CIS 351-Data Structure-Abstract Classes and Polymorphism
Feb 20, 2020

Dr. Farzana Rahman

Syracuse University

Polymorphism

- Suppose we create the following object reference variable (`Holiday` can be a class or an interface):

```
 Holiday day;
```

- Java allows this reference to point to a `Holiday` object or to any object of <u>any compatible type</u>

- If `class Christmas extends Holiday` or if `class Christmas implements Holiday`, a `Christmas` object is a compatible type with a `Holiday` object and a reference to one can be stored in the reference variable `day`:

```
 day = new Christmas();
```

References and Inheritance

- Assigning a child object to a parent reference is considered to be a **widening conversion (Upcasting)**, and can be performed by **simple assignment**
- The widening conversion is the most useful
- Assigning a parent object to a child reference can be done, but it is considered a **narrowing conversion (Downcasting)** and two rules/guidelines apply:
  - A narrowing conversion must be done with a cast
  - A narrowing conversion should only be used to restore an object back to its original class (back to what it was "born as" with the new operator)

```
public abstract class Animal implements Mammal
{
    public void eat() {
        System.out.println("Eating...");
    }
    public void move() {
        System.out.println("Moving...");
    }
    public void sleep() {
        System.out.println("Sleeping...");
    }
}
```

```
public interface Mammal
{
    public void eat();
    public void move();
    public void sleep();
}
```

```
public class Dog extends Animal {
    public void bark() {
        System.out.println("Gow gow!");  }
    public void eat() {
        System.out.println("Dog is eating..."); }
}

public class Cat extends Animal {
    public void meow() {
        System.out.println("Meow Meow!");  }
}
```

Upcasting

- ***Upcasting*** is casting a subtype to a supertype, upward to the inheritance tree. Let's see an example:

```
Dog dog = new Dog();
Animal anim = (Animal) dog;
anim.eat();
```

- Here, we cast the Dog type to the Animal type. Because Animal is the supertype of Dog, this casting is called upcasting. Upcasting is always safe, as we treat a type to a more general one.

Example:

```
Mammal mam = new Cat();
Animal anim = new Dog();
```

Downcasting

- ***Downcasting*** is casting to a subtype, downward to the inheritance tree. Let's see an example:

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

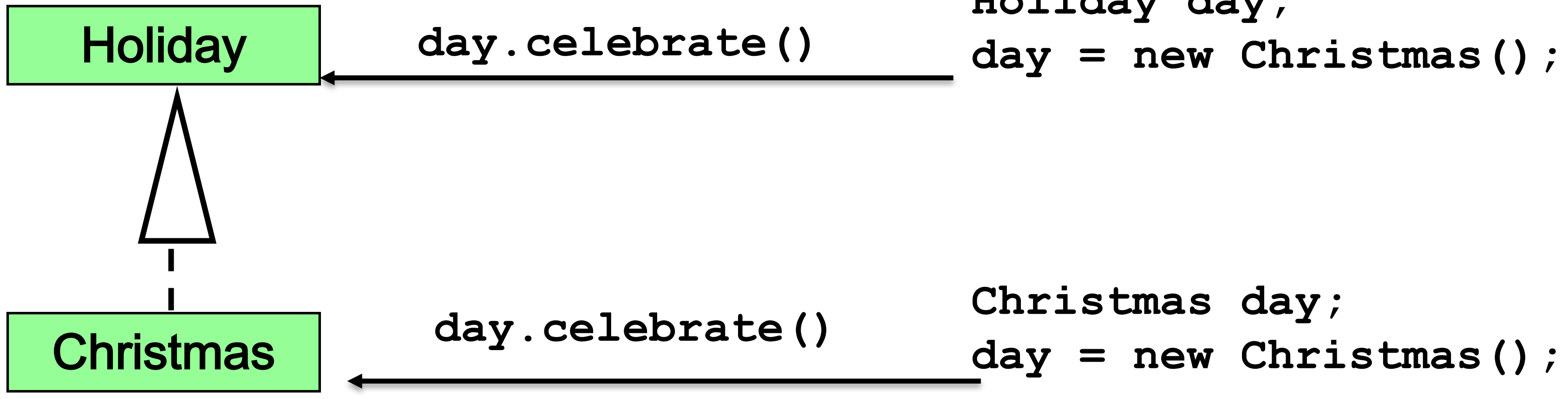
- Here, we cast the Animal type to the Cat type. As Cat is subclass of Animal, this casting is called **downcasting**.
- Unlike upcasting, downcasting can **fail** if the actual object type is not the target object type. For example:

```
Animal anim = new Cat();
Dog dog = (Dog) anim;
```

Polymorphism via Inheritance

- It is the type of the object **being referenced**, that determines which method is invoked
- If the `Holiday` class has a `celebrate` method, and the `Christmas` class overrides it, consider the following invocation:

```
day.celebrate();
```



```
Holiday day;
day = new Christmas();
```

```
Christmas day;
day = new Christmas();
```

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable

```
Speaker current;
```

- The `current` reference can be used to point to any object of any class that implements the `Speaker` interface
- The version of `speak` that the following line invokes depends on the type of object that `current` is referencing

```
current.speak();
```

Polymorphism via Interfaces

- Suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, but each provides a distinct version of the `speak` method
- In the following code, the first call to `speak` invokes the `Philosopher` method and the second invokes the `Dog` method:

```
 Speaker guest = new Philosopher();
guest.speak();  // To be or not to be
guest = new Dog();
guest.speak();  // Arf, Arf
```

Abstract classes

- We' ve covered two extremes of inheritance:
  - **interfaces:** all methods are *abstract* in superclass and superclass (ie interface) serves to define common **type**
  - **non-abstract superclasses:** all methods are *non-abstract* in superclass and subclass actually inheritents implementation.
    - good for code re-use
    - also good for defining common type

Abstract classes, cont.

- Using abstract methods, we can actually program in between these two models.
- This is done by creating superclasses that are mixtures of abstract and non-abstract methods plus instance variables.
- The non-abstract methods and instance variables are inherited just like with regular classes.
- The abstract methods are treated just like interface methods – they must be implemented.

Guess Output

```
abstract class Base {
    abstract void fun();
}
class Derived extends Base {
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {

        // Uncommenting the following line will cause compiler
error as the
        // line tries to create an instance of abstract class.
        // Base b = new Base();

        // We can have references of Base type.
        Base b = new Derived();
        b.fun();
    }
}
```

Toy Class Demo

Rules for abstract classes

- Any method in a class may be given the *abstract* keyword.

```
public abstract void foo(...)
```

- If one or more methods in a class are abstract, the class itself must be declared abstract

```
abstract class Foo{ ...}
```

- Abstract classes may be subclassed, but not instantiated.

More on abstract classes

- Abstract methods must have no meat.
- It is not required that a subclass implement every (or any) abstract method in an abstract superclass.
- However, if **all** abstract methods are not implemented in the subclass, the subclass must be declared abstract.
- classes with all abstract methods are almost exactly like interfaces (what are the differences?)

More on abstract classes

- In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base {
    void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base { }

class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.fun();
    }
}
```

More on abstract classes

- An abstract class can contain constructors and it is called when an instance of a inherited class is created.

```
// An abstract class with constructor
abstract class Base {
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
       Derived d = new Derived();
    }
}
```