Objects First With Java
A Practical Introduction Using BlueJ

Further abstraction techniques

Abstract classes and interfaces

Main concepts to be covered

- Abstract classes
- Interfaces
- Multiple inheritance

Idea:

- Enhance class structures
- Improve maintainability & extendibility

Simulations – nothing strange for CSE people

- Programs often used to simulate real-world activities or phenomena:
  - traffic (rail, vehicle, pedestrian, …)
  - weather and climate
  - chemistry
  - fluid dynamics
  - population predictions
  - …

Simulations

- They are typically only partial simulations – tackling and revealing part of the story only.
- They usually involve simplifications (models).
  - Greater detail has the potential to provide greater accuracy.
  - Greater detail typically requires more resources:
    - Processing power.
    - Memory.
    - Simulation time.

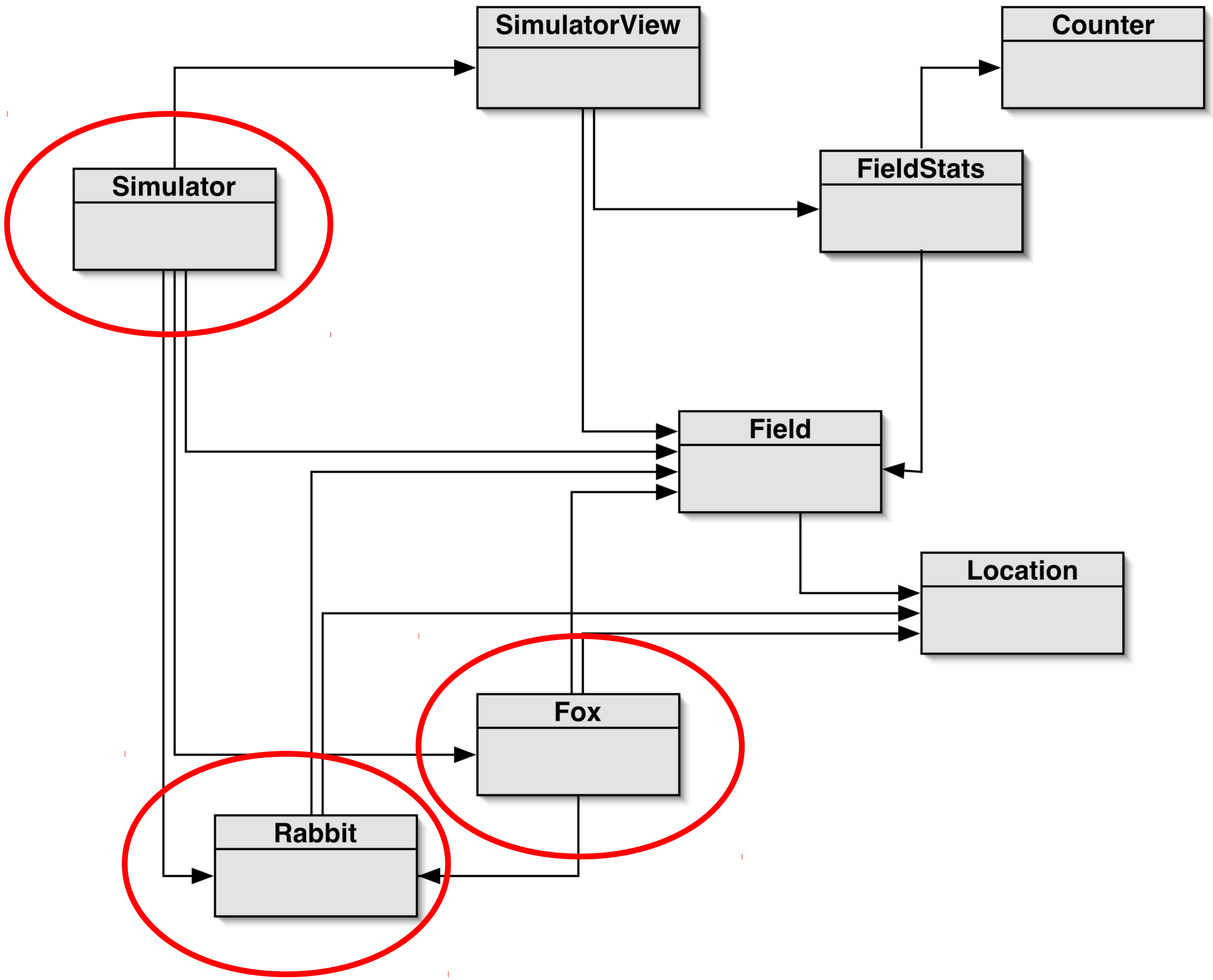
Benefits of simulations – who is still to be convinced???

- Understand – optimize – predict.
- Support useful prediction.
  - The weather.
- Allow experimentation.
  - Safer, cheaper, quicker.
- Example:
  - ‘How will the wildlife be affected if we cut a highway through the middle of this national park?’

Predator-prey simulations

- A standard scenario in population dynamics.
- 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



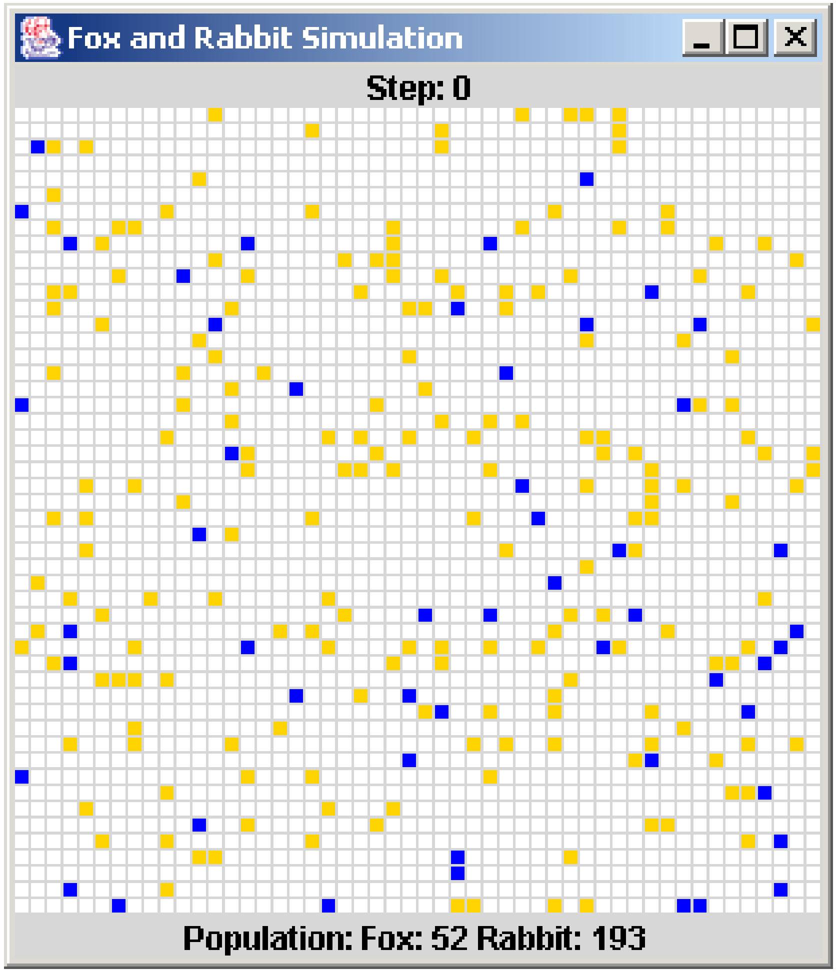
Main classes of interest

- `Fox`
  - Simple model of a type of predator.
- `Rabbit`
  - Simple model of a type of prey.
- `Simulator`
  - Creates the simulation's initial state.
  - Manages the overall simulation task (i.e. performs a sequence of simulation steps where each animal is allowed to move).
  - Holds a collection of foxes and rabbits.

The remaining classes

- `Field`
  - Represents a 2D enclosed field with positions arranged in matrix-type.
- `Location`
  - Represents a 2D position and can hold one animal at most.
- `SimulatorView`, `FieldStats`, `Counter`
  - Maintain statistics and present a view of the field – nothing to do with the model!

Example of the visualization



A Rabbit's state

```
public 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;

    Methods omitted.
}
```

Properties valid for all rabbits:
Breeding age, maximum age,
breeding probability, max. litter size, ...

A Rabbit’s behaviour

- Managed from the `run` method.
- Movement executed and 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.
- Random components: direction, breed yes/no.

Rabbit simplifications

- Rabbits do not have different genders.
  - In effect, all are female (which makes breeding really interesting …).
- The same rabbit could breed at every step (they are hard workers …).
- All rabbits die at the same age.
- Others?

A Fox’s state

A lot of similarity!

```
public 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 fox's food level, which is increased
    // by eating rabbits.
    private int foodLevel;

    Methods omitted.
}
```

A Fox's behaviour

- Managed from the `hunt` method.
- Foxes also age and breed.
- They get hungry.
- Hence, they hunt for food in adjacent locations.
- If a fox finds a rabbit in an adjacent location, the rabbit is killed, and the fox's food level is increased.

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 of the simulation in the constructor.
  - The `populate` method:
    - Each animal is given a random starting age.
  - The `simulateOneStep` method:
    - Iterates over the population.
    - Two `Field` objects are used: `field` and `updatedField`.

The update step – core of `simulateOneStep`

```
if(animal instanceof Rabbit) {
    Rabbit rabbit = animal;
    rabbit.run(updatedField, newAnimals);
}
else if(animal instanceof Fox) {
    Fox fox = animal;
    fox.hunt(field, updatedField, newAnimals);
}
```

`instanceof`: checks whether a given object is an instance of a given class

Missing: check whether the animal is still alive – and removing it if not!

Room for improvement (revisited, to some extent)

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

The Animal superclass

- Place common fields in `Animal`:
  - `age`, `alive`, `location`
- Method renaming (polymorphic method calls) to support information hiding:
  - `run` and `hunt` become `act`.
- `Simulator` can now be significantly decoupled (neither rabbits nor foxes, just animals!).

Revised (decoupled) iteration

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

The `act` method of Animal

- Static type checking requires an `act` method in `Animal`– although it will never be executed!
- There is no obvious shared implementation – either `run` or `hunt` shall be executed, and nothing else.
- Define `act` as abstract:

```
abstract public void act(Field currentField,
                         Field updatedField,
                         List newAnimals);
```

Abstract classes and methods

- Abstract methods
  - have `abstract` in the signature;
  - have no body;
  - make the class abstract.
- Abstract classes
  - cannot be instantiated;
  - are the only classes with abstract methods.
- Concrete subclasses complete the implementation (i.e. must provide implementations for all inherited abstract methods).

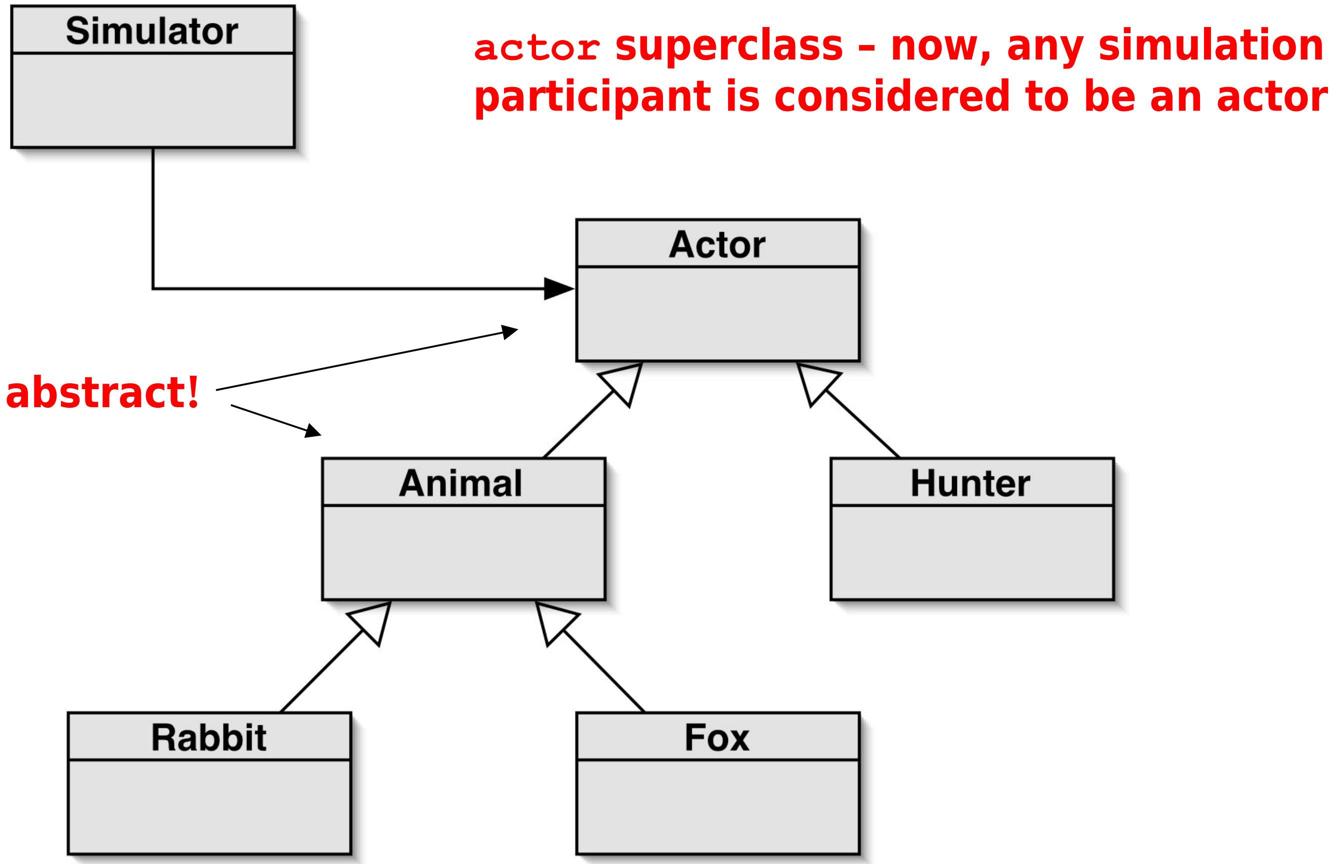
The Animal class

```
public abstract class Animal
{
    fields omitted

    /**
     * Make this animal act - that is: make it do
     * whatever it wants/needs to do.
     */
    abstract public void act(Field currentField,
                             Field updatedField,
                             List newAnimals);

    other methods omitted
}
```

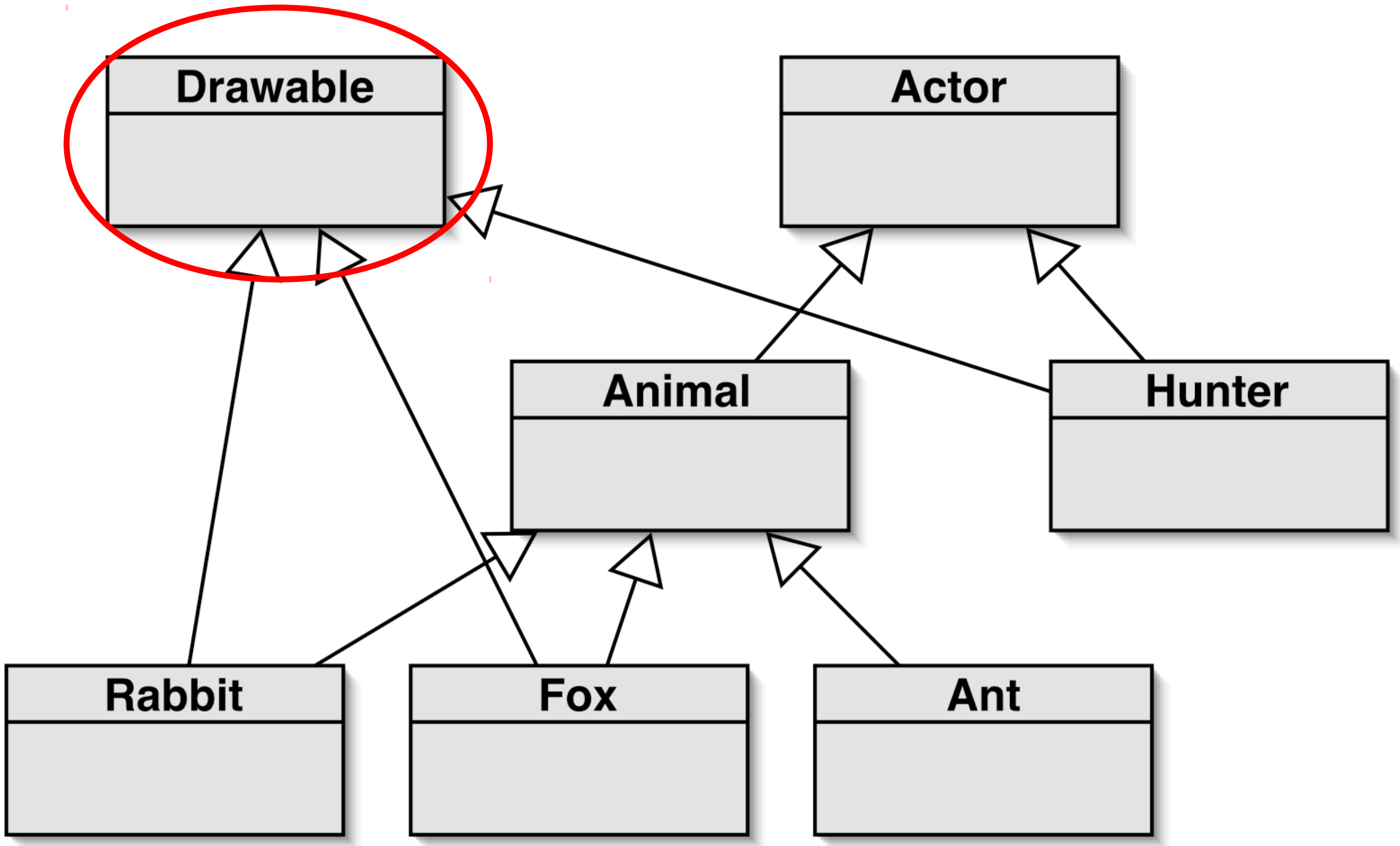
Further abstraction



Selective drawing (multiple inheritance)

- Idea: separate the visualization from the simulation.
- Not everything existing/simulated shall be drawn on the screen.
- Hence, we may need a second superclass!

Selective drawing (multiple inheritance)



Multiple inheritance

- Having a class inherit directly from multiple ancestors.
- Then, the subclass has all the features of all superclasses, and those defined in the subclass itself!
- Each language has its own rules.
  – How to resolve competing definitions?
- Java forbids it for classes.
- Java permits it for interfaces.
  – No competing implementation.

Interfaces

- Specification of behaviour for usage outside.

- At first glance, interfaces are similar to classes.
- However, they do not include method bodies.
- Hence, they are similar to abstract classes with abstract methods only.
- Properties:
  - Key word `interface` instead of `class`.
  - Abstract methods only, no constructors.
  - Public method signatures only.
  - `final static` fields only.

An Actor interface

```
public interface Actor
{
    /**
     * Perform the actor's daily behaviour.
     * Transfer the actor to updatedField if it is
     * to participate in further steps of the simulation.
     * @param currentField The current state of the field.
     * @param location The actor's location in the field.
     * @param updatedField The updated state of the field.
     */
    void act(Field currentField, Location location,
             Field updatedField);
}
```

Classes implement an interface

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

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

`extend` for class inheritance, `implement` for interface inheritance.

Extend at most one class, implement any number of interfaces!

`Animal` stays a class - it contains concrete methods with bodies!

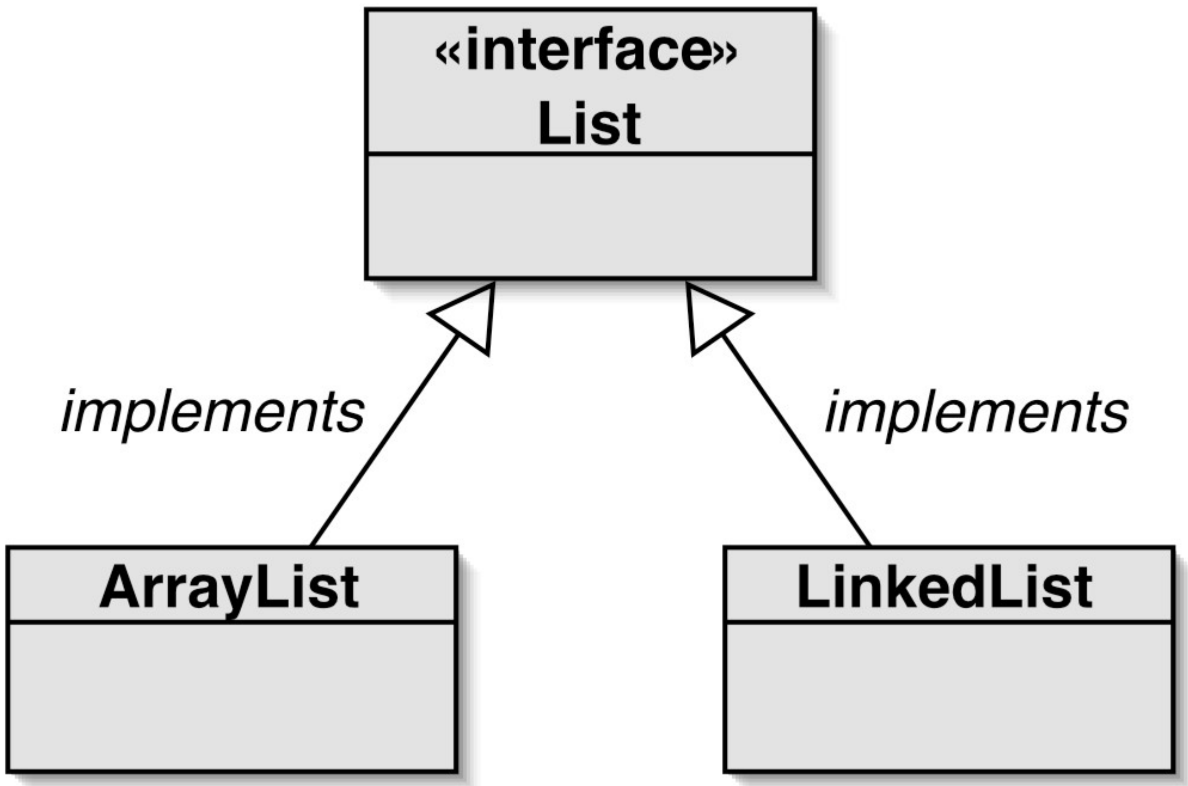
Interfaces as types

- What’s the gain if no code is implemented to be inherited?
- Implementing classes do not inherit code, but ...
- ... implementing classes are subtypes of the interface type.
- So, polymorphism is available with interfaces as well as classes.

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.

Alternative implementations



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.

Interfaces

- Interfaces provide specification without implementation.
  - Interfaces are fully abstract.
- Interfaces support polymorphism.
- Java interfaces support multiple inheritance.