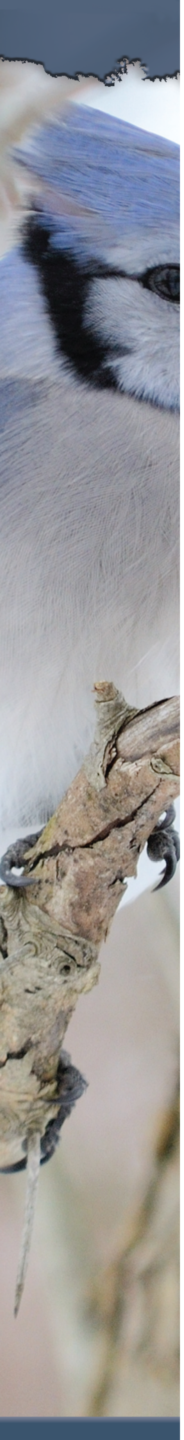
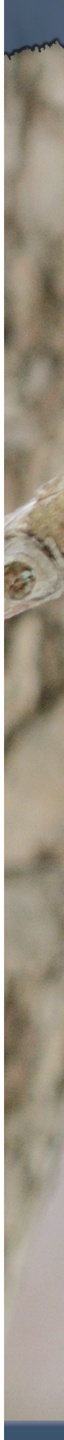


Further abstraction techniques

Abstract classes and interfaces



Main concepts to be covered

- Abstract classes
- Interfaces
- Multiple inheritance

Simulations

- Programs regularly used to simulate real-world activities:
  - city traffic;
  - the weather;
  - nuclear processes;
  - stock market fluctuations;
  - environmental impacts;
  - space flight.

Simulations

- They are often only partial simulations.
- They often involve simplifications.
  - Greater detail has the potential to provide greater accuracy.
  - Greater detail typically requires more resource:
    - Processing power;
    - Simulation time.

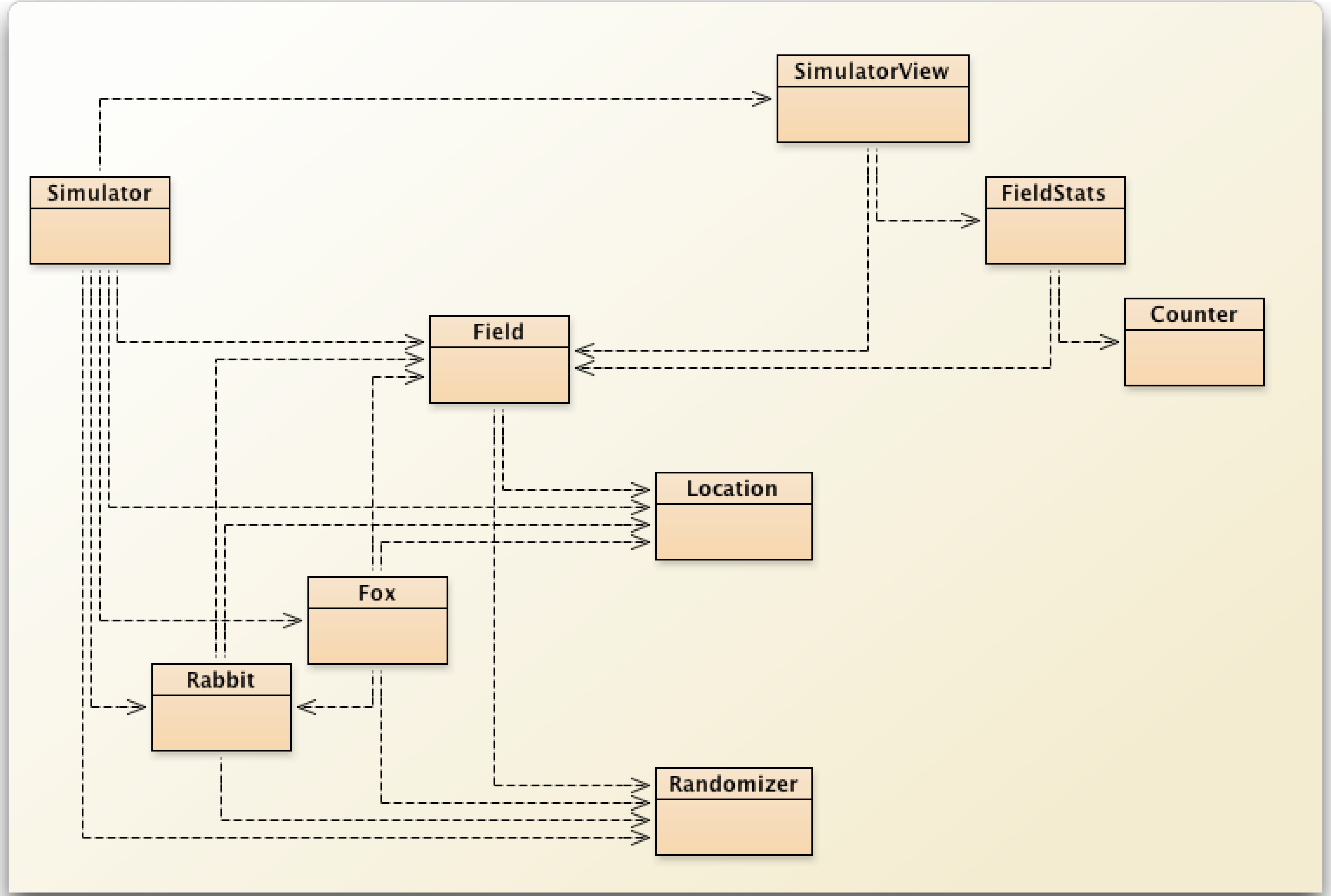
Benefits of simulations

- Support useful prediction.
  - E.g., the weather.
- Allow experimentation.
  - Safer, cheaper, quicker.
- Our example:
  - ‘How will the wildlife be affected if we cut a highway through the middle of this national park?’

Predator-prey simulations

- There is often a delicate balance between species.
  - A lot of prey means a lot of food.
  - A lot of food encourages higher predator numbers.
  - More predators eat more prey.
  - Less prey means less food.
  - Less food means …

The foxes-and-rabbits project



Main classes of interest

- **`Fox`**
  - Simple model of a type of predator.
- **`Rabbit`**
  - Simple model of a type of prey.
- **`Simulator`**
  - Manages the overall simulation task.
  - Holds a collection of foxes and rabbits.

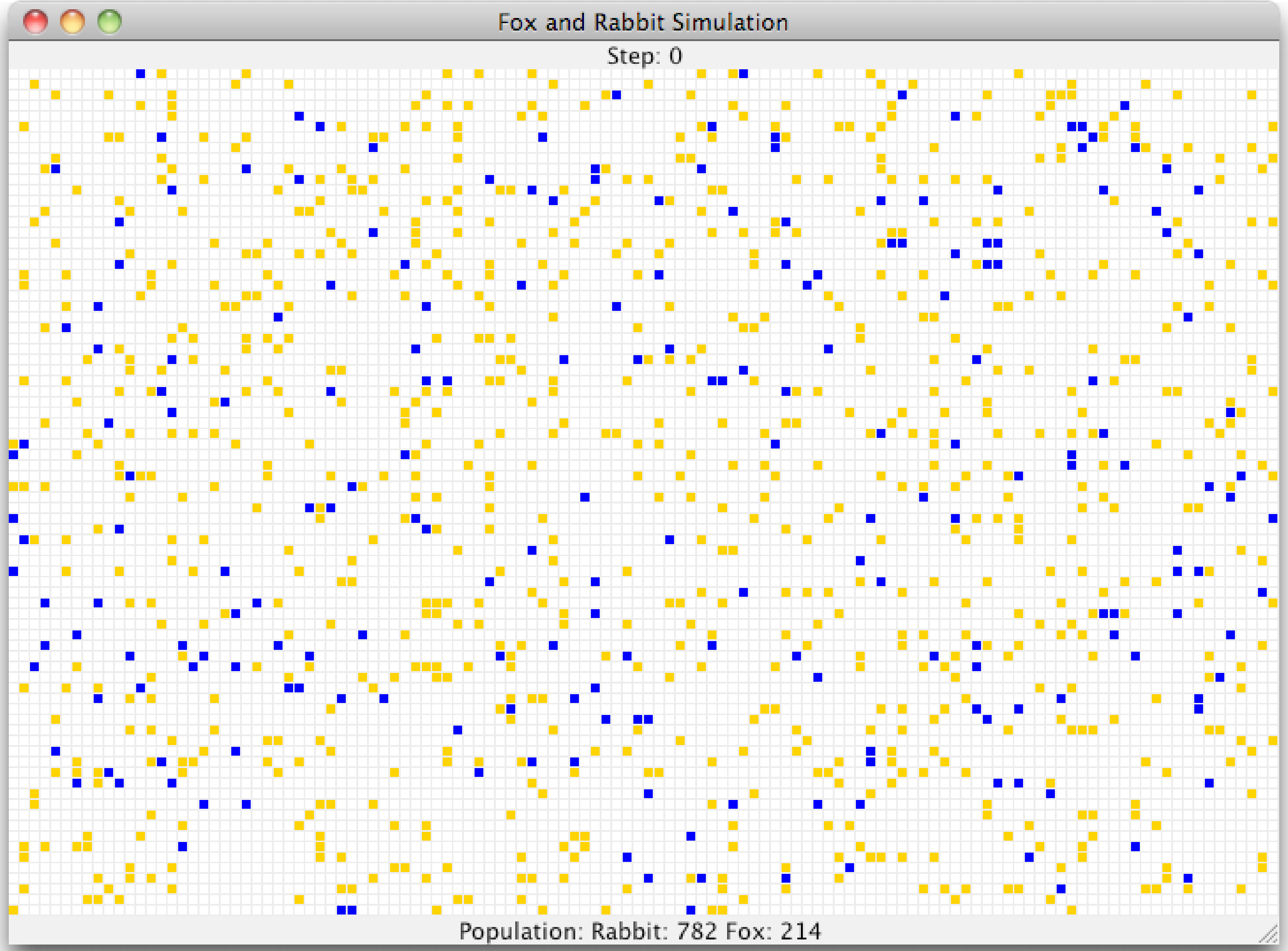
Modeling the environment

- **`Field`**
  - Represents a 2D field.
- **`Location`**
  - Represents a 2D position in the environment.

Monitoring the simulation

- `SimulatorView`
  - Presents a view of the environment.
- `FieldStats`, `Counter`
  - Maintain statistics.
- `Randomizer`
  - Supports reproducibility.

Example of the visualization



A Rabbit's state

```
class Rabbit {
    Static fields omitted.

    // Individual characteristics (instance fields).

    // The rabbit's age.
    private int age;
    // Whether the rabbit is alive or not.
    private boolean alive;
    // The rabbit's position
    private Location location;
    // The field occupied
    private Field field;

    Methods omitted.
}
```

A Rabbit's behavior

- Managed from the `run` method.
- Age incremented at each simulation 'step'.
  - A rabbit could die at this point.
- Rabbits that are old enough might breed at each step.
  - New rabbits could be born at this point.

Rabbit simplifications

- Rabbits do not have different genders.
  - In effect, all are female.
- The same rabbit could breed at every step.
- All rabbits die at the same age.
- Others?

A Fox's state

```
class Fox {
    Static fields omitted

    // The fox's age.
    private int age;
    // Whether the fox is alive or not.
    private boolean alive;
    // The fox's position
    private Location location;
    // The field occupied
    private Field field;
    // The fox's food level, which is increased
    // by eating rabbits.
    private int foodLevel;

    Methods omitted.
}
```

A Fox's behavior

- Managed from the `hunt` method.
- Foxes also age and breed.
- They become hungry.
- They hunt for food in adjacent locations.

Configuration of foxes

- Similar simplifications to rabbits.
- Hunting and eating could be modeled in many different ways.
  - Should food level be additive?
  - Is a hungry fox more or less likely to hunt?
- Are simplifications ever acceptable?

The Simulator class

- Three key components:
  - Setup in the constructor.
  - The `populate` method.
    - Each animal is given a random starting age.
  - The `simulateOneStep` method.
    - Iterates over separate populations of foxes and rabbits.
    - Two `Field` objects are used: `field` and `updatedField`.

The update step

```
for (Iterator<Rabbit> it = rabbits.iterator();
        it.hasNext(); ) {
    Rabbit rabbit = it.next();
    rabbit.run(newRabbits);
    if (! rabbit.isAlive()) {
        it.remove();
    }
}
…
for (Iterator<Fox> it = foxes.iterator();
        it.hasNext(); ) {
    Fox fox = it.next();
    fox.hunt(newFoxes);
    if (! fox.isAlive()) {
        it.remove();
    }
}
```

Room for improvement

- `Fox` and `Rabbit` have strong similarities but do not have a common superclass.
- The update step involves similar-looking code.
- The `Simulator` is tightly coupled to specific classes.
  - It ‘knows’ a lot about the behavior of foxes and rabbits.

The Animal superclass

- Place common fields in **`Animal`**:
  - **`age`**, **`alive`**, **`location`**
- Method renaming to support information hiding:
  - **`run`** and **`hunt`** become **`act`**.
- **`Simulator`** can now be significantly decoupled.

Revised (decoupled) iteration

```
for (Iterator<Animal> it = animals.iterator();
        it.hasNext(); ) {
    Animal animal = iter.next();
    animal.act(newAnimals);
    // Remove dead animals from simulation
    if (! animal.isAlive()) {
        it.remove();
    }
}
```

The **`act`** method of Animal

- Static type checking requires an **`act`** method in **`Animal`**.
- There is no obvious shared implementation.
- Define **`act`** as abstract:

```
abstract void act(List<Animal> newAnimals);
```

The **`act`** method of Animal

- Static type checking requires an **`act`** method in **`Animal`**.
- There is no obvious shared implementation.
- Define **`act`** as abstract:

```
abstract void act(List<Animal> newAnimals);
```

What? No code?

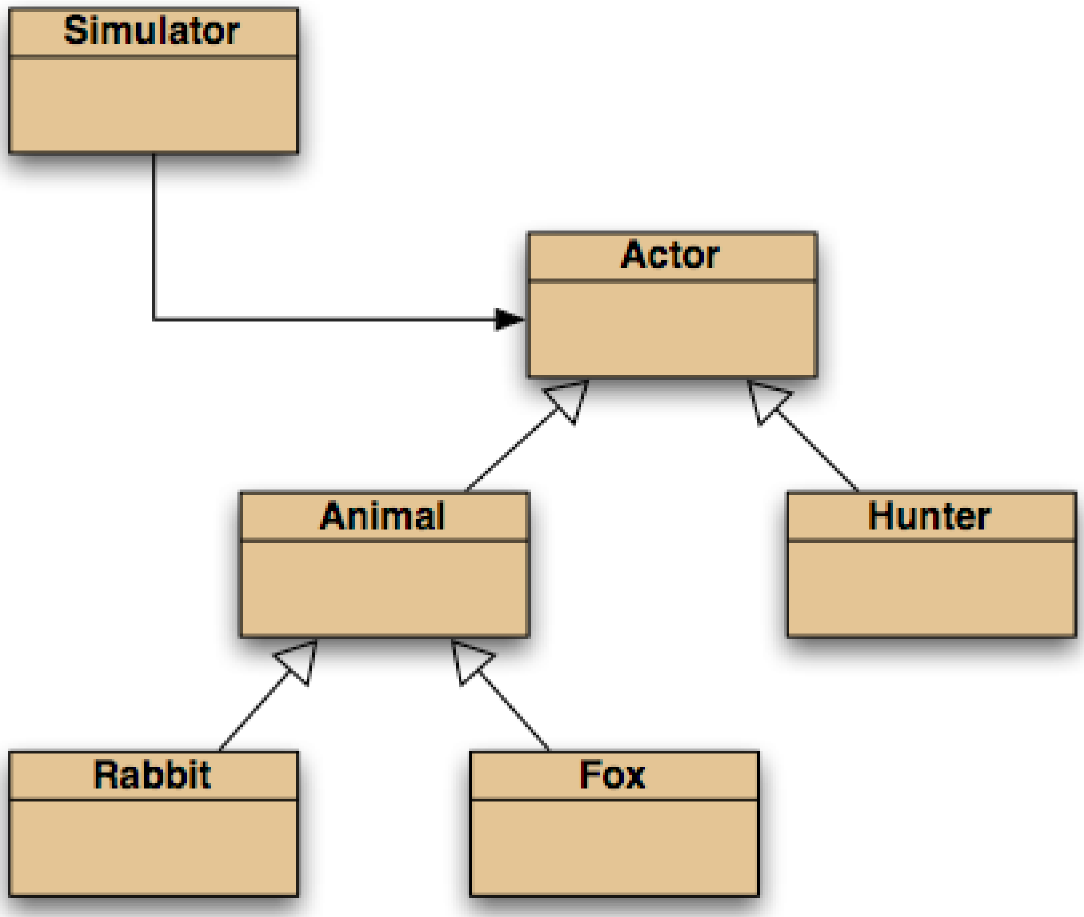
Abstract classes and methods

- Abstract methods have `abstract` in the signature.
- Abstract methods have no body.
- Abstract methods make the class abstract.
- *Abstract classes cannot be instantiated.*
- Concrete subclasses complete the implementation.

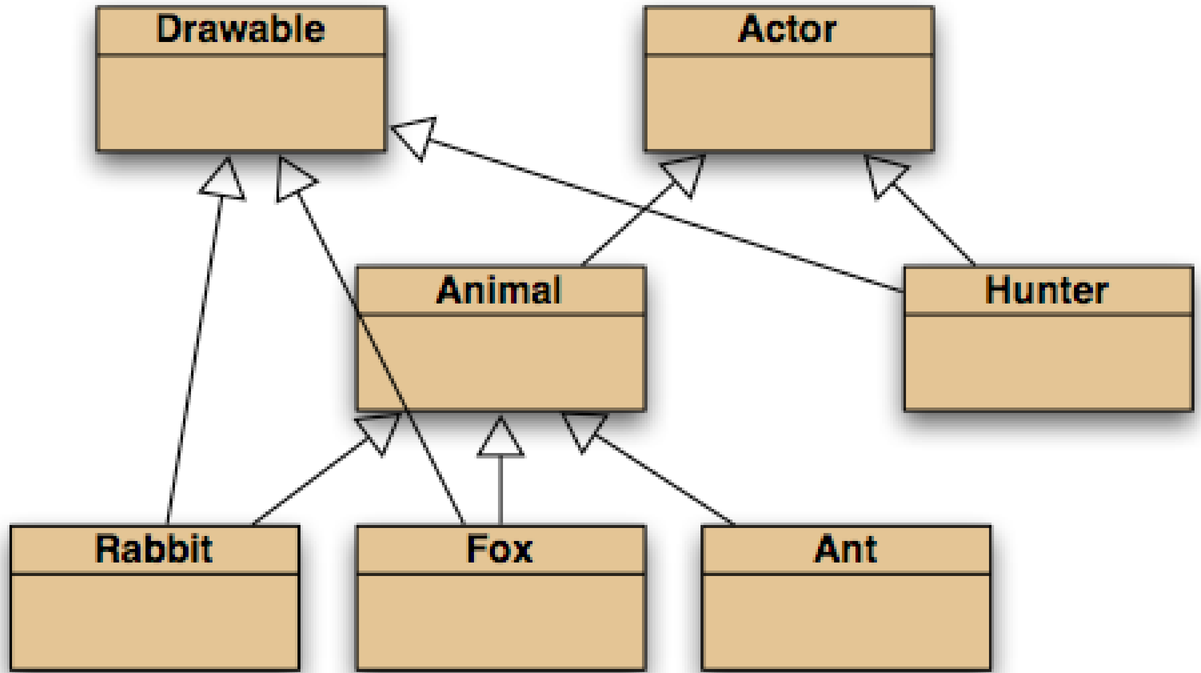
The Animal class

```
abstract class Animal {
    fields omitted

    /**
     * Make this animal act - that is: make it do
     * whatever it wants/needs to do.
     */
    abstract void act(List<Animal> newAnimals);

    other methods omitted
}
```

Further abstraction



Selective drawing (multiple inheritance)



Multiple inheritance

- Having a class inherit directly from multiple ancestors.
- Each language has its own rules.
  - How to resolve competing definitions?
- Java forbids it for classes.
- Java permits it for interfaces.

Multiple inheritance

- Having a class inherit directly from multiple ancestors.
- Each language has its own rules.
  - How to resolve competing definitions?
- Java forbids it for classes.
- Java permits it for interfaces.

For *what*??

An Actor interface

```
interface Actor {
    /**
     * Perform the actor's regular behavior.
     * @param newActors A list for storing newly created
     *                  actors.
     */
    void act(List<Actor> newActors);

    /**
     * Is the actor still active?
     * @return true if still active, false if not.
     */
    boolean isActive();
}
```

Classes *implement* an interface

```
class Fox extends Animal
    implements Drawable {
   ...
}


class Hunter
    implements Actor, Drawable {
   ...
}
```

Interfaces as types

- Implementing classes are subtypes of the interface type.
- So, polymorphism is available with interfaces as well as classes.

```
List<Toto> l = new ArrayList<Toto>();
l = new LinkedList<Toto>();
```

Features of interfaces

- Use **`interface`** in their declaration.
- They do not define constructors.
- All fields are **`public`**, **`static`** and **`final`**. (Keywords may be omitted.)
- Methods can be declared:
  - **`abstract public`**
  - **`default public`**
  - **`static public`**
  - **`private`**
  - **`static private`**

Abstract methods

- All abstract methods are **`public`**. (Keywords may be omitted.)
- Abstract methods have no implementation.
- Meant to be overridden in implementing classes.
- Describe behaviour without committing to implementation.

Abstract methods

- All abstract methods are **`public`**. (Keywords may be omitted.)
- Abstract methods have no implementation.
- Meant to be overridden in implementing classes.
- Describe behaviour without committing to implementation.

Most important interface methods!

Default methods

- Methods marked as **`default`** have a method body – they are not abstract.
- Don't have to be overridden in implementing class.
- Default and static methods can complicate multiple inheritance of interfaces.

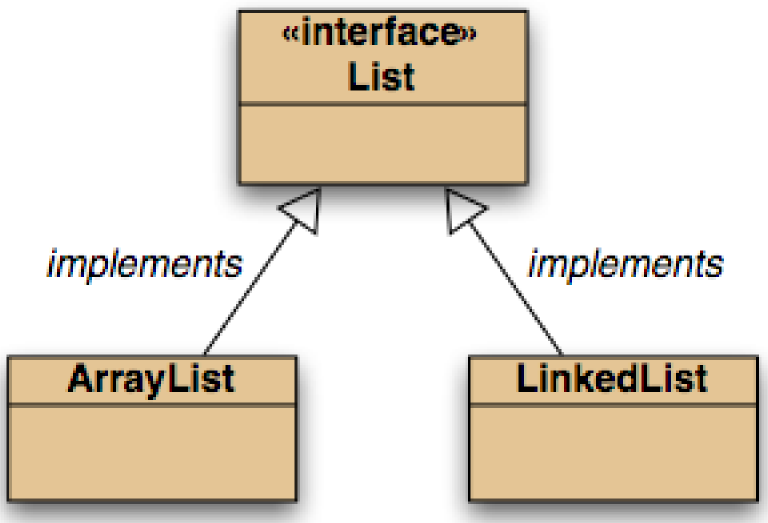
Default methods

- To adapt legacy interfaces; e.g., **`java.util.List`**.
- Allow adding new methods to interfaces without breaking existing code.
- Classes inheriting from two with the same signature must override the method.
- Syntax for accessing the overriden version: ***InterfaceType*.super.*method*(…)**

Interfaces as specifications

- Strong separation of functionality from implementation.
  - Though parameter and return types are mandated.
- Clients interact independently of the implementation.
  - But clients can choose from alternative implementations.
- **`List`**, **`Map`** and **`Set`** are examples.

Alternative implementations



Functional interfaces and lambdas (advanced)

- Interfaces with a single abstract method are *functional interfaces*.
- **`@FunctionalInterface`** is the associated annotation.
- A lambda may be used where a functional interface is required.
- **`java.util.function`** defines some functional interfaces.

Common functional interfaces (advanced)

- **`Consumer`**: for lambdas with a **`void`** return type.

```
void sumer(String s, Consumer<String> c) {
    c.accept(s);
}

sumer("foo", e -> System.out.println(e));
sumer("foo",
    e -> System.out.print(e.toUpperCase()));
```

Common functional interfaces (advanced)

- **`BinaryOperator`**: for lambdas with two parameters and a matching result type.

```
double boper(double a, double b,
    BinaryOperator<Double> bop) {
    return bop.apply(a, b);
}

boper(42, 1.0, (a, b) -> 2 * (a + b));
boper(42, 1.0, (a, b) -> a - b);
```

Common functional interfaces (advanced)

- **`Supplier`**: for lambdas returning a result.

```
int supp(Supplier<Integer> s) {
    return s.get();
}

supp(() -> new Random().nextInt(10));
supp(
    () -> (int) Math.abs(new Random().nextInt()));
```

Common functional interfaces (advanced)

- **`Predicate`**: returns a **`boolean`**.

```
boolean boo(String s, Predicate<String> p) {
    return p.test(s);
}

boo("foo", s -> s.trim().length() > 2);
```

Functional interfaces (advanced)

- Functional types can be assigned to variables, eg,:

```
BinaryOperator<String> akaOp =
    (name, alias) -> name + " (AKA " + alias + ")";

String aka(String s, String t,
        BinaryOperator<String> bo) {
   return bo.apply(s, t);
}

aka("foo", "bar", akaOp);
```

Functional interfaces (advanced)

- Functional types can be assigned to variables, eg,:

```
BinaryOperator<String> akaOp =
     (name, alias) -> name + " (AKA " + alias + ")";

String aka(String s, String t,
         BinaryOperator<String> bo) {
   return bo.apply(s, t);
}

aka("foo", "bar", akaOp);
```

```
foo (AKA bar)
```

The `Class` class

- A `Class` object is returned by `getClass()` in `Object`.
- The `.class` suffix provides a `Class` object:
  `Fox.class`
- Used in `SimulatorView`:
  `Map<Class, Color> colors;`
- `String getName()` for the class name.

Review

- Inheritance can provide shared implementation.
  - Concrete and abstract classes.
- Inheritance provides shared type information.
  - Classes and interfaces.

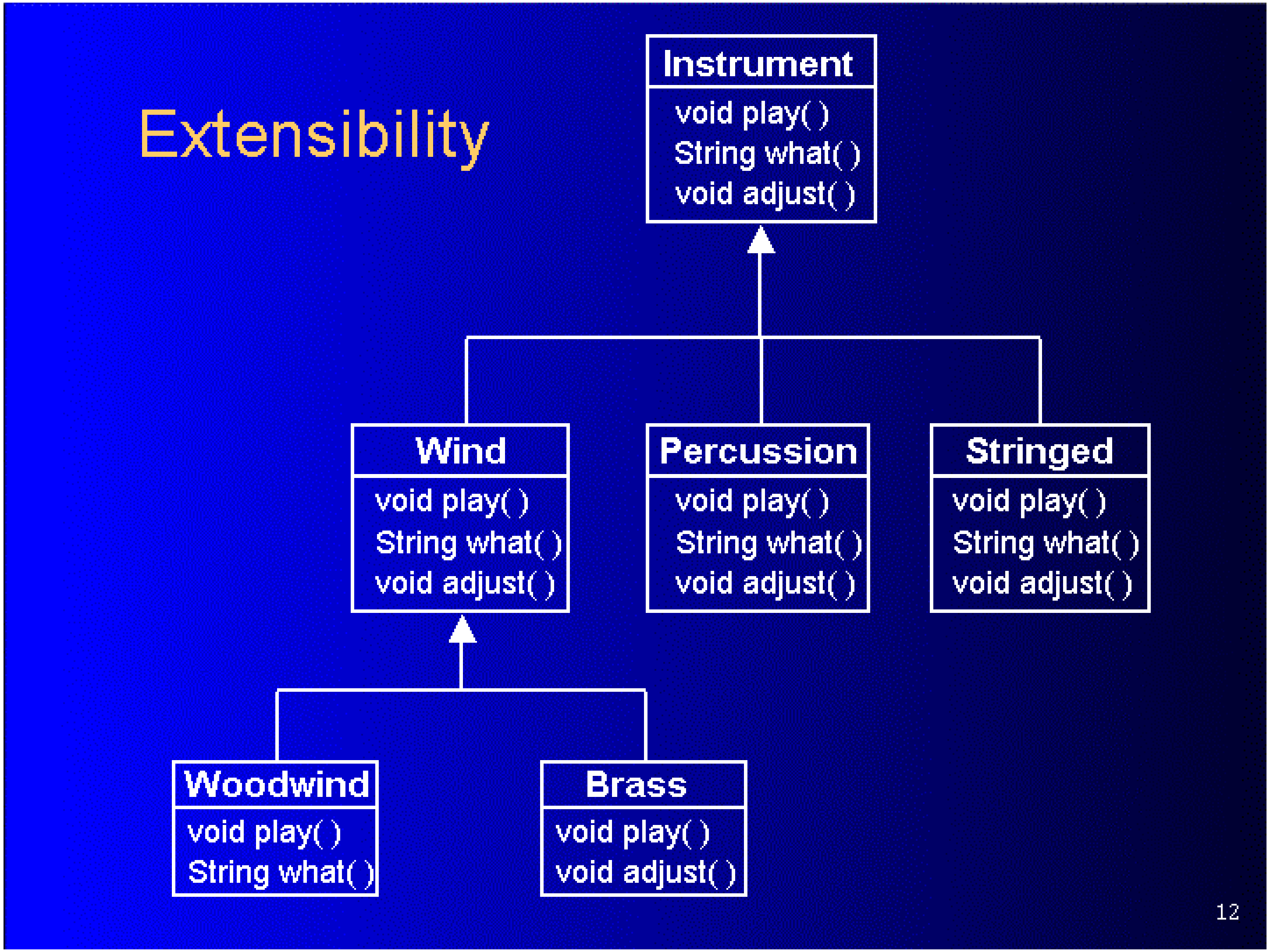
Review

- Abstract methods allow static type checking without requiring implementation.
- Abstract classes function as incomplete superclasses.
  - No instances.
- Abstract classes support polymorphism.

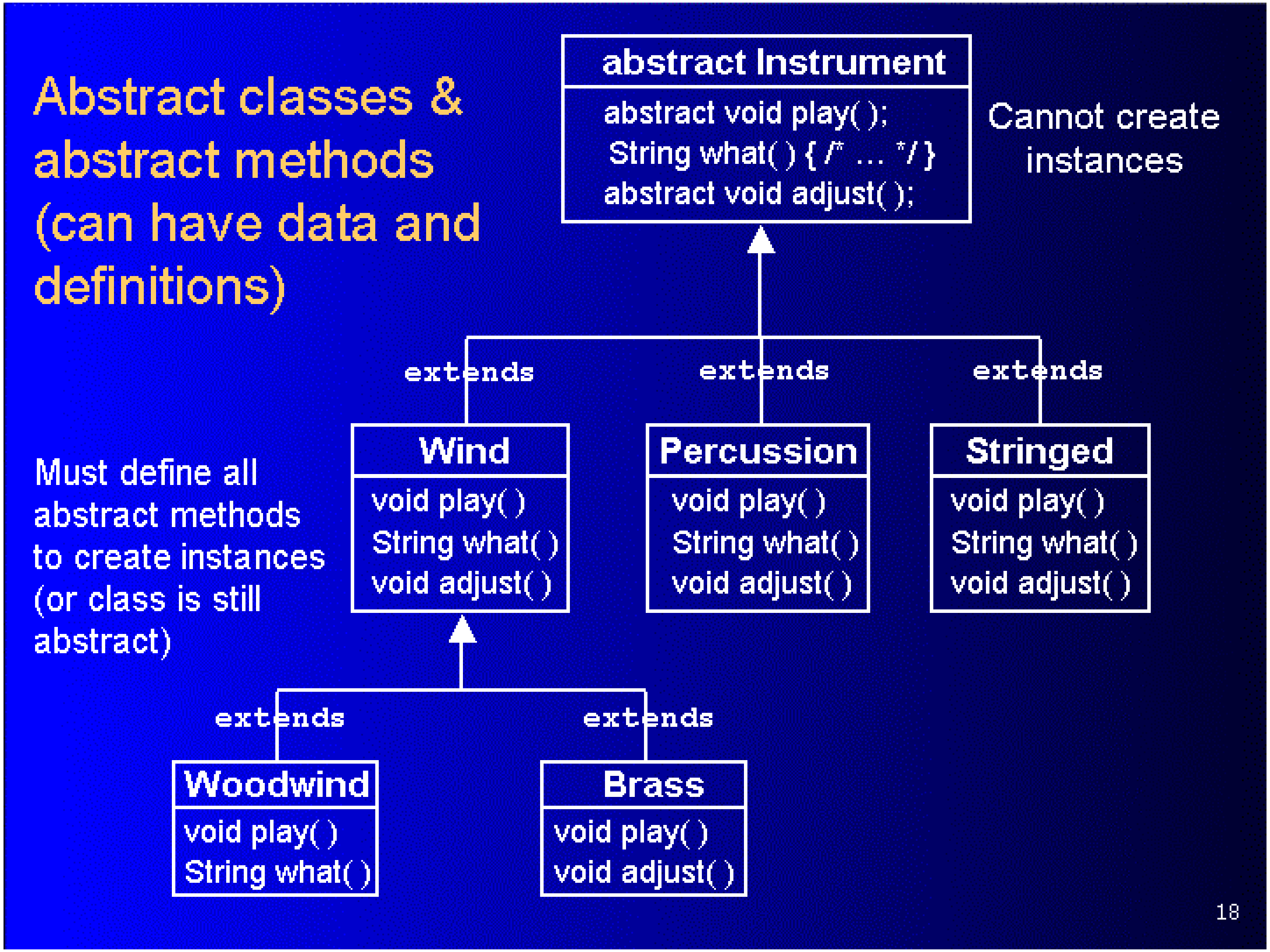
Review

- Interfaces provide specification – usually without implementation.
  - Interfaces are abstract apart from their default and private methods.
- Interfaces support polymorphism.
- Java interfaces support multiple inheritance.

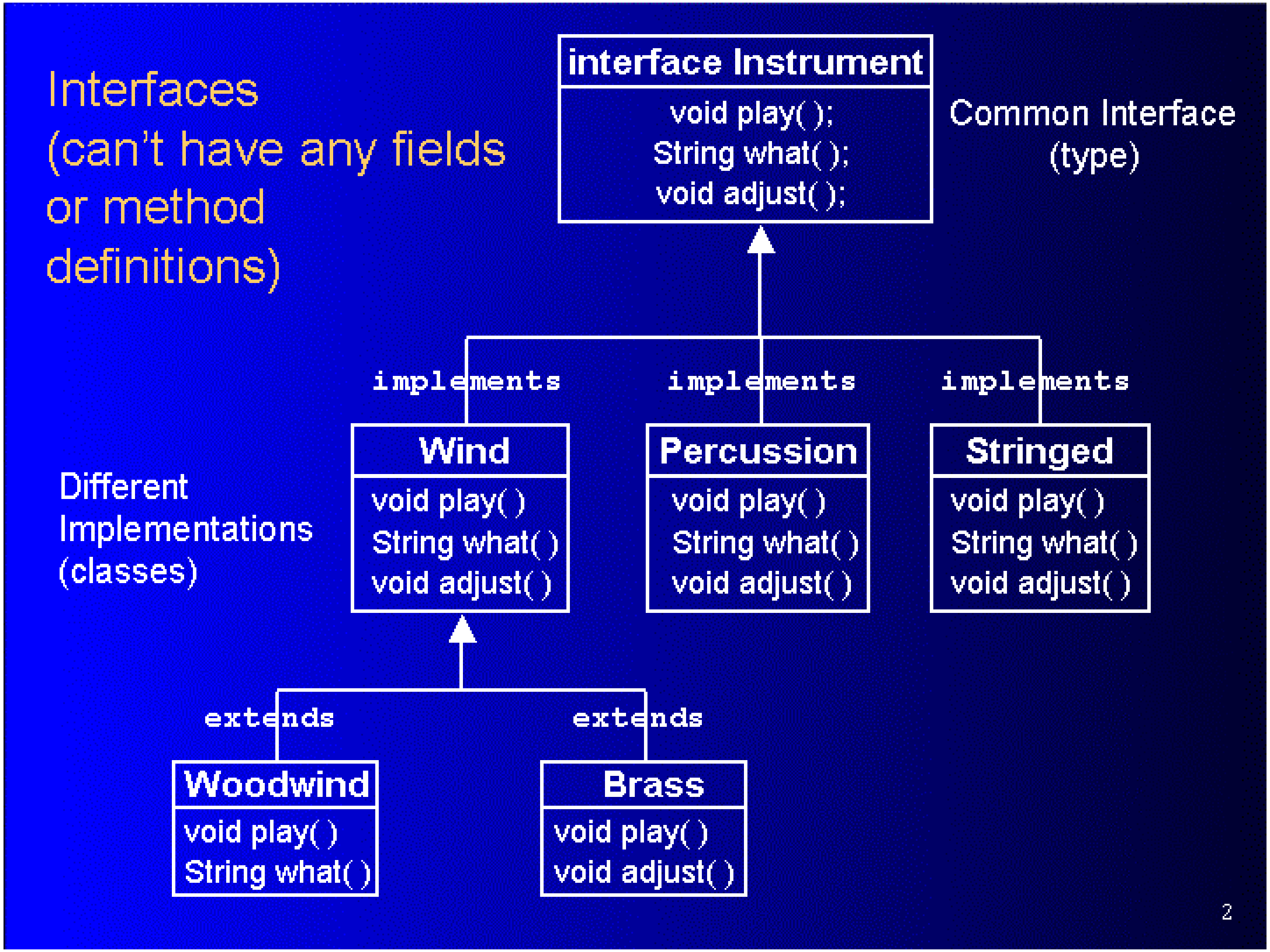
Example – orchestra with concrete classes



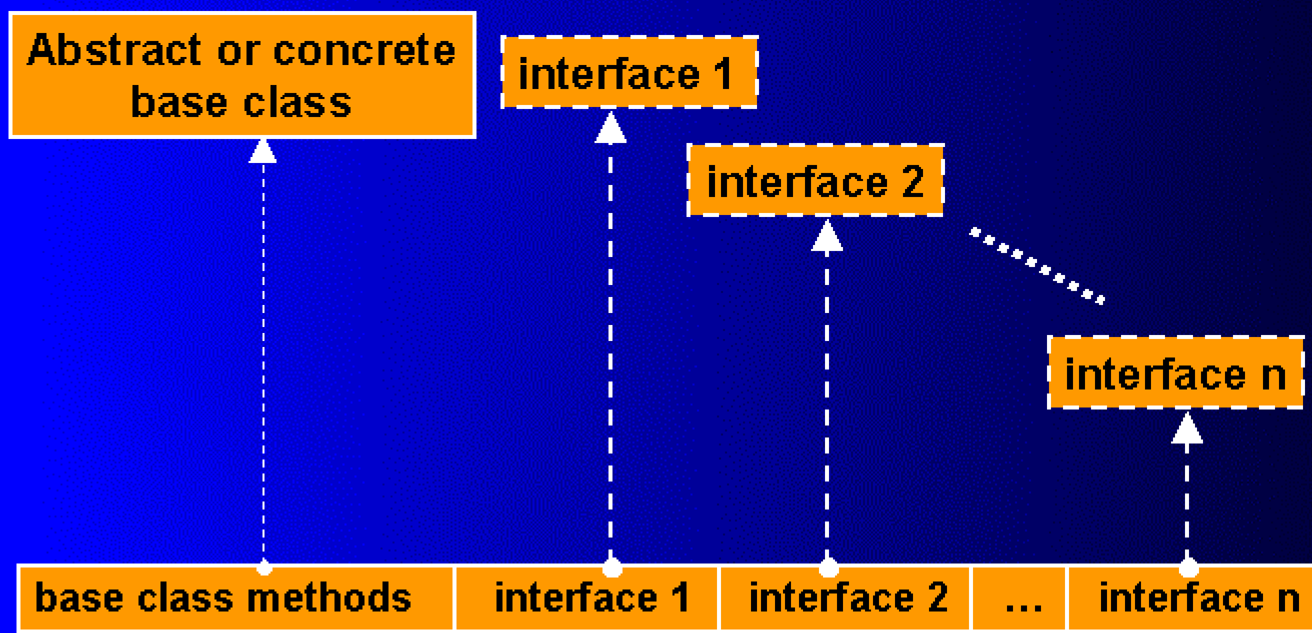
Example – orchestra with abstract classes



Example – orchestra with interfaces



“Multiple Inheritance” in Java



New class has combined interfaces of all types, but uses only one physical implementation: that of the concrete base class

```java
interface CanFight {
  void fight();
}
interface CanSwim {
  void swim();
}
interface CanFly {
  void fly();
}
class ActionCharacter {
  public void fight() {}
}
class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
}
```

```java
public class Adventure {
  static void t(CanFight x) { x.fight(); }
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) { x.fly(); }
  static void w(ActionCharacter x) { x.fight(); }
  public static void main(String args[]) {
    Hero h = new Hero();
    t(h); // Treat it as a CanFight
    u(h); // Treat it as a CanSwim
    v(h); // Treat it as a CanFly
    w(h); // Treat it as an ActionCharacter
  }
}
```

Java “Multiple Inheritance”

- To add extra interfaces, *not* to combine implementations (use composition for that)
- Use it if you need to upcast to more than one base type
- Guideline: use interfaces when possible, avoid abstract classes
  - You never know when you’ll need to combine interfaces; any sort of concreteness prevents it