = new Dog();`

Tells the JVM to allocate space for a reference variable. The reference variable is, forever, of type Dog. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket.

Dog

2 Create an object

`Dog myDog = new Dog();`

Tells the JVM to allocate space for a new Dog object on the garbage collectible heap.

Dog object

3 Link the object and the reference

`Dog myDog = new Dog();`

Assigns the new Dog to the reference variable myDog. In other words, ***program the remote control.***

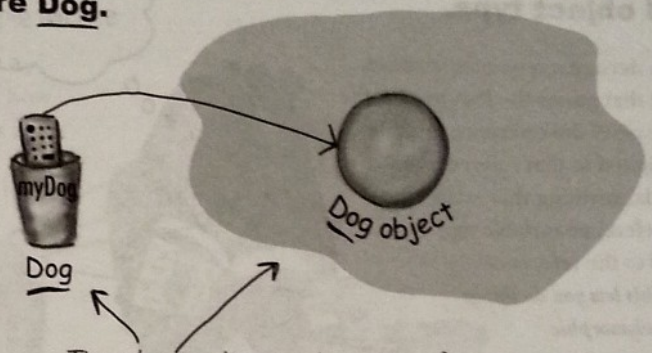
Dog object

Dog

184 chapter 7

The important point is that the reference type AND the object type are the same.

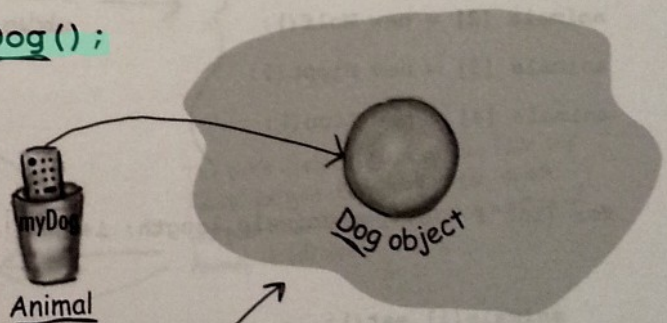
In this example, both are Dog.



These two are the same type. The reference variable type is declared as Dog, and the object is created as new Dog().

But with polymorphism, the reference and the object can be *different*.

```
Animal myDog = new Dog();
```



These two are NOT the same type. The reference variable type is declared as Animal, but the object is created as new Dog().

Let’s look at some code:

```java
import java.awt.Color;
import java.awt.Rectangle;

public class Inheritance {

	class Ball extends Rectangle {
		Color colour = Color.RED;
		Ball(int i, int j, int k, int l) {
			super(i,j,k,l);
		}
	}

	void main(){

		//create a ball object
		Ball b = new Ball(1,2,3,4);
		b.colour = Color.GREEN;

//The object type is still Ball, but now we assign it to a reference
variable of type Rectangle (its parent)
		Rectangle r = b;

		//The actual object is still a ball ** see printout below
		System.out.println(r.toString());

//But Java accesses it through a rectangle, so it cannot see the
colour variable
		System.out.println(r.colour);  //<<< causes a compile error

//We cast it to a Ball object because we KNOW that it really is a
Ball.
		Ball b2 = (Ball) r;
		//And it still has its green colour ** see printout below
		System.out.println(b2.colour);

//But a normal Rectangle is NOT a Ball. We can cast it, but then the
program crashes when it runs. ** See error below.
		//This Rectangle is NOT a Ball object.
		Rectangle rr = new Rectangle(9,8,7,6);
		Ball bb = (Ball) rr;
		System.out.println(bb.colour);

	}

	public static void main(String[] args) {
		new Inheritance().main();
	}
}
```

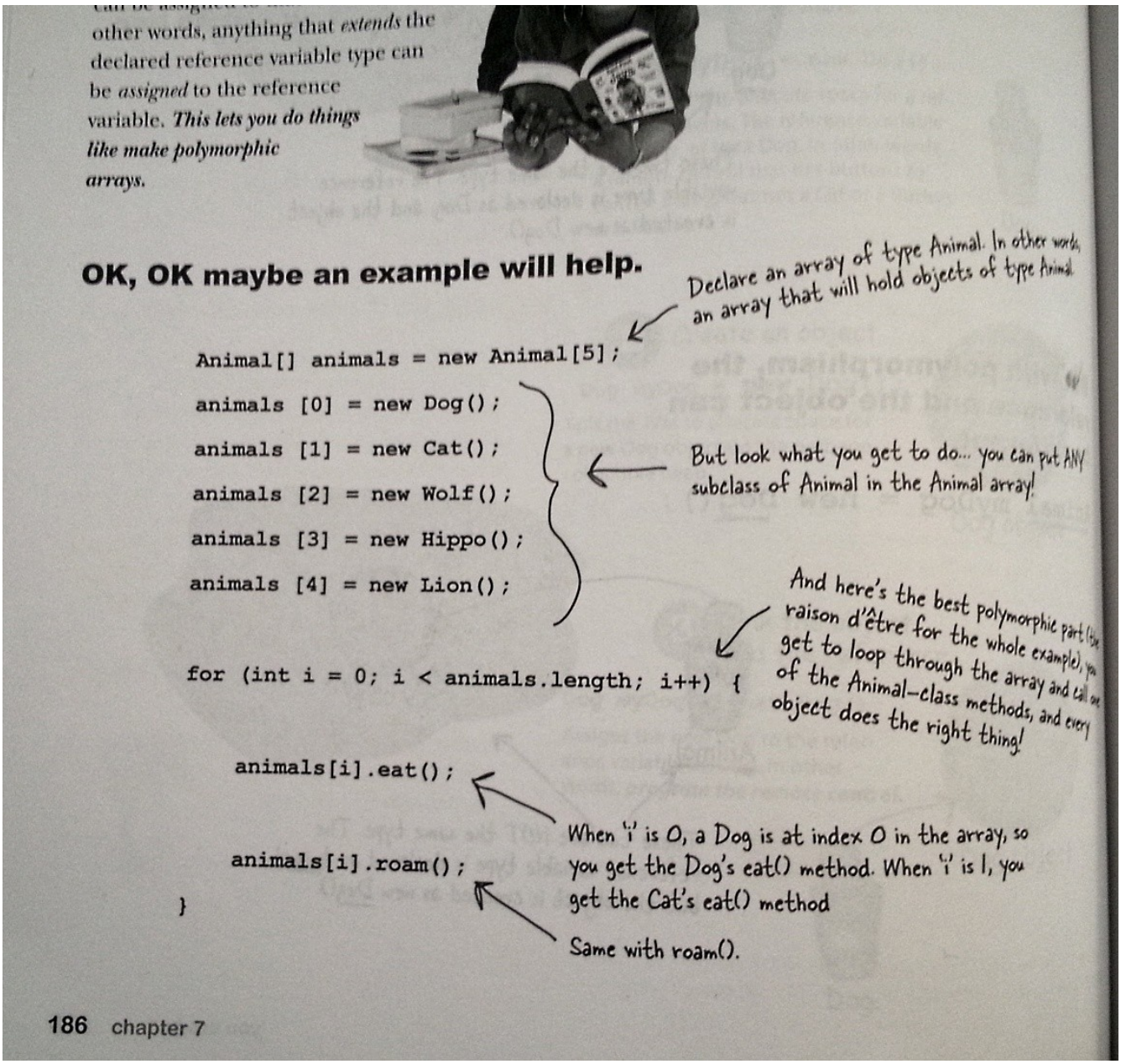
```
---------------------- CONSOLE OUTPUT ----------------------------

//The actual object is still a ball **
Inheritance$Ball[x=1,y=2,width=3,height=4]

//And it still has its green colour **
java.awt.Color[r=0,g=255,b=0]

Exception in thread "main" java.lang.ClassCastException:
java.awt.Rectangle cannot be cast to Inheritance$Ball
     at Inheritance.main(Inheritance.java:33)
     at Inheritance.main(Inheritance.java:39)

------------------------------------------------------------------
```

To summarize:

Class Child extends Parent {}
Child cc = new Child();

Parent pp = cc;	child can always be accessed via parent object (Polymorphism – child acts as if it is the parent) But when the child is accessed this way, you cannot access extra **fields** in the Child object.
Child c2 = (Child) pp;	parent can be cast to child object only if it really is a child All Child **fields** *(methods and variables)* are now accessible.

Why would anyone want to do this? Here's an example.



other words, anything that *extends* the declared reference variable type can be *assigned* to the reference variable. ***This lets you do things like make polymorphic arrays.***

OK, OK maybe an example will help.

Declare an array of type Animal. In other words, an array that will hold objects of type Animal.

```
Animal[] animals = new Animal[5];
animals [0] = new Dog();
animals [1] = new Cat();
animals [2] = new Wolf();
animals [3] = new Hippo();
animals [4] = new Lion();
```

But look what you get to do... you can put ANY subclass of Animal in the Animal array!

And here's the best polymorphic part (the raison d'être for the whole example), you get to loop through the array and call one of the Animal-class methods, and every object does the right thing!

```
for (int i = 0; i < animals.length; i++) {
    animals[i].eat();
    animals[i].roam();
}
```

When 'i' is 0, a Dog is at index 0 in the array, so you get the Dog's eat() method. When 'i' is 1, you get the Cat's eat() method

Same with roam().

186 chapter 7

Finally, if you want to make sure that no one EVER creates and Animal object, you make the Animal class to be **ABSTRACT**.

abstract class Animal {}

Now you can never create an Animal object, but you can still extend it and create Dog, Cat, Lion, Hippo, Wolf objects. We can still use the Animal type to hold the various subclasses of animals as shown in the photo above.