Inheritance (Part 4)

Abstract Classes

Abstract Classes

- sometimes you will find that you want the API for a base class to have a method that the base class cannot define
  - e.g. you might want to know what a **`Dog`**'s bark sounds like but the sound of the bark depends on the breed of the dog
    - you want to add the method bark to **`Dog`** but only the subclasses of **`Dog`** can implement **`bark`**

Abstract Classes

- sometimes you will find that you want the API for a base class to have a method that the base class cannot define
  - e.g. you might want to know the breed of a **`Dog`** but only the subclasses have information about the breed
    - you want to add the method **`getBreed`** to **`Dog`** but only the subclasses of **`Dog`** can implement **`getBreed`**

- if the base class has methods that only subclasses can define *and* the base class has fields common to all subclasses then the base class should be abstract
  - if you have a base class that just has methods that it cannot implement then you probably want an interface
- abstract :
  - (dictionary definition) existing only in the mind

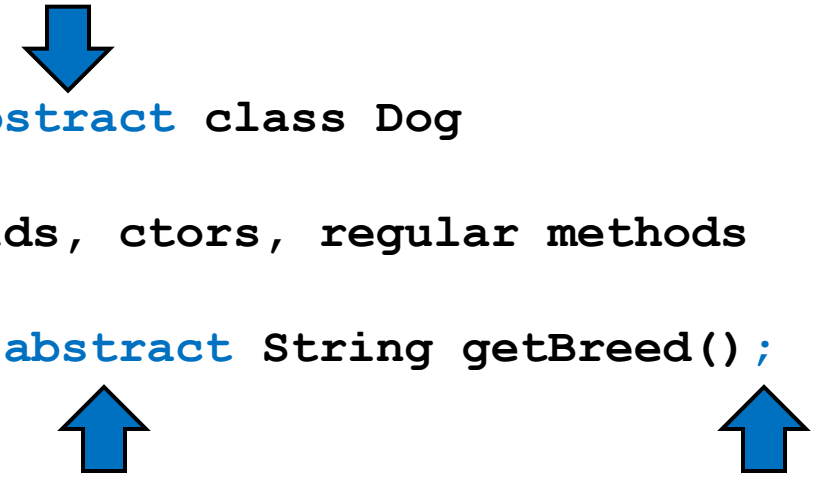
- in Java an abstract class is a class that you cannot make instances of
  - e.g. http://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html

- an abstract class provides a partial definition of a class
  - the "partial definition" contains everything that is common to all of the subclasses
  - the subclasses complete the definition

- an abstract class can define fields and methods
  - subclasses *inherit* these
- an abstract class can define constructors
  - subclasses *must call* these
- an abstract class can declare abstract methods
  - subclasses *must define* these (unless the subclass is also abstract)

Abstract Methods

- an abstract base class can declare, *but not define*, zero or more abstract methods



```
public abstract class Dog
{
  // fields, ctors, regular methods

  public abstract String getBreed();
}
```

- the base class is saying "all **`Dog`**s can provide a **`String`** describing the breed, but only the subclasses know enough to implement the method"

Abstract Methods

- the non-abstract subclasses must provide definitions for all abstract methods
  - consider **`getBreed`** in **`Mix`**

```
public class Mix extends Dog
{ // stuff from before...

  @Override
  public String getBreed() {
    if(this.breeds.isEmpty()) {
      return "mix of unknown breeds";
    }
    StringBuffer b = new StringBuffer();
    b.append("mix of");
    for(String breed : this.breeds) {
      b.append(" " + breed);
    }
    return b.toString();
}
```

PureBreed

- a purebreed dog is a dog with a single breed
  - one **`String`** field to store the breed
- note that the breed is determined by the subclasses
  - the class **`PureBreed`** cannot give the **`breed`** field a value
  - but it can implement the method **`getBreed`**
- the class **`PureBreed`** defines an field common to all subclasses and it needs the subclass to inform it of the actual breed
  - **`PureBreed`** is also an abstract class

```
public abstract class PureBreed extends Dog
{
  private String breed;

  public PureBreed(String breed) {
    super();
    this.breed = breed;
  }

  public PureBreed(String breed, int size, int energy) {
    super(size, energy);
    this.breed = breed;
  }
```

```
  @Override public String getBreed()
  {
    return this.breed;
  }

}
```

Subclasses of PureBreed

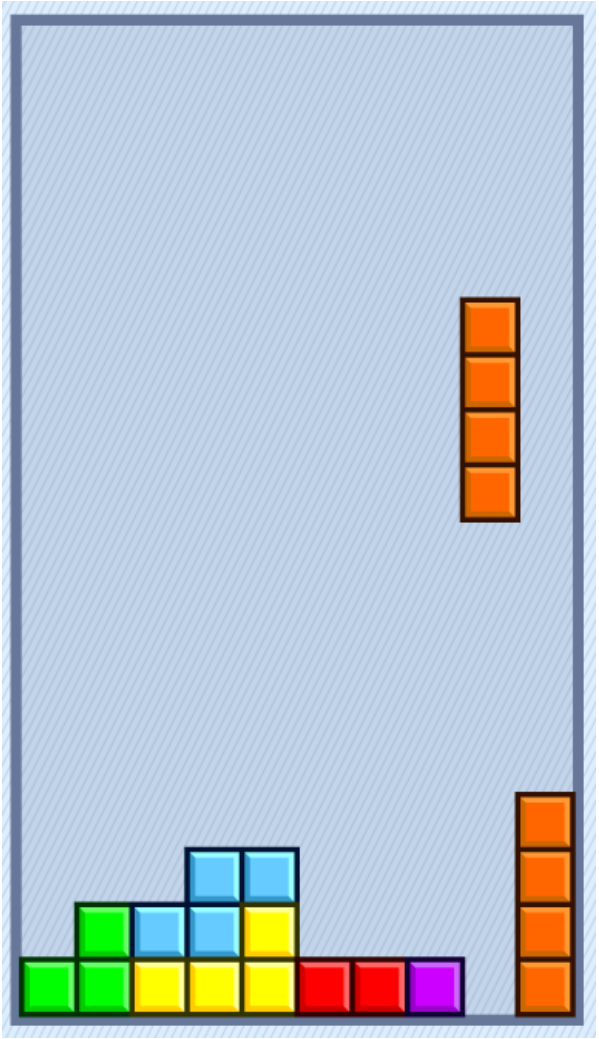
- the subclasses of **`PureBreed`** are responsible for setting the breed
  - consider **`Komondor`**

Komondor

```
public class Komondor extends PureBreed
{
  private final String BREED = "komondor";

  public Komondor() {
    super(BREED);
  }

  public Komondor(int size, int energy) {
    super(BREED, size, energy);
  }

  // other Komondor methods...
}
```

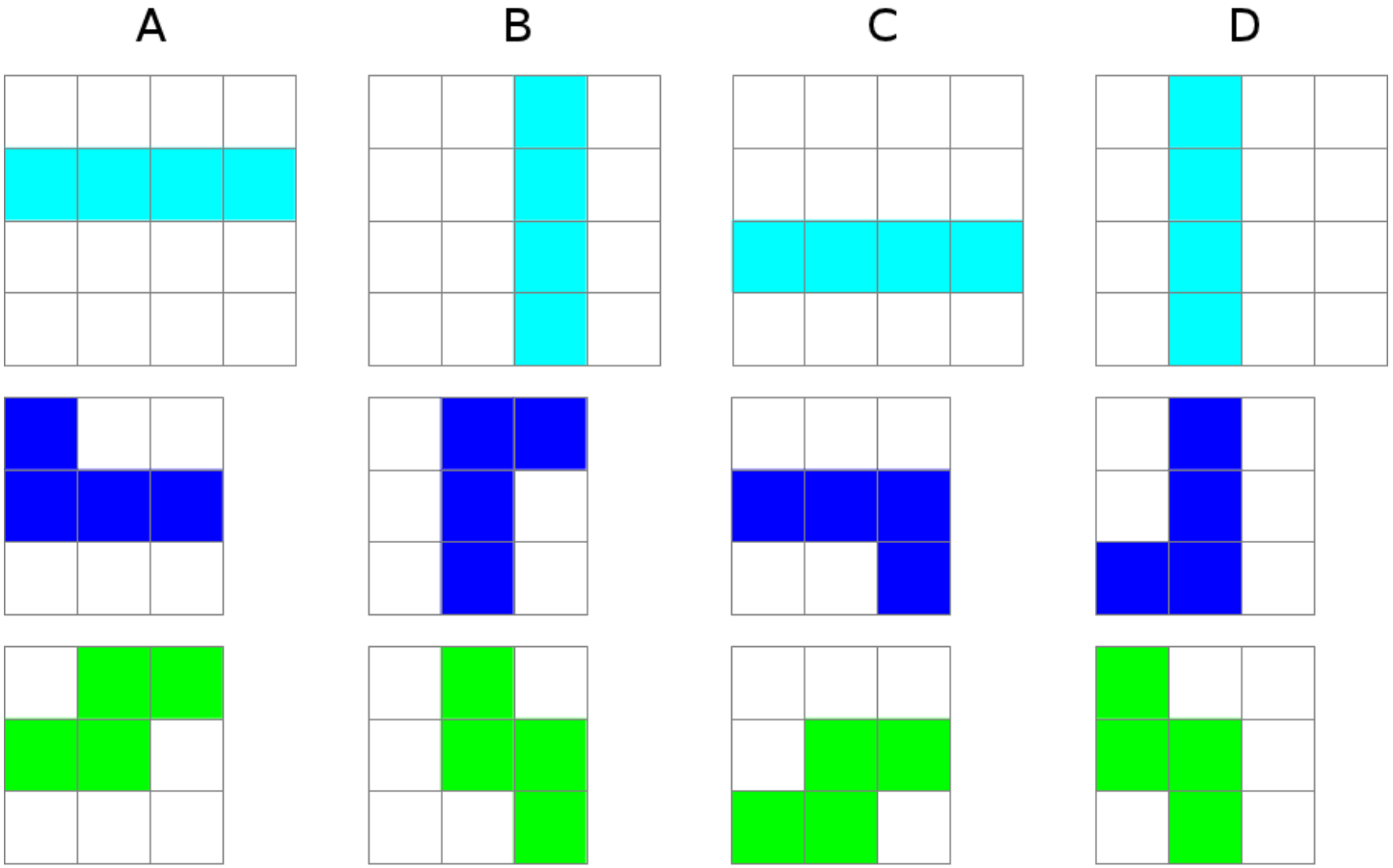
Another example: Tetris

- played with 7 standard blocks called tetriminoes
- blocks drop from the top
- player can move blocks left, right, and down
- player can spin blocks left and right



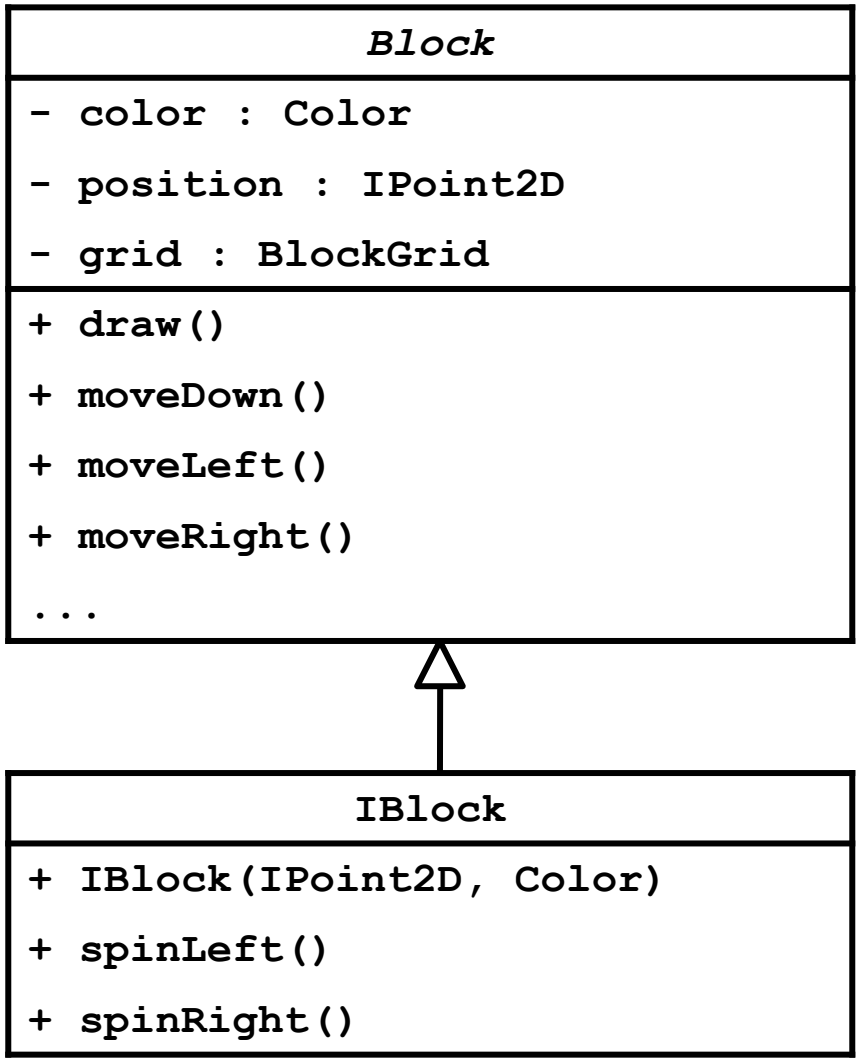
Tetriminoes

- spinning the I, J, and S blocks



Tetriminoes

- features common to all tetriminoes
  - has-a color
  - has-a shape
  - has-a position
  - draw
  - move left, right, and down
- features unique to each kind of tetrimino
  - the actual shape
  - spin left and right



- class name in italics for abstract classes
- an immutable 2D point
- a grid object that stores the shape
- constructor defines the shape
- methods modify the shape to produce the rotated version of the block

http://www.eecs.yorku.ca/course_archive/2013-14/F/1030/labs/06/lab6.html

Inheritance (Part 5)

Static Features; Interfaces

Static Fields and Inheritance

- static fields behave the same as non-static fields in inheritance
  - public and protected static fields are inherited by subclasses, and subclasses can access them directly by name
  - private static fields are not inherited and cannot be accessed directly by name
    - but they can be accessed/modified using public and protected methods

Static Fields and Inheritance

- the important thing to remember about static fields and inheritance
  - *there is only one copy of the static field shared among the declaring class and all subclasses*
- consider trying to count the number of **`Dog`** objects created by using a static counter

```
// the wrong way to count the number of Dogs created
public abstract class Dog {
  // other fields...
  static protected int numCreated = 0;

  Dog() {
    // ...
    Dog.numCreated++;
  }

  public static int getNumberCreated() {
    return Dog.numCreated;
  }

  // other contructors, methods...
}
```

protected, not private, so that subclasses can modify it directly

```
// the wrong way to count the number of Dogs created
public class Mix extends Dog
{
  // fields...

  Mix()
  {
    super();
    Mix.numCreated++;
  }

  // other contructors, methods...
}
```

```
// too many dogs!

public class TooManyDogs
{
  public static void main(String[] args)
  {
    Mix mutt = new Mix();
    System.out.println( Mix.getNumberCreated() );
  }
}
```

prints 2

What Went Wrong?

- there is only one copy of the static field shared among the declaring class and all subclasses
  - **`Dog`** declared the static field
  - **`Dog`** increments the counter everytime its constructor is called
  - **`Mix`** inherits *and shares* the single copy of the field
  - **`Mix`** constructor correctly calls the superclass constructor
    - which causes **`numCreated`** to be incremented by **`Dog`**
  - **`Mix`** constructor then incorrectly increments the counter

Counting Dogs and Mixes

- suppose you want to count the number of **`Dog`** instances and the number of **`Mix`** instances
  - **`Mix`** must also declare a static field to hold the count
    - somewhat confusingly, **`Mix`** can give the counter the same name as the counter declared by **`Dog`**

```
public class Mix extends Dog
{
  // other fields...
  private static int numCreated = 0;  // bad style

  public Mix()
  {
    super();       // will increment Dog.numCreated
    // other Mix stuff...
    numCreated++; // will increment Mix.numCreated
  }

  // ...
```

Hiding Fields

- note that the **`Mix`** field **`numCreated`** has the same name as an field declared in a superclass
  - whenever **`numCreated`** is used in **`Mix`**, it is the **`Mix`** version of the field that is used

- if a subclass declares an field with the same name as a superclass field, we say that the subclass field hides the superclass field
  - considered bad style because it can make code hard to read and understand
    - should change **`numCreated`** to **`numMixCreated`** in **`Mix`**

Static Methods and Inheritance

- there is a significant difference between calling a static method and calling a non-static method when dealing with inheritance

- *there is no dynamic dispatch on static methods*
  - therefore, you cannot override a static method

```
public abstract class Dog {
  private static int numCreated = 0;
  public static int getNumCreated() {
    return Dog.numCreated;
  }
}

public class Mix {
  private static int numMixCreated = 0;
  public static int getNumCreated() {
    return Mix.numMixCreated;
  }
}

public class Komondor {
  private static int numKomondorCreated = 0;
  public static int getNumCreated() {
    return Komondor.numKomondorCreated;
  }
}
```

notice no @Override

notice no @Override

```
public class WrongCount {
  public static void main(String[] args) {
    Dog mutt = new Mix();
    Dog shaggy = new Komondor();
    System.out.println( mutt.getNumCreated() );
    System.out.println( shaggy.getNumCreated() );
    System.out.println( Mix.getNumCreated() );
    System.out.println( Komondor.getNumCreated() );
  }
}
```

prints 2
2
1
1

What's Going On?

- *there is no dynamic dispatch on static methods*
- because the declared type of **`mutt`** is **`Dog`**, it is the **`Dog`** version of **`getNumCreated`** that is called
- because the declared type of **`shaggy`** is **`Dog`**, it is the **`Dog`** version of **`getNumCreated`** that is called

Hiding Methods

- notice that **`Mix.getNumCreated`** and **`Komondor.getNumCreated`** work as expected
- if a subclass declares a static method with the same name as a superclass static method, we say that the subclass static method hides the superclass static method
  - *you cannot override a static method, you can only hide it*
  - hiding static methods is considered bad form because it makes code hard to read and understand

- the client code in **`WrongCount`** illustrates two cases of bad style, one by the client and one by the implementer of the **`Dog`** hierarchy
  1. the client should not have used an instance to call a static method
  2. the implementer should not have hidden the static method in **`Dog`**