

Outline

- Interfaces
- Abstract classes
- Casting revisited

Interfaces

- Define a contract that a class must live by.
- A class may implement unlimited interfaces
 - May only inherit from one parent
- Allows customization of algorithm.

Interface Example

```
public interface Racer {  
    // constants  
    int MAX_SPEED = 576;  
    // method signatures  
    void run();  
}
```

- `class` replaced by `interface` keyword.
- Has attribute (generally should not).
- No method body.
- Access modifier? Default: `public`.

Implementing an Interface

```
public class Cheetah extends Animal implements Racer {  
    public void run() {  
        setSpeed(70);  
        move(2.5);  
    }  
}
```

- Notice the keyword `implements` .

Missing interface method

- What happens if we forget?

```
public class Cheetah implements Racer {  
    private int spots = 42;  
}
```

```
$ javac Cheetah.java  
Cheetah.java:1: error:  
Cheetah is not abstract and does not override abstract  
method run() in Racer  
public class Cheetah implements Racer {  
      ^  
1 error
```

- Thanks compiler. I did forget.

Example 2: Old McDonald

```
public interface OldMcDonald {  
    String name();  
    String speak();  
}
```

```
public class Dog extends Animal implements OldMcDonald {  
    public String name() {  
        return "Dog";  
    }  
  
    public String speak() {  
        return "Bark";  
    }  
}
```

Example 2: Old McDonald (cont.)

```
public class OldMcDonaldSong {  
    public void sing(List<OldMcDonald> animals) {  
        for (OldMcDonald animal: animals) {  
            System.out.println("...had a " + animal.name());  
            System.out.println("...with a " + animal.speak());  
        }  
    }  
}
```

- A list of animals will sing several verses.

Mixing interfaces and inheritance

```
public class FarmAnimals {  
    private List<Animal> animals = new ArrayList<Animal>();  
    public FarmAnimals(List<Animal> animals) {  
        animals.addAll(animals);  
    }  
  
    public putToWork() {  
        for (Animal a : animals) {  
            a.work();  
        }  
    }  
}
```

- Now put the farm animals back to work.

Interface inheritance

```
public interface Rude {  
    void mock();  
}
```

```
public interface Grumpy extends Rude {  
    void complain();  
}
```

- Implementing Grumpy means you need to implement both:
 - void mock()
 - void complain()

Abstract Classes



Abstract Classes

- Does this make sense?

```
Animal genericAnimal = new Animal();
```

- How do you prevent this?

Abstract Example: Cannot be instantiated

```
public abstract class Animal {  
    public Animal() {  
    }  
}
```

```
Animal a = new Animal();
```

```
error: Animal is abstract; cannot be instantiated  
        Animal a = new Animal();  
                        ^
```

1 error

Inherited Fields

```
public abstract class Animal {  
    protected int age;  
}
```

```
public class Cat extends Animal {  
    public void speakAge() {  
        System.out.println("I am " + age + " years old");  
    }  
}
```

- Parent fields can be accessed as though they are part of the child.

Inherited Fields: Trivia

```
public abstract class Animal {  
    protected int age;  
}
```

```
public class Hippo extends Animal {  
    public Hippo(int a) {  
        age = a;  
    }  
}
```

```
Hippo happy = new Hippo(24);  
Hippo sleepy = new Hippo(12);
```

- Is happy.age 12 or 24?
- What keyword would switch that?

Abstract Methods

```
public abstract class Animal {  
    public abstract void speak();  
}
```

- What does that look like?
- Why not just use an interface?
- `abstract public void speak();` also works.

Abstract Methods: Mixed

```
public abstract class Animal {  
    public abstract void speak();  
  
    public void recitePoetry() {  
        System.out.println("Two roads diverged in a ...");  
    }  
}
```

- Some methods implemented in parent.
- Some methods implemented in children.

Abstract Methods: Mixed

```
public abstract class Animal {  
    public abstract void speak();  
  
    public void recitePoetry() {  
        System.out.println("Two roads diverged in a ...");  
    }  
}
```

```
public class Rat extends Animal {  
    public void speak() {  
        System.out.println("squeak!");  
    }  
  
    public void recitePoetry() {  
        System.out.println("I do not do poetry.");  
    }  
}
```

Polymorphic Parameters

```
class Plane {}  
class Jet extends Plane {}  
  
public class Main {  
    public String fly(Plane p) { return "plane"; }  
    public String fly(Jet j) { return "jet"; }  
  
    public static void main(String[] args) {  
        Plane plane = new Plane();  
        System.out.println("I am a " + fly(plane));  
  
        Jet jet = new Jet();  
        System.out.println("I am a " + fly(jet));  
  
        Plane mix = new Jet();  
        System.out.println("I am a " + fly(mix));  
    }  
}
```

- What will the output be?

Real World Example: AbstractList

```
public abstract class AbstractList<E> {  
    abstract public void add(int index, E element);  
  
    public boolean addAll(int index, Collection<E> c) {  
        boolean modified = false;  
        for (E e : c) {  
            add(index++, e);  
            modified = true;  
        }  
        return modified;  
    }  
}
```

- **addAll** same for each type of list.
- **add** method will vary. Implementation detail supplied by each child.

Casting revisited (upcasting)

- Classes can be safely upcast to:
 - Any interfaces it implements
 - Its parent class
 - Anything its parent can be cast to

```
interface GPInterface1{}  
interface GPInterface2{}  
public class Grandparent implements GPInterface1, GPInterface2{}  
  
public class Parent extends Grandparent{}  
  
interface ChildInterface{}  
public class Child extends Parent implements ChildInterface{}  
  
// Grandparent can be cast to GPInterface1, GPInterface2, Parent, and Object  
// Parent can be cast to Grandparent, GPInterface1, GPInterface2, and Object  
// Child can be cast to Parent, GPInterface1, GPInterface2, and Object
```

Casting revisited (Downcasting)

Downcasting is casting to a more specific type. It is not safe and may crash the application.

```
Parent actuallyAChild = new Child();  
Parent notAChild = new Parent();  
  
Child works = (Child)actuallyaChild;  
child fails = (Child)notAChild;
```

Casting revisited (instanceof)

Instance of checks if an object is of a more specific type

```
Parent notAChild = new Parent();  
  
if(notAChild instanceof Child) {  
    child skipped = (Child)notAChild;  
}
```

Summary

- Interface
 - Contract for all implementers to follow.
 - Contains one or more empty methods.
 - May contain fields, but generally shouldn't.
- Abstract Class
 - Cannot be instantiated.
 - May contain fields.
 - Abstract methods required by children.
 - Can contain method bodies.
 - Children may override non-abstract methods.
- Casting
 - Upcasts are safe
 - Downcasts can crash an application