Module 5
OOP in Java

Local and instance variables

1 **Instance** variables are declared inside a class but not within a method.

```
class Horse {
   private double height = 15.2;
   private String breed;
   // more code...
}
```

2 **Local** variables are declared within a method.

```
class AddThing {
   int a;
   int b = 12;

   public int add() {
      int total = a + b;
      return total;
   }
}
```

3 **Local** variables MUST be initialized before use!

```
class Foo {
   public void go() {
      int x;
      int z = x + 3;
   }
}
```

Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out.

```
File Edit Window Help Yikes
% javac Foo.java
Foo.java:4: variable x might
not have been initialized
      int z = x + 3;
1 error       ^
```

Java Beans

```
public class Clock {
        private String time;

        public void setTime(String time) {
                time = this.time;
        }

        public String getTime() {
                return time;
        }
}

class ClockTestDrive {
        public static void main(String [] args) {
                Clock c = new Clock();
                c.setTime("1245");
                String tod = c.getTime();
                System.out.println("time: " + tod);
        }
}
```

Access modifiers

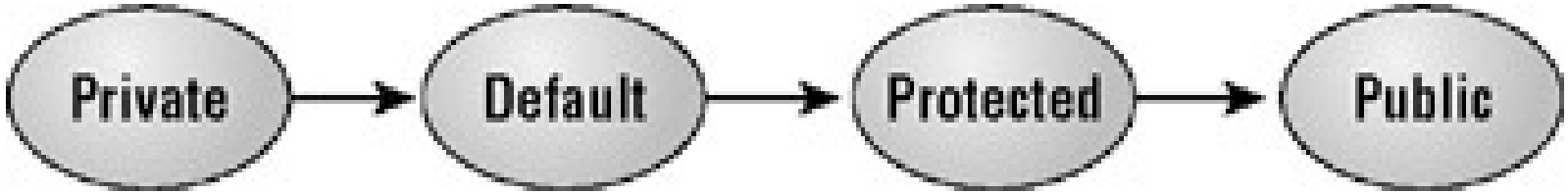
- Access modifiers:
  - public
  - protected
  - default (friendly, package)
  - private
- Only one modifier can be used:

```
class Parser {…}
public class EightDimensionalComples { … }
private int i;
protected double getChiSquared() {…}
private class Horse {…}
Button getBtn(){…}
```

Access modifiers

- When overriding a method you can't make its scope less visible.

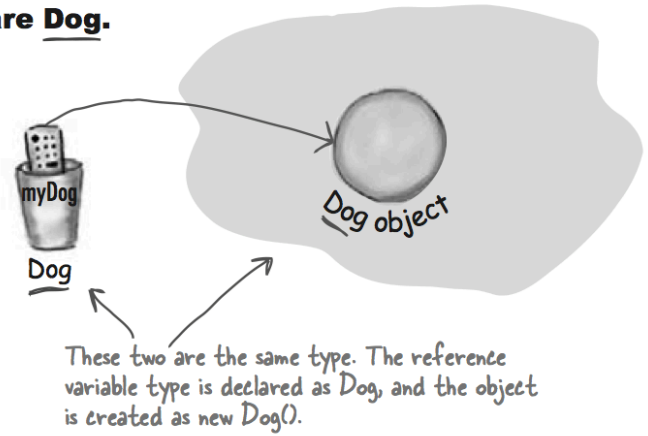
- Visibility scope of overridden method can be the same as visibility scope of an overriding method or even higher.



Using polymorphism

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

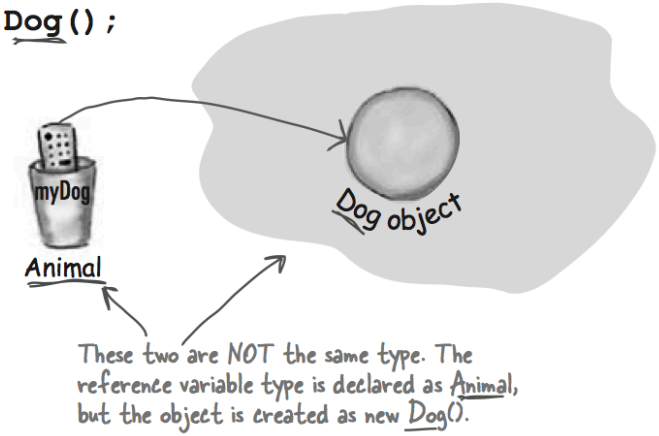
In this example, both are Dog.




```
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();

for (int i = 0; i < animals.length; i++)

    animals[i].eat();

    animals[i].roam();
}
```

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

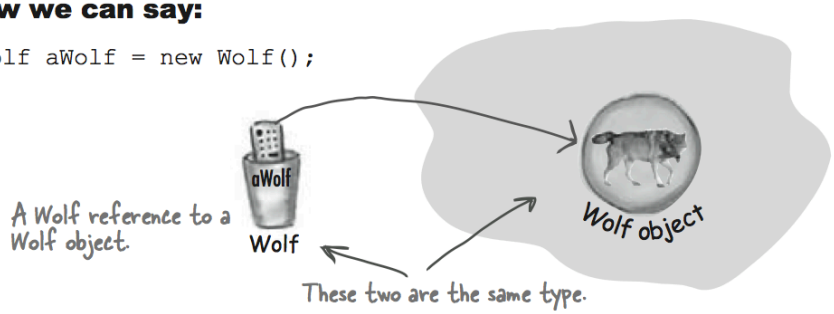
```
Animal myDog = new Dog();
```



Abstract classes

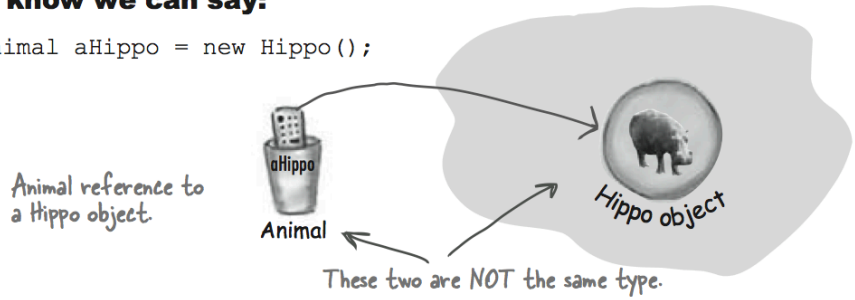
We know we can say:

```
Wolf aWolf = new Wolf();
```



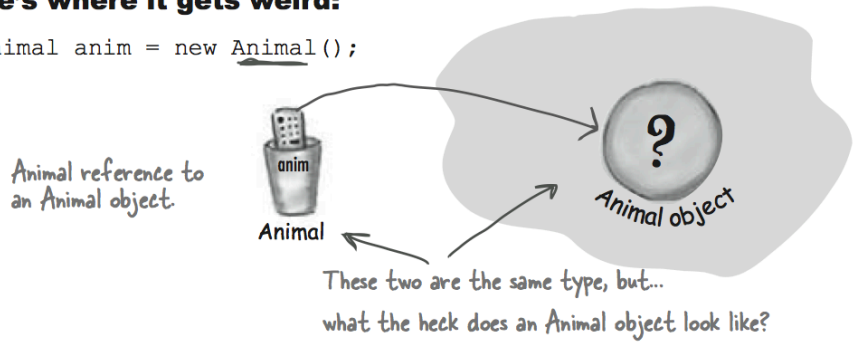
And we know we can say:

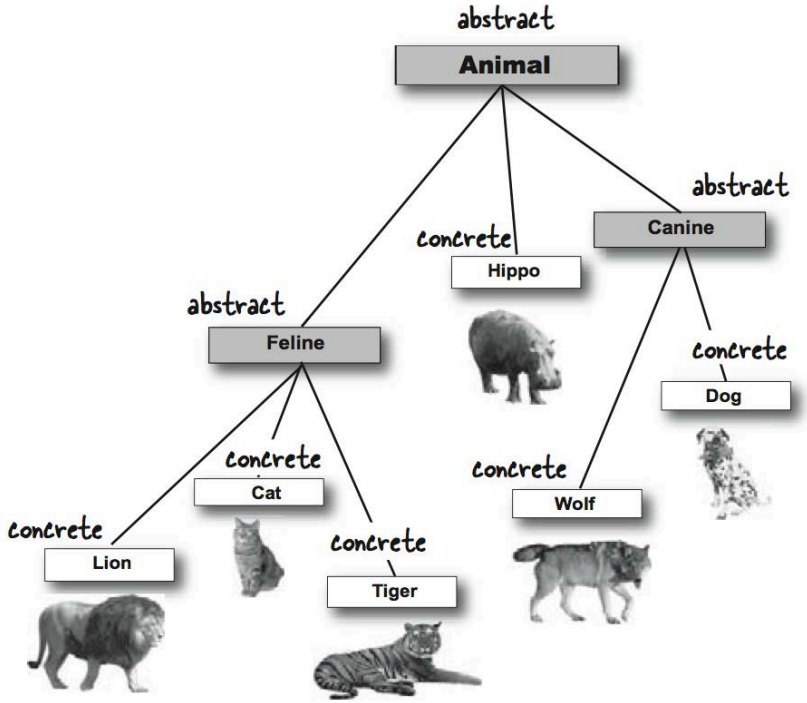
```
Animal aHippo = new Hippo();
```



But here's where it gets weird:

```
Animal anim = new Animal();
```



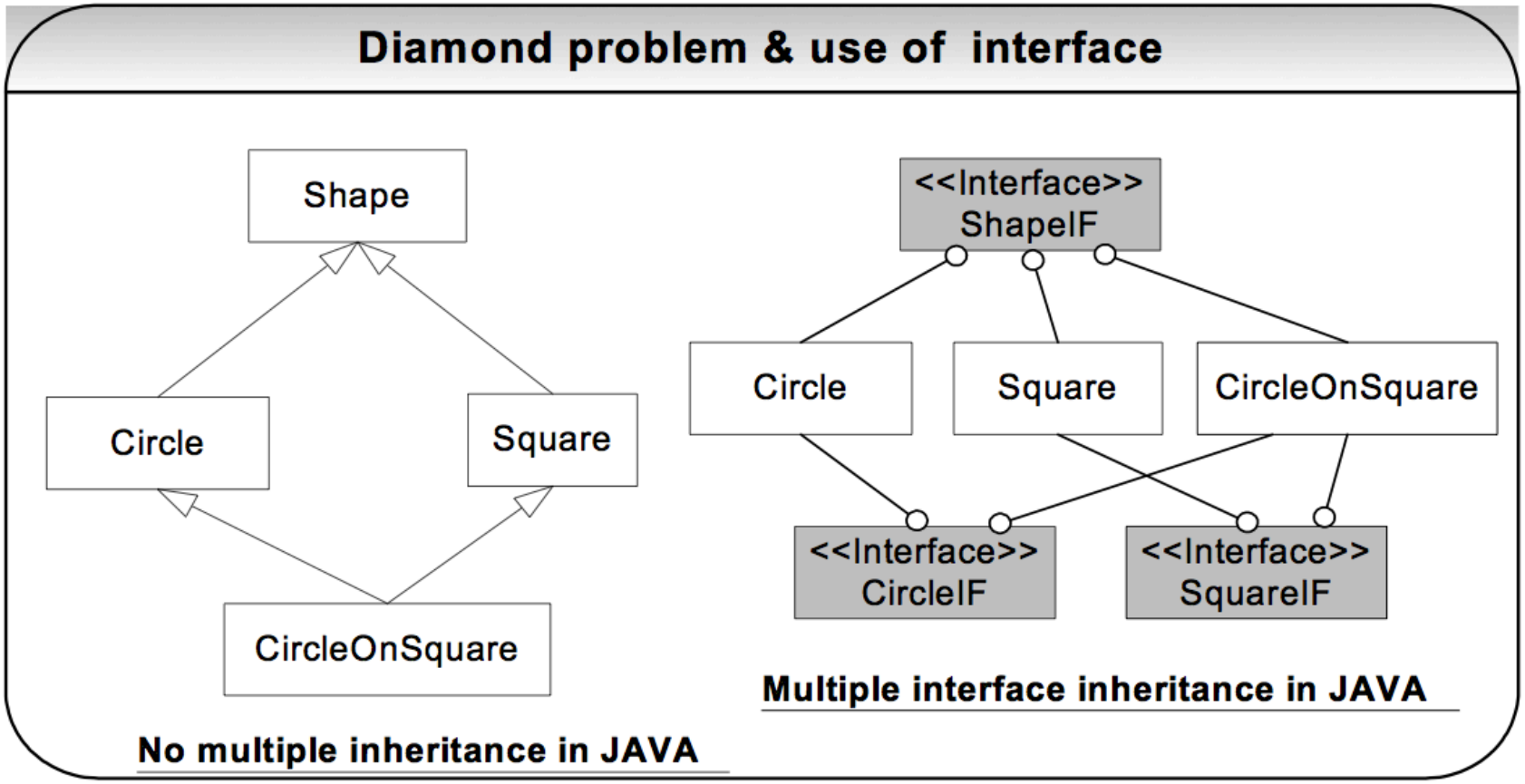



```
abstract class Canine extends Animal {
    public void roam() { }
}
```

```
public abstract void eat();
```

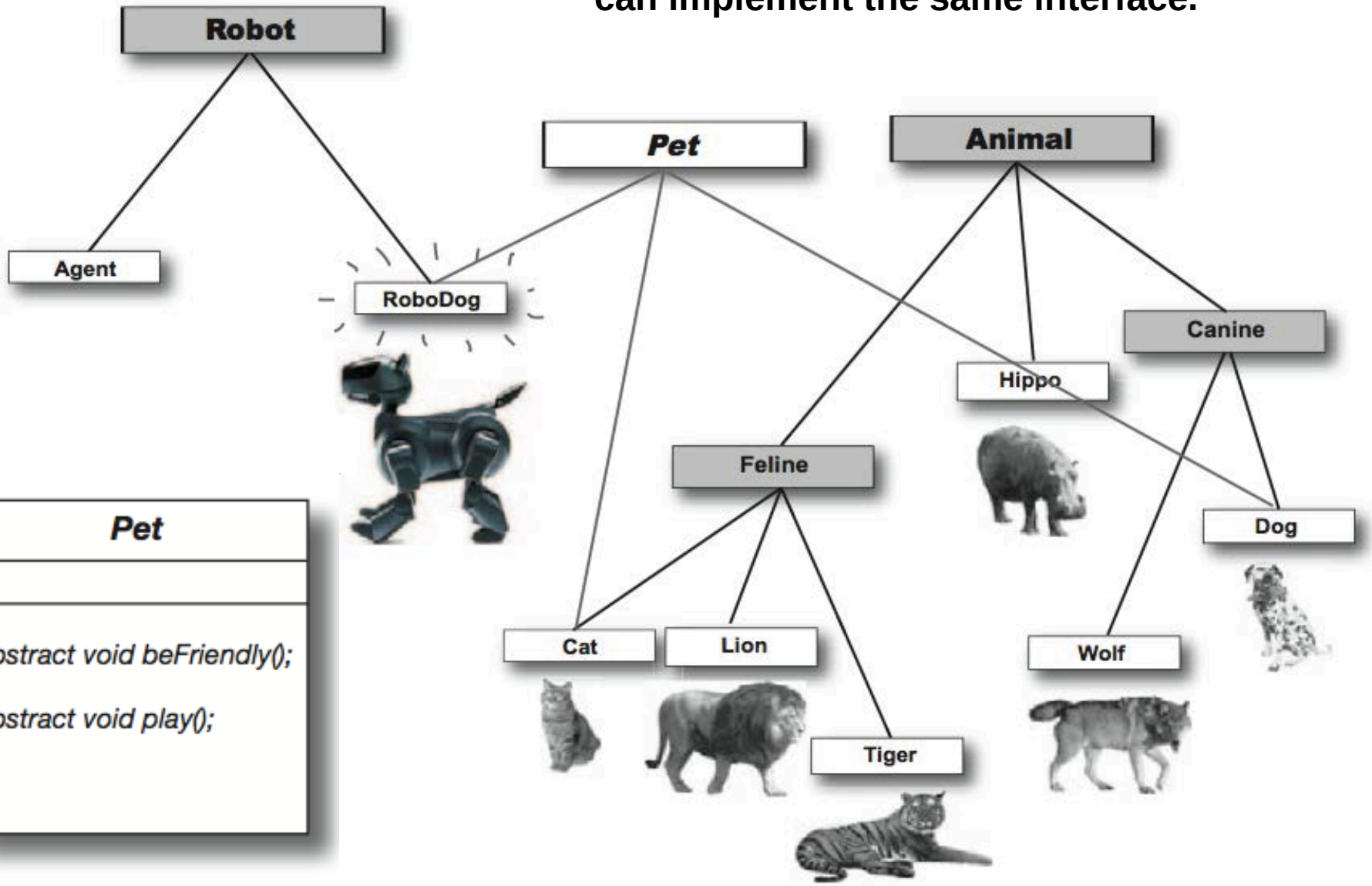
No method body!
End it with a semicolon.

Multiple inheritance



Using interfaces

Classes from different inheritance trees can implement the same interface.



Using interfaces

```
public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
}
```

```
public class Dog extends Canine implements Pet {
    public void beFriendly() {...}

    public void play() {...}

    public void roam() {...}

    public void eat() {...}
}
```

Better still, a class can implement multiple interfaces:

```
public class Dog extends Animal
        implements Pet, Saveable, Paintable { ... }
```

Interfaces

- Interface forms a contract between the client code and a class that implements this interface.

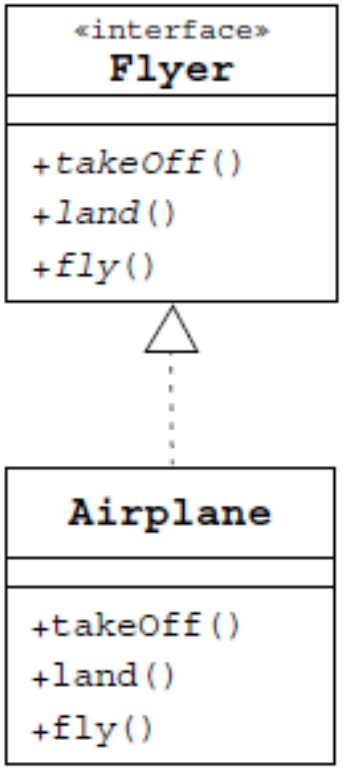
- An interface may be considered as an abstract class whose methods are all abstract.

- Java interface is declared with the `interface` keyword.

- A class that implements an interface contains the `implements` clause in the class declaration.

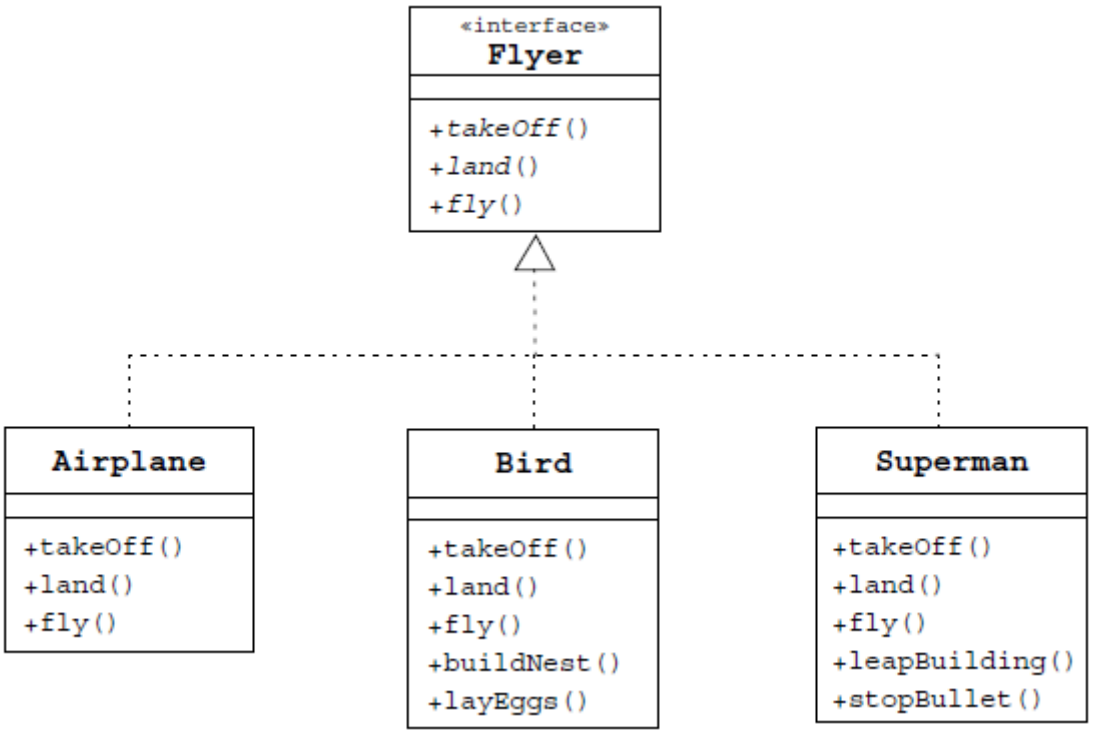
Interfaces

```
public interface Flyer {
    public void takeOff();
    public void land();
    public void fly();
}

public class Airplane implements Flyer {
    public void takeOff() {
        // accelerate until lift-off
        // raise landing gear
    }
    public void land() {
    }
    public void fly() {

    }
}
```



Interfaces

- As long as an interface is a specification of certain behavior, the class can implement several interfaces:



Interfaces

```
public interface Flyer {
    public void fly();
}
public interface Swimer {
    public void swim();
}
public class Penguin implements Swimer {
    public void swim() {
        // A penguin is able to swim, but not able to fly
    }
}
public class Duck implements Flyer, Swimer {
    public void fly() {

    }
    public void swim() {

    }
}
```

Interfaces

- As long as an interface is a contract rather than an implementation:
  - It cannot be instantiated
  - There are no constructors
  - There is no instance data

Note! An interface can contain static final data

Interfaces

- The `public static final` modifier is optional.

```
interface Flyer {
   public final static int NB_WINGS = 2;
   void fly();
}
// This equals:
interface Flyer {
   int NB_WINGS = 2;
   void fly();
}
```

Interfaces

- An interface that is not nested cannot be **`private.`**
- An interface cannot be **`protected.`**
- A **`public`** interface can be implemented by any class.
- A default interface can be implemented by any class from the package that defines the interface itself.

Interfaces

- It is assumed that all interface methods are declared as `public abstract`.

```
interface Flyer {
    public final static int NB_WINGS = 2;
    public astract void fly();
}
// This equals to:
interface Flyer {
    int NB_WINGS = 2;
    void fly();
}
```

Interfaces

- When overriding the method you cannot reduce its visibility.

```
public class Penguin implements Swimer {
   void swim() {
      // Compiler Error! Cannot reduce the
visibility of the inherited method from Swimer
   }
}
```

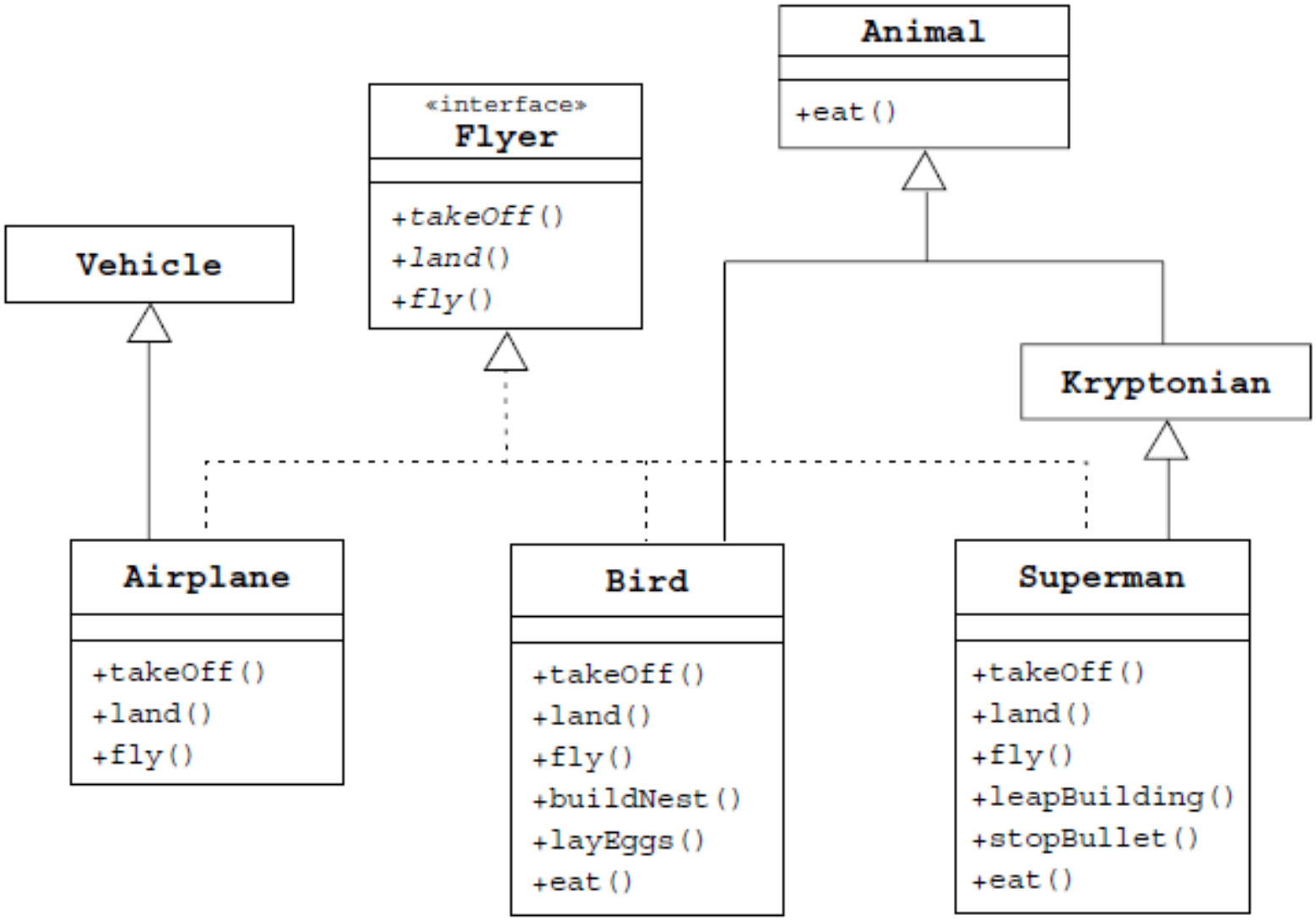
Interfaces

- A behavior can extend another behavior.

```
public interface Set extends Collection, Comparator
{
   ...
}
```

Note! the class that implements an extended interface must implement methods of both interfaces.

Complex example



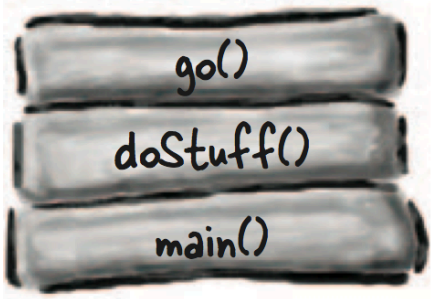
Complex example

```
public class Bird extends Animal implements Flyer {
    public void takeOff() { /* take-off implementation */ }
    public void land() { /* landing implementation */ }
    public void fly() { /* fly implementation */ }
    public void buildNest() { /* nest building behavior */
    public void layEggs() { /* egg laying behavior */ }
    public void eat() { /* override eating behavior */ }
}
```

The stack and the heap

The Stack

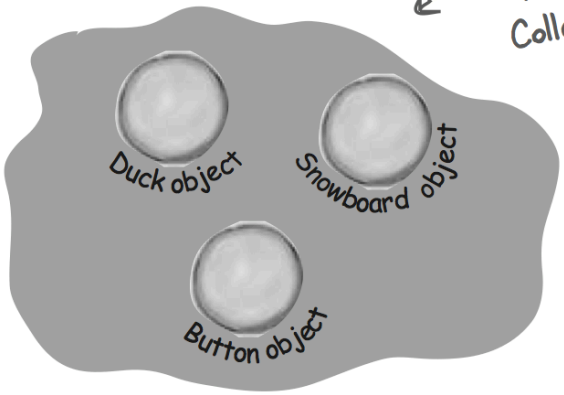
Where method invocations and local variables live



The Heap

Where ***ALL*** objects live

also known as "The Garbage-Collectible Heap"



Instance Variables

Instance variables are declared inside a *class* but *not* inside a method. They represent the "fields" that each individual object has (which can be filled with different values for each instance of the class). Instance variables live inside the object they belong to.

```
public class Duck {

    int size;

}
```

Every Duck has a "size" instance variable.

Local Variables

Local variables are declared inside a *method,* including method parameters. They're temporary, and live only as long as the method is on the stack (in other words, as long as the method has not reached the closing curly brace).

```
public void foo(int x) {

    int i = x + 3;

    boolean b = true;

}
```

The parameter x and the variables i and b are all local variables.

Constructors

- It should be ensured that child constructor will be called after parent constructor.

- The `super(arg1, …)` keyword is used to call superclass constructor.

- The needed constructor is selected by the list of the arguments.

Constructors

- The `super(arg1, …)` keyword is used to call superclass constructor.

```
class SuperClass {
      public SuperClass(int foo) {
        // do something with foo
      }
}

class SubClass extends SuperClass
{
      public SubClass(int foo, int bar)
      {
            super(foo);
            // do something with bar
      }
}
```

Constructors

The `this` reference can be used to call overloaded constructor.

```
public Employee(String name, double salary, Date birth)
{
}
public Employee(String name, double salary) {
    this(name, salary, null);
}
public Employee(String name, Date birth) {
    this(name, 0, birth);
}
```

Method overloading

```
public void aMethod(String s) { }
public void aMethod() { }
public void aMethod(int i, String s) { }
public void aMethod(String s, int i) { }
```

- Only the type of an argument is considered, not a parameter name. Therefore, the following method is not considered overloaded:

```
public void aMethod(int j, String name) { }
```

Method overriding

- Overriding method must:
  - Have the same name and arguments list as parent class method
  - The return type should be the same or its subclass.
- Requirements to overridden method:
  - `final` cannot be overridden
  - Access modifier shall not be narrower
  - Overridden method should throw checked exceptions of the same type or subclass

Method overriding example

```
class TheSuperclass {
   Number getValue() throws Exception{
      return new Long(1);
   }
}

class TheSubclass extends TheSuperclass {
   Long getValue() throws IOException {
      return new Long(2);
   }
}

class Main {
   public static void main(String[] args) {
      Superclass s;
      s = new Superclass();
      s.getValue(); // 1
      s = new Subclass();
      s.getValue(); // 2
   }
}
```

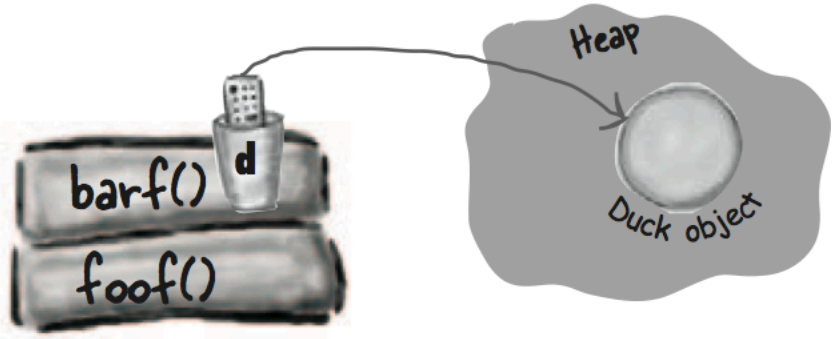
Object killer #1

Reference goes out of scope, permanently.

```
public class StackRef {
        public void foof() {
                barf();
        }
        public void barf() {
                Duck d = new Duck();
        }
}
```

1. *foof()* is pushed onto the Stack, no variables are declared.

2. *barf()* is pushed onto the Stack, where it declares a reference variable, and creates a new object assigned to that reference. The object is created on the Heap, and the reference is alive and in scope.



3. *barf()* completes and pops off the Stack. Its frame disintegrates, so 'd' is now dead and gone. Execution returns to *foof()*, but *foof()* can't use 'd'.

Object killers #2 and #3

Assign the reference to another object

```
public class ReRef {
        Duck d = new Duck();

        public void go() {
                d = new Duck();
        }
}
```

Explicitly set the reference to null

```
public class ReRef {
        Duck d = new Duck();
        public void go() {
                d = null;
        }
}
```

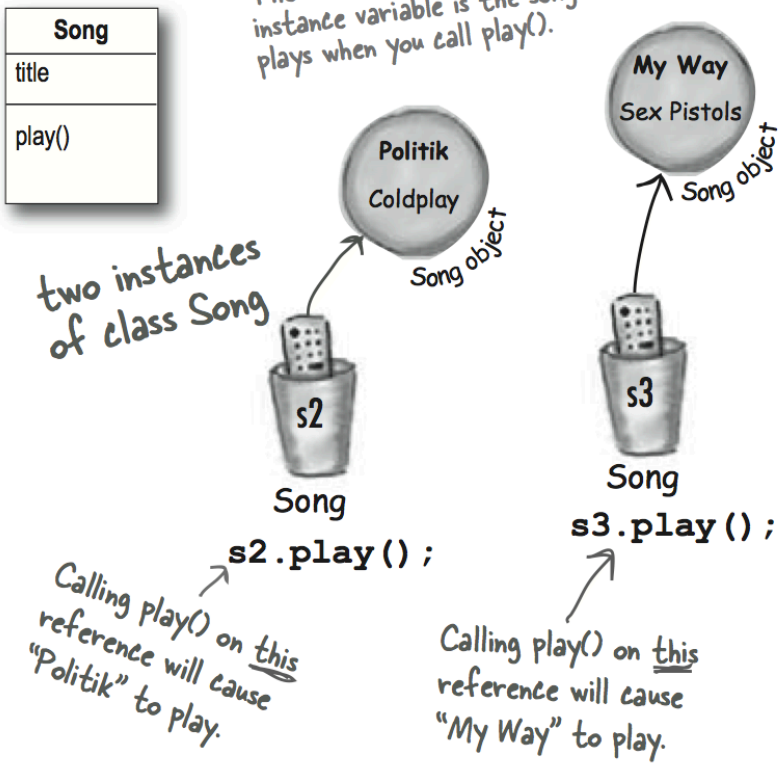
Static methods

regular (non-static) method

```
public class Song {
   String title;
   public Song(String t) {
     title = t;
   }
   public void play() {
      SoundPlayer player = new SoundPlayer();
      player.playSound(title);
   }
```

Instance variable value affects the behavior of the play() method.

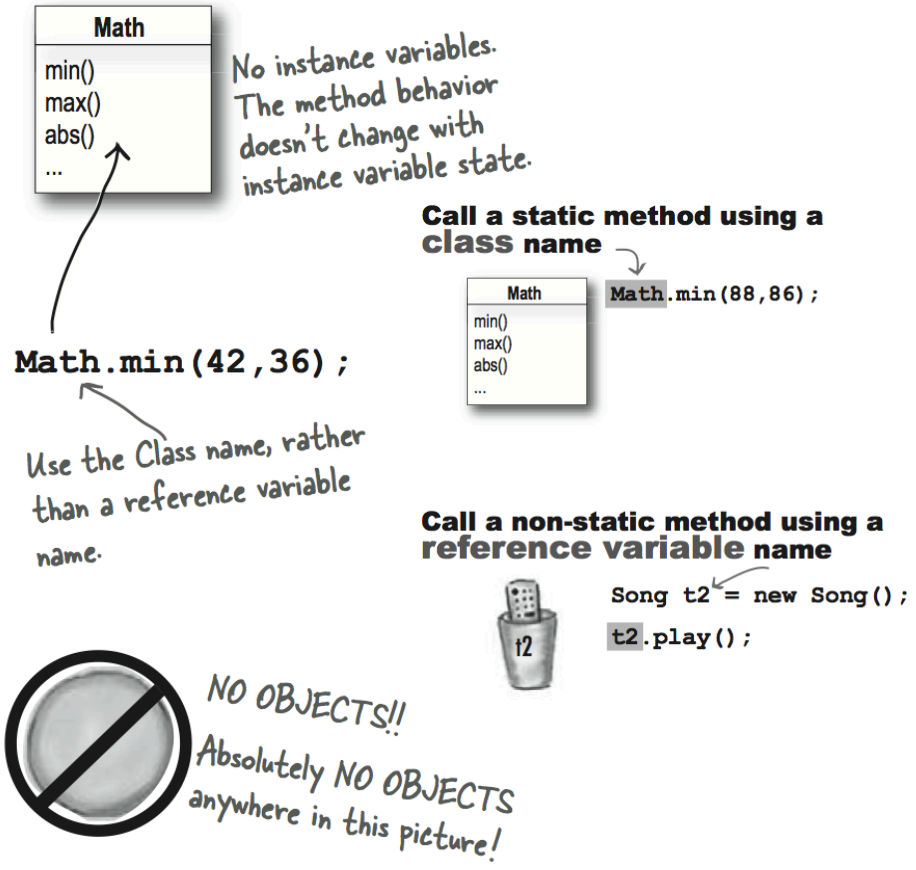
The current value of the 'title' instance variable is the song that plays when you call play().



static method

Static methods can't use non-static (instance) variables!

```
public static int min(int a, int b){
    //returns the lesser of a and b
}
```



Static imports

Some old-fashioned code:

```
import java.lang.Math;

class NoStatic {

  public static void main(String [] args) {

    System.out.println("sqrt " + Math.sqrt(2.0));

    System.out.println("tan " + Math.tan(60));

  }

}
```

Same code, with static imports:

```
import static java.lang.System.out;

import static java.lang.Math.*;

class WithStatic {

  public static void main(String [] args) {

    out.println("sqrt " + sqrt(2.0));

    out.println("tan " + tan(60));

  }

}
```

The syntax to use when declaring static imports.

Static imports in action.

Constants

static final variables

are constants

public static final double PI =

3.141592653589793;

Initialize a *final* static variable:

① At the time you declare it:

```
public class Foo {
   public static final int FOO_X = 25;
}
```

notice the naming convention -- static final variables are constants, so the name should be all uppercase, with an underscore separating the words

OR

② In a static initializer:

```
public class Bar {
   public static final double BAR_SIGN;

   static {
        BAR_SIGN = (double) Math.random();
   }
}
```

this code runs as soon as the class is loaded, before any static method is called and even before any static variable can be used.

Final modifier

non-static final variables

```
class Foof {
    final int size = 3;
    final int whuffie;

    Foof() {
        whuffie = 42;
    }

    void doStuff(final int x) {
        // you can't change x
    }

    void doMore() {
        final int z = 7;
        // you can't change z
    }
}
```

now you can't change size

now you can't change whuffie

final class

```
final class MyMostPerfectClass {
    // cannot be extended
}
```

final method

```
class Poof {
    final void calcWhuffie() {
        // important things
        // that must never be overridden
    }
}
```

Singleton pattern

```
class Shop {
   // configuration
   private static Shop instance = new Shop();

   private Shop() {}

   public void buyProduct(Product p);

   public static getInstance() {
        return instance;
   }
}
```

```
Shop shop = Shop.getInstance();
// allows to always get the same instance
```

Factory method pattern


```
class Shop {
  // configuration
  static String type= ”online”;

  private Shop() {}
  public void buyProduct(Product p) {}

  public static createShop() {
        if (type.equals(”online”)) {
          return new OnlineShop();
        } else {
          return new OfflineShop();
        }
  }
```

Every class is Object

1 equals(Object o)

```
Dog a = new Dog();
Cat c = new Cat();

if (a.equals(c)) {
    System.out.println("true");
} else {
    System.out.println("false");
}
```

```
File Edit Window Help Stop
% java TestObject
false
```

Tells you if two objects are considered 'equal' (we'll talk about what 'equal' really means in appendix B).

2 getClass()

```
Cat c = new Cat();
System.out.println(c.getClass());
```

```
File Edit Window Help Faint
% java TestObject
class Cat
```

Gives you back the class that object was instantiated from.

3 hashCode()

```
Cat c = new Cat();
System.out.println(c.hashCode());
```

```
File Edit Window Help Drop
% java TestObject
8202111
```

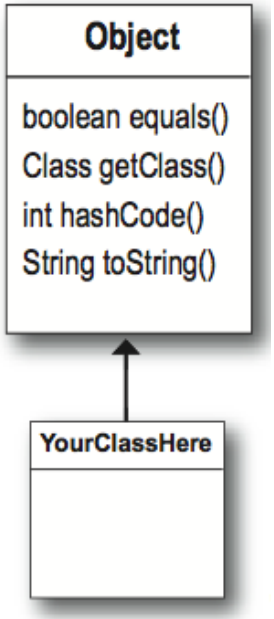
Prints out a hashcode for the object (for now, think of it as a unique ID).

4 toString()

```
Cat c = new Cat();
System.out.println(c.toString());
```

```
File Edit Window Help LapseIntoComa
% java TestObject
Cat@7d277f
```

Prints out a String message with the name of the class and some other number we rarely care about.



Wrapper classes

Wrapper classes

Boolean

Character

Byte

Short

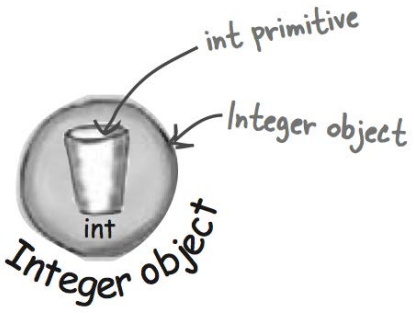
Integer

Long

Float

Double

Watch out! The names aren't mapped exactly to the primitive types. The class names are fully spelled out.



Give the primitive to the wrapper constructor. That's it.

wrapping a value

```
int i = 288;
Integer iWrap = new Integer(i);
```

All the wrappers work like this. Boolean has a booleanValue(), Character has a charValue(), etc.

***un*wrapping a value**

```
int unWrapped = iWrap.intValue();
```

Autoboxing/unboxing

```
Integer i = 3;

public void doNumsNewWay() {
        List<Integer> listOfNumbers = new ArrayList<Integer>();
        listOfNumbers.add(3);
        int num = listOfNumbers.get(0);
}
```

Primitive wrappers

- Java provides wrapper classes for each primitive data types.
- Wrappers are found in `java.lang.`

`Byte, Short, Integer, Long, Float, Double, Character`

- In new Java versions you may use wrappers absolutely transparently.

```
Integer count   = 1;
Boolean isReady = false;
```

Primitive wrappers

- Java automatically converts an object to a primitive type if it is required.

- The Boolean object will contain `true`, if the constructor parameter will be equal to "true" string in any case.

```
Boolean isReady = new Boolean("TRue"); //true

Boolean isReady = new Boolean("Yes"); //false
```

Primitive wrappers

- Each class has a set of constants with maximum and minimal values.

```
Integer.MIN_VALUE     Integer.MAX_VALUE
```

- Each class has static methods for converting type from string.

```
Double.parseDouble(String s)
```

- Numeric types are inherited from the `Number` class.

Primitive wrappers

- The Double class has the method of checking whether the number is infinite.

```
Double.isInfinity(double d)
```

- The Integer class has useful methods for work with binary representation of integers.

```
Integer.reverse(int i)

Integer.bitCount(int i)

Integer.numberOfLeadingZeros(int i)
```

Primitive wrappers

- To work with bigger numbers, classes from the `java.math` package can be used:

`BigInteger and BigDecimal`

- Numbers are stored as strings.

```
BigInteger number = new BigInteger("33");
BigInteger big = number.pow(10000);
```

Exercise

Lab guide:

- Exercise 9
- Exercise 10

Enumerations

Enumerations

- Very often we have to introduce enumerated types:

```
public class PlayingCard {
   public static final int SUIT_SPADES = 0;
   public static final int SUIT_HEARTS = 1;
   public static final int SUIT_CLUBS = 2;
   public static final int SUIT_DIAMONDS = 3;
   private int suit;
   public PlayingCard(int suit) {
      this.suit = suit;
   }

   public String getSuitName() {
      String name = "";
      switch (suit) {
      case SUIT_SPADES:
         name = "Spades";
         break;
      case SUIT_HEARTS:
         name = "Hearts";
         break;
      ...
```

Enumerations

- Java 1.5 introduced new enumeration mechanism (enum).

- Enumeration is subclass of the `java.lang.Enum` class. Enumeration solves this problem and can be applied to the `switch` operator.

- Enumeration is a usual class with some limitations.

Difference between enumerations and classes

- Declared with the help of **`enum`**.
- Enumeration instance cannot be explicitly created.
- Enumeration cannot be extended.
- Enumeration can be the **`switch`** argument.
- Has embedded `name()` method that prints enumeration values.

Enumerations

```
public enum LightState {
   RED, YELLOW, GREEN;
}

public static void main(String[] args) {
   switch (nextTrafficLight.getState()) {
   case LightState.RED:
      stop();
      break;
   case LightState.YELLOW:
      floorIt();
      break;
   ...
}
```

Enumerations

```
enum Suit {
   DIAMOND(true), HEART(true), CLUB(false),
SPADE(false);

   private boolean red;

   Suit(boolean b) {
      red = b;
   }
   public boolean isRed() {
      return red;
   }
   public String toString() {
      String s = name();
      s += red ? ":red" : ":black";
      return s;
   }
}
```

Exercise

Lab guide:

- Exercise 11
- Exercise 12