CSM 61B
Fall 2017

Inheritance

Mentoring 4: September 18, 2017

1 Dogs Yay

1.1
```
class Dog {
    public void walk() {
        System.out.println("The dog is walking");
    }
}
class Beagle extends Dog {
    @Override
    public void walk() {
        System.out.println("The beagle is walking");
    }
}
```

What would Java display?

(a)
```
Dog fido1 = new Dog();
fido1.walk();
```

The dog is walking

(b)
```
Beagle fido2 = new Beagle();
fido2.walk();
```

The beagle is walking

(c)
```
Beagle fido3 = new Dog();
fido3.walk();
```

Compile-time error.

A container meant for `Beagles` can't contain `Dogs`.

(d)
```
Dog fido4 = new Beagle();
fido4.walk();
```

The beagle is walking

A container for `Dogs` can contain `Beagles`. At compile time, `fido.walk()` is linked to `Dog.walk()` but at runtime, this method is overridden by `Beagle.walk()`.

Meta: It can be useful to talk about the static types of variables as boxes with labels and the dynamic type of an object as what you place in the box. So long as what you place in the box fits the label (i.e. a `Beagle` is a `Dog`, so a dynamic `Beagle` can be placed in a static `Dog`) then no compiler errors occur. Remember to mention dynamic method lookup.

1.2 What would each call in `Poodle.main` print? If a line would cause an error, determine if it is a compile-error or runtime-error.

```
class Dog {
    void bark(Dog dog) {
        System.out.println("bark");
    }
}

class Poodle extends Dog {
    void bark(Dog dog) {
        System.out.println("woof");
    }
    void bark(Poodle poodle) {
        System.out.println("yap");
    }
    void play(Dog dog) {
        System.out.println("no");
    }
    void play(Poodle poodle) {
        System.out.println("bowwow");
    }
    public static void main(String[] args) {
        Dog dog = new Poodle();
        Poodle poodle = new Poodle();

        dog.play(dog)         // Compile-error
        dog.play(poodle)      // Compile-error
        poodle.play(dog)      // no
        poodle.play(poodle)   // bowwow

        poodle.bark(dog)      // woof
        poodle.bark(poodle)   // yap
        dog.bark(dog)         // woof
        dog.bark(poodle)      // woof

    }
}
```

Meta: This requires a walkthrough. First, setup the rules: dynamic method lookup, method overriding, static and dynamic types, and compile-error vs. runtime error. Give students a chance to work, then clear misconceptions.

2 An Appealing Appetizer

2.1

```
public interface Consumable {
    public void consume();
}
public abstract class Food implements Consumable {
    String name;
    public abstract void prepare();
    public void play() {
        System.out.println("Mom says, 'Don't play with your food.'");
    }
}
public class Snack extends Food {
    public void prepare() {
        System.out.println("Taking " + name + " out of wrapper");
    }
    public void consume() {
        System.out.println("Snacking on " + name);
    }
}
```

(a) Compare and contrast interfaces and abstract classes.

- Java classes cannot extend multiple superclasses (unlike Python) but classes can implement multiple interfaces.
- Interfaces are implicitly public.
- Interfaces can't have fields declared as instance variables; any fields that are declared are implicitly **`static`** and **`final`**.
- Interfaces use the **`default`** keyword to declare *concrete* implementations while abstract classes use the **`abstract`** keyword to declare *abstract* implementations.
- Interfaces define the way we interact with an implementing object or functions of an object. Conversely, abstract classes define an "is-a" relationship and tell us more about the object's fundamental identity.

Meta: Warn students to be careful searching about this topic as CS 61B uses Java 8 but most content online covers Java 7, which behaves differently.

(b) Do we need the `play` method in `Snack`?

No, we do not need the `play` method because it's already defined in the abstract class. Java will lookup the parent class's method if it cannot find it in the child class.

(c) Does this compile? `Consumable chips = new Snack();`

Yes, the code compiles since `Snack` inherits from the `Food` class which implements the `Consumable` interface.

Meta: Use this question as a mini-lecture for interfaces to build up to the next question. This question should not take a lot of time.

3 Iterator Interface

In Java, an **iterator** is an object which allows us to traverse a data structure in linear fashion. Every iterator has two methods: `hasNext` and `next`.

```
interface IntIterator {
    boolean hasNext();
    int next();
}
```

3.1 Consider the following code that demonstrates the `IntArrayIterator`.

```
int[] arr = {1, 2, 3, 4, 5, 6};
IntIterator iter = new IntArrayIterator(arr);
if (iter.hasNext()) {
    System.out.println(iter.next());     // 1
}
if (iter.hasNext()) {
    System.out.println(iter.next() + 3); // 5
}
while (iter.hasNext()) {
    System.out.println(iter.next());     // 3 4 5 6
}
```

Define an `IntArrayIterator` class that works as described above.

```
public class IntArrayIterator implements IntIterator {
    private int index;
    private int[] array;
    public IntArrayIterator(int[] arr) {
        array = arr;
        index = 0;
    }
    public boolean hasNext() {
        return index < array.length;
    }
    public int next() {
        int value = array[index];
        index += 1;
        return value;
    }
}
```

3.2 Define an `IntListIterator` class that adheres to the `IntIterator` interface.

```
public class IntListIterator implements IntIterator {
    private IntList node;
    public IntListIterator(IntList list) {
        node = list;
    }
    public boolean hasNext() {
        return node != null;
    }
    public int next() {
        int value = node.first;
        node = node.rest;
        return value;
    }
}
```

Meta: This can be gone through and explained faster since the students just implemented `IntArrayIterator`.

3.3 Define a method, `printAll`, that prints every element in an `IntIterator` **regardless of how the iterator is implemented.**

```
public static void printAll(IntIterator iter) {
    while (iter.hasNext()) {
        System.out.println(iter.next());
    }
}
```

Meta: In general, for all parts of this question, give students time to work on the problem as well as prompting them to think about what they need in order to start them off. (For example, what do you need to do to implement the `IntIterator` interface? What do you need to know for `hasNext` to work?)

4 Pokemon *Extra Practice*

4.1 Identify the errors that occur when running the code to the right.

```
Pikachu p = new Pikachu();
Pokemon a = p;
// (1)
// p = a;
a.eat();
a = new Squirtle();
// (2)
// a.attack();
((Squirtle) a).attack();
Pokemon z = new Pikachu();
// (3)
// Squirtle s = (Squirtle) z;
((Pokemon) p).attack(z);
```

```
public class Pokemon {
    public int hp, power;
    public String cry;
    public String secret;
    public Pokemon() {
        hp = 50;
        cry = "Poke?";
    }
    public Pokemon(String c, int hp) {
        cry = c;
        this.hp = hp;
    }
    public void attack(Pokemon p) {
        p.hp -= power;
    }
    public void eat() {
        System.out.println("nom nom");
    }
}
public class Pikachu extends Pokemon {
    public Pikachu() {
        hp = 100;
    }
    public Pikachu(int hp) {
        super("Pika pika pikachu", hp);
    }
    public void attack(Pokemon p) {
        p.hp = 0;
    }
    public void eat() {
        System.out.println("nom Pika nom");
    }
}
public class Squirtle extends Pokemon {
    public void attack() {
        System.out.println("Water gun!!");
    }
}
```

`(1)` is a compile-time error since you can't assign a static type `Pokemon` to a variable who has a static type `Pikachu`. Can be solved with a cast.

`(2)` is a compile-time error because `Pokemon` does not have a method with the signature `attack()`.

`(3)` is a run-time error because the dynamic type of `z` (`Pikachu`) cannot be cast to a `Squirtle`.