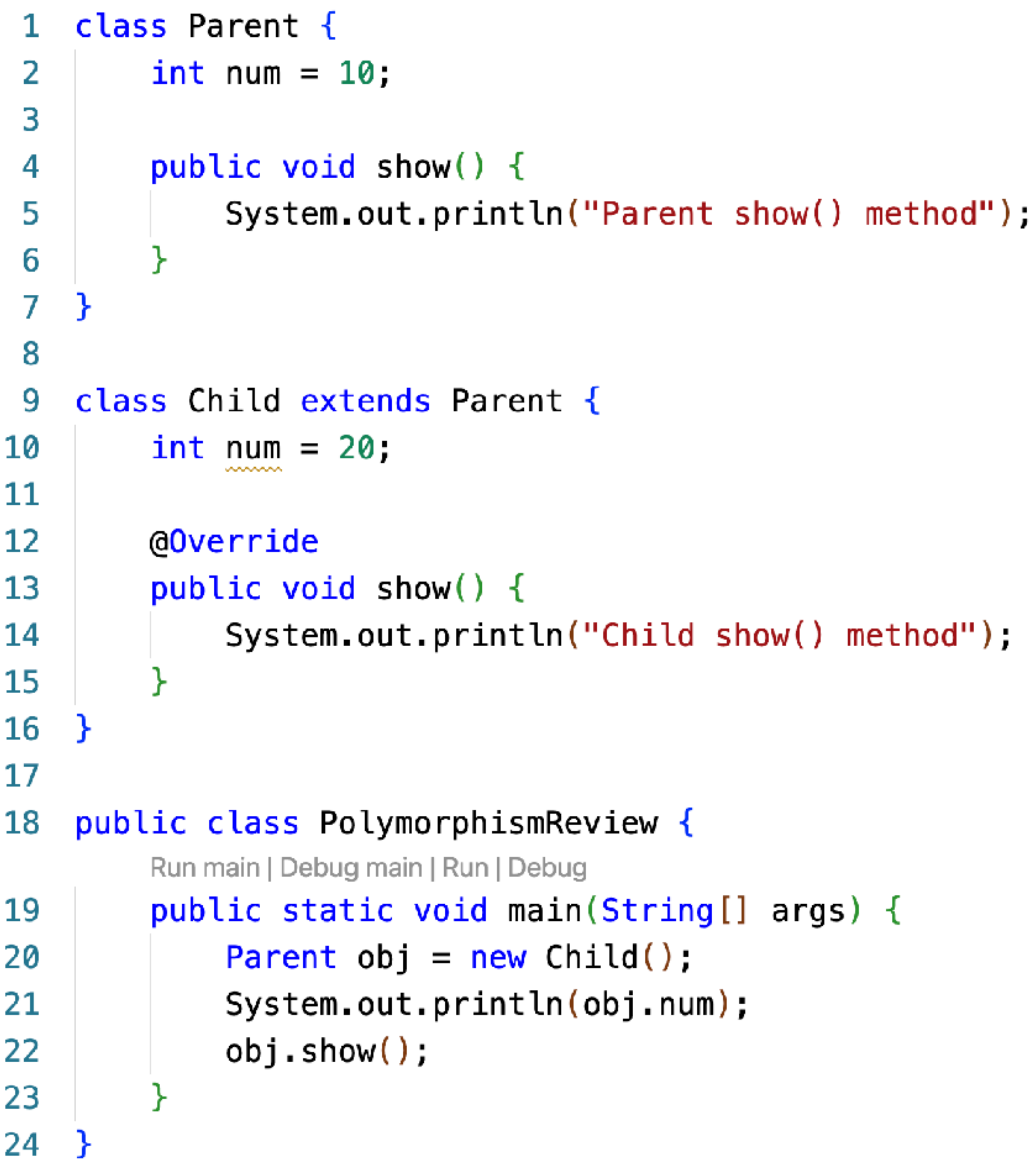
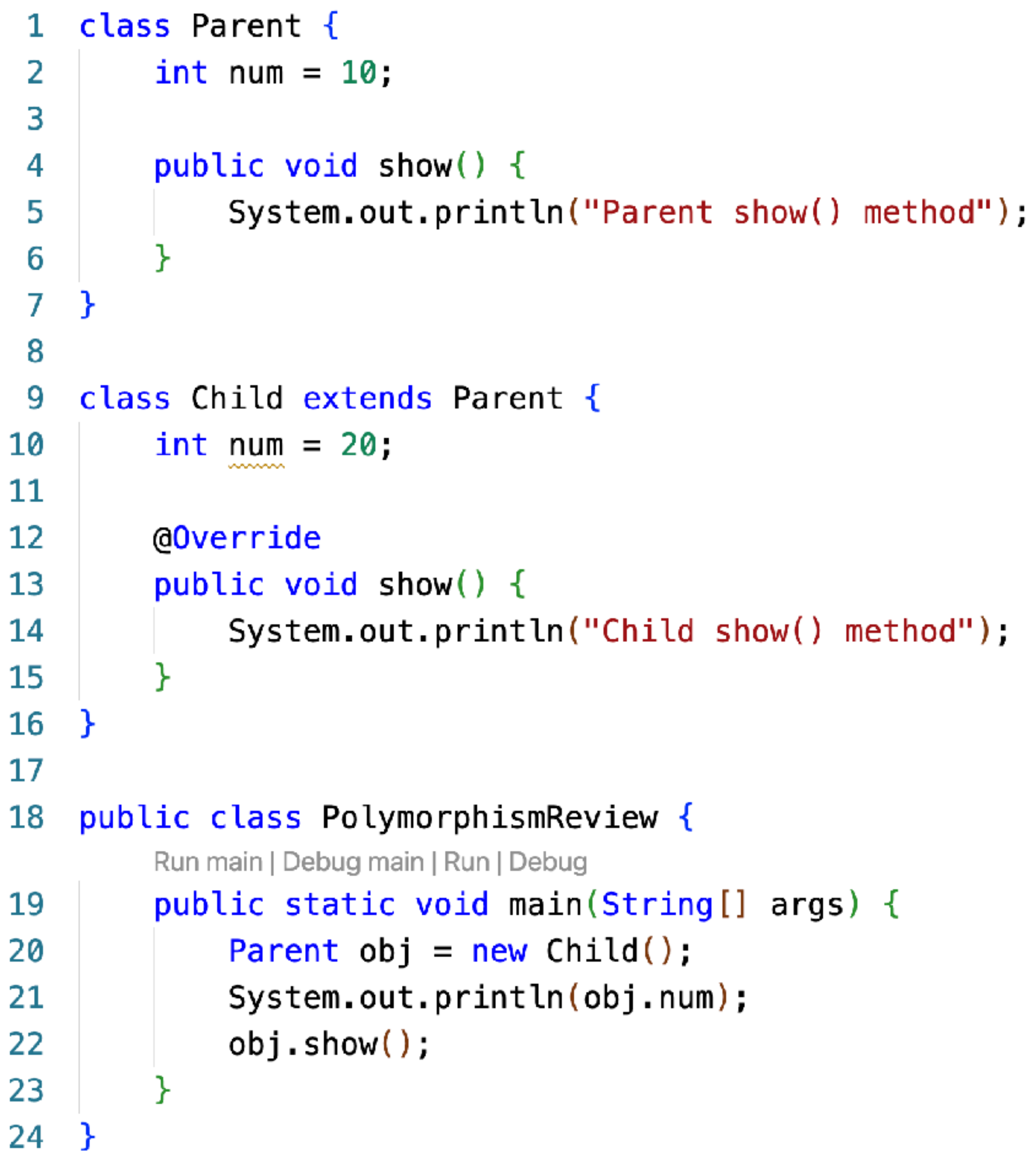
Review: polymorphism



```java
class Parent {
    int num = 10;

    public void show() {
        System.out.println("Parent show() method");
    }
}

class Child extends Parent {
    int num = 20;

    @Override
    public void show() {
        System.out.println("Child show() method");
    }
}

public class PolymorphismReview {
    public static void main(String[] args) {
        Parent obj = new Child();
        System.out.println(obj.num);
        obj.show();
    }
}
```

- What’s printed? Why?
- Why does .show() behave differently from num?
- What keyword can be used to access the value of num in the parent class from the child class?
- How can we modify the code so we can print out 20 for num in main(), while keeping the type of obj as Parent?

Review: polymorphism

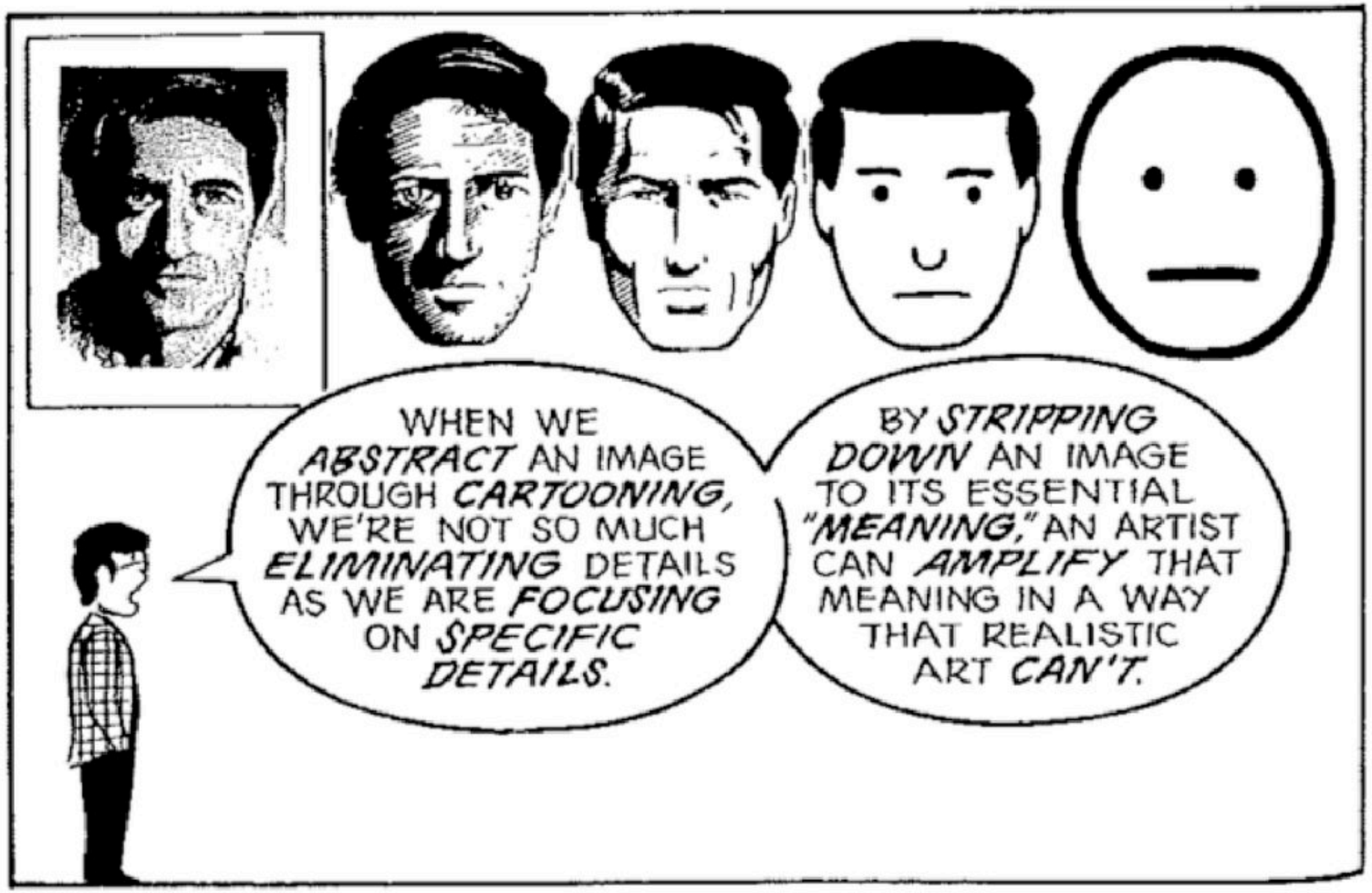


```java
class Parent {
    int num = 10;

    public void show() {
        System.out.println("Parent show() method");
    }
}

class Child extends Parent {
    int num = 20;

    @Override
    public void show() {
        System.out.println("Child show() method");
    }
}

public class PolymorphismReview {
    public static void main(String[] args) {
        Parent obj = new Child();
        System.out.println(obj.num);
        obj.show();
    }
}
```

- What's printed? Why?
  - 20, Child show() method
- Why does .show() behave differently from num?
  - .show() is overridden in the child class, while num hides the parent value
- What keyword can be used to access the value of num in the parent class from the child class?
  - super
- How can we modify the code so we can print out 20 for num in main(), while keeping the type of obj as Parent?
  - create a getter (getNum()) so overriding happens
- **SUMMARY**: instance methods get overridden, but variables (and static methods) are hidden

CS62 Class 5: Interfaces, Generics

Java Fundamentals


WHEN WE ABSTRACT AN IMAGE THROUGH CARTOONING, WE'RE NOT SO MUCH ELIMINATING DETAILS AS WE ARE FOCUSING ON SPECIFIC DETAILS.
BY STRIPPING DOWN AN IMAGE TO ITS ESSENTIAL "MEANING," AN ARTIST CAN AMPLIFY THAT MEANING IN A WAY THAT REALISTIC ART CAN'T.

From Understanding Comics by Scott McCloud

AKA: Today is all about abstraction!

Lecture 5 agenda

- Interfaces
- Generics
- ArrayLists (brief intro)

Interfaces

Interfaces: managing abstraction

- An interface is a form of **abstraction** that is a **contract** of what a class must do. As an abstraction, it does not say *how* a class should do it.
- In Java, an interface is a reference type (like a class), that contains **abstract methods** and **default methods**.
- A class that implements an interface is obliged to implement its abstract methods.
- Interfaces cannot be instantiated (no new keyword). They can only be *implemented* by classes or *extended* by other interfaces.

Example

```
public interface Enrollable{
    void enrollInCourse(String course);
    void withdrawFromCourse(String course);
    void viewCourseSchedule();

    default int getMaxCredits(){
        return 4;
    }
}
```

abstract methods - just include the signature. any class that “implements” the interface has to have these signatures filled out

note syntax - just ; no {}

default method: need “default” keyword in beginning. everything that implements this interface can use this method

all methods are implicitly public in an interface
- no need for “public” modifier

Example

```
class PomonaStudent implements Enrollable{

…
   public void enrollInCourse(String course) {
        // implementation
    }

   public void withdrawFromCourse(String course) {
        // implementation
    }

   public void viewCourseSchedule() {
        // implementation
    }
```

Example

Q: Why don’t we need “implements Enrollable” for FourthYearPomonaStudent?

```
class FourthYearPomonaStudent extends PomonaStudent{

…
   public int getMaxCredits(){
        return 6;
    }
}
```

can override default methods of interfaces

Interfaces

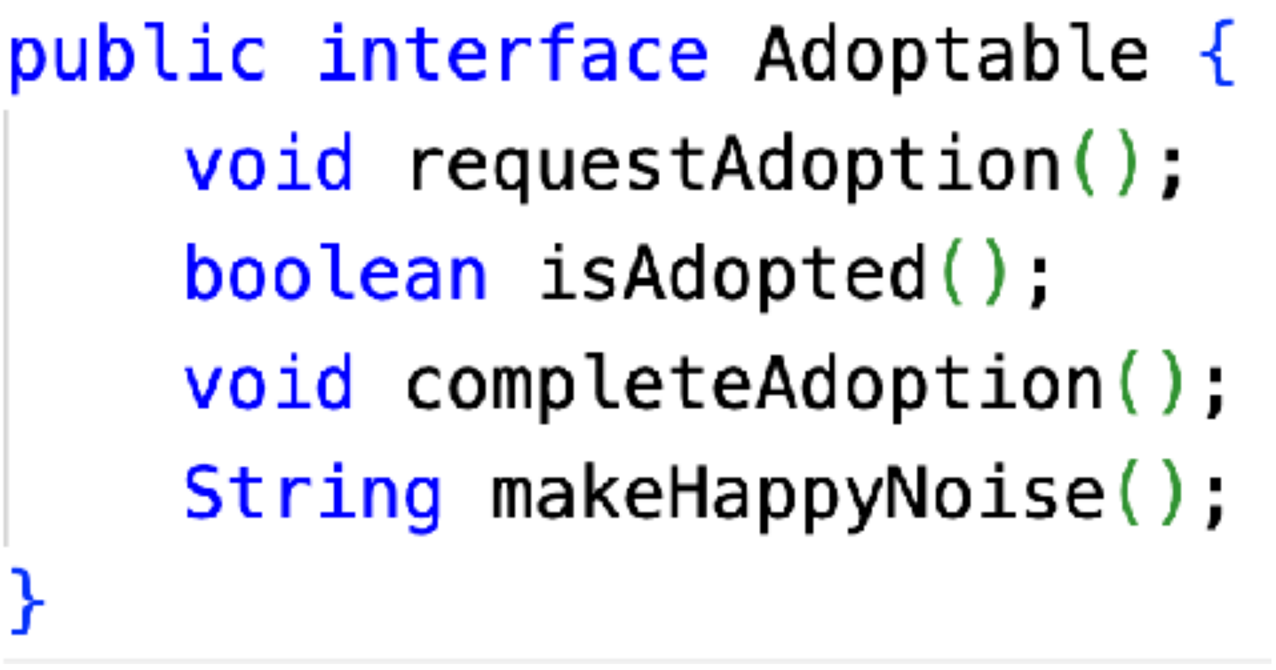
Remember: a class can only extend one class

- A class can implement multiple interfaces.
  - `class A implements Interface1, Interface2{…}`
- An interface can extend multiple interfaces.
  - `public interface GroupedInterface extends Interface1,Interface2{…}`

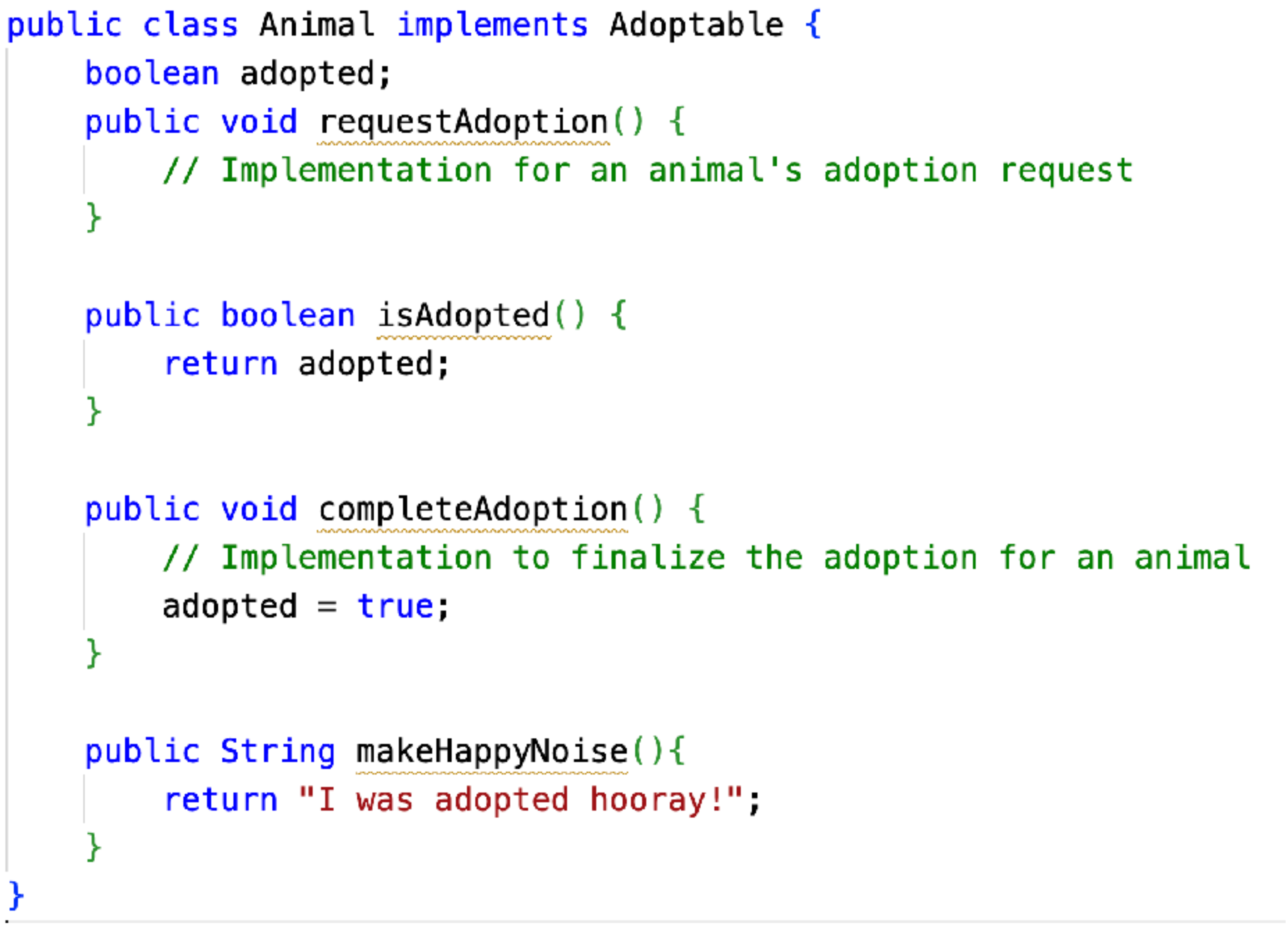
Worksheet time!

- Create an interface called `Adoptable` that contains four abstract methods: a `void requestAdoption()`, `boolean isAdopted()`, `void completeAdoption(), and String makeHappyNoise()`.
- Have the class `Animal` implement the interface. You can provide some very minimal implementation of the methods so that you don’t receive a compile-time error.
- Override the `makeHappyNoise()` in the `Cat` and `Dog` subclasses.

Worksheet answers

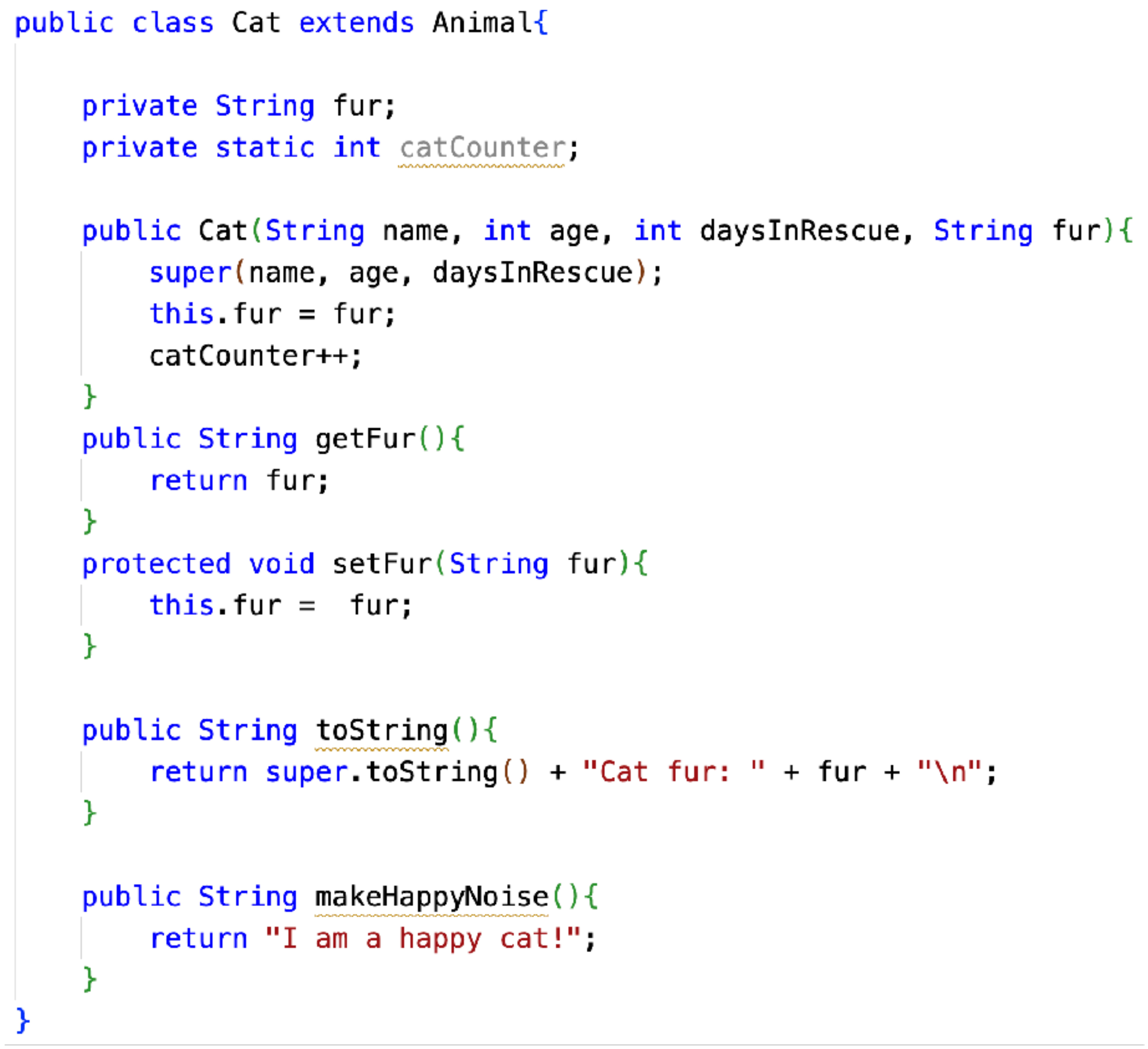


```java
public interface Adoptable {
    void requestAdoption();
    boolean isAdopted();
    void completeAdoption();
    String makeHappyNoise();
}
```

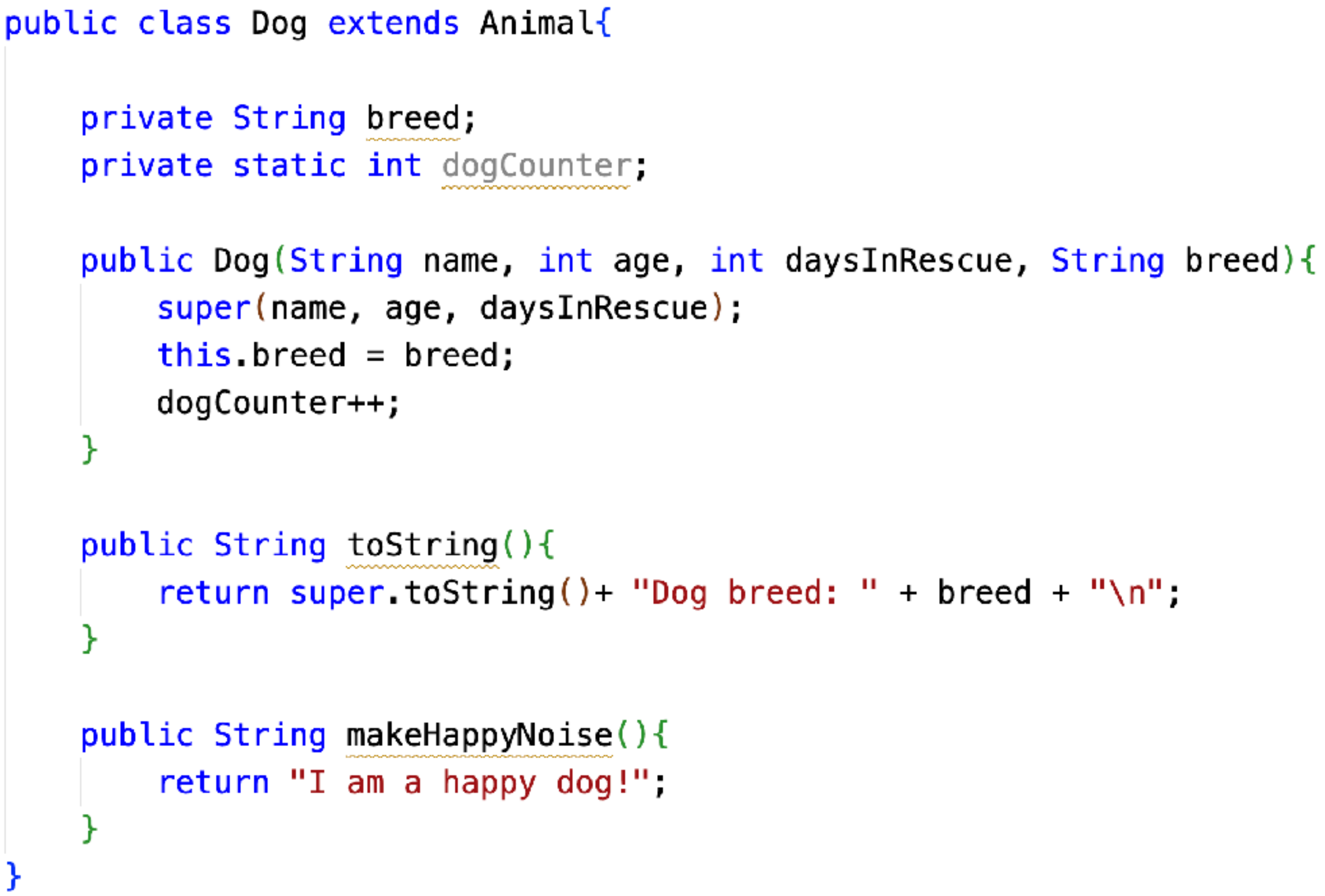


```java
public class Animal implements Adoptable {
    boolean adopted;
    public void requestAdoption() {
        // Implementation for an animal's adoption request
    }

    public boolean isAdopted() {
        return adopted;
    }

    public void completeAdoption() {
        // Implementation to finalize the adoption for an animal
        adopted = true;
    }

    public String makeHappyNoise(){
        return "I was adopted hooray!";
    }
}
```

Worksheet answers



```java
public class Cat extends Animal{

    private String fur;
    private static int catCounter;

    public Cat(String name, int age, int daysInRescue, String fur){
        super(name, age, daysInRescue);
        this.fur = fur;
        catCounter++;
    }
    public String getFur(){
        return fur;
    }
    protected void setFur(String fur){
        this.fur =  fur;
    }

    public String toString(){
        return super.toString() + "Cat fur: " + fur + "\n";
    }

    public String makeHappyNoise(){
        return "I am a happy cat!";
    }
}
```



```java
public class Dog extends Animal{

    private String breed;
    private static int dogCounter;

    public Dog(String name, int age, int daysInRescue, String breed){
        super(name, age, daysInRescue);
        this.breed = breed;
        dogCounter++;
    }

    public String toString(){
        return super.toString()+ "Dog breed: " + breed + "\n";
    }

    public String makeHappyNoise(){
        return "I am a happy dog!";
    }
}
```

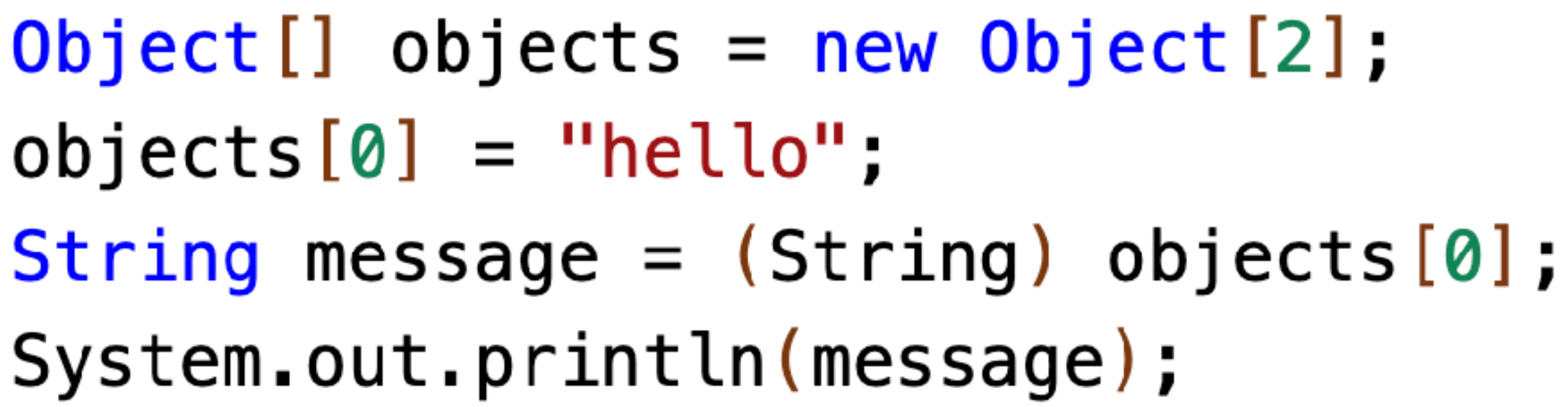
Generics

Towards building our own data structures…

- Arrays in Java are OK, but they're not resizable. Let's define our own data structure that supports adding elements, getting them at an index, removing them, etc…
- As such, we will build an **interface List** that forces any data structure that implements it to implement these operations.
- But what about types? We want our List interface to be able to hold objects of any type.

Lists should support any type of element

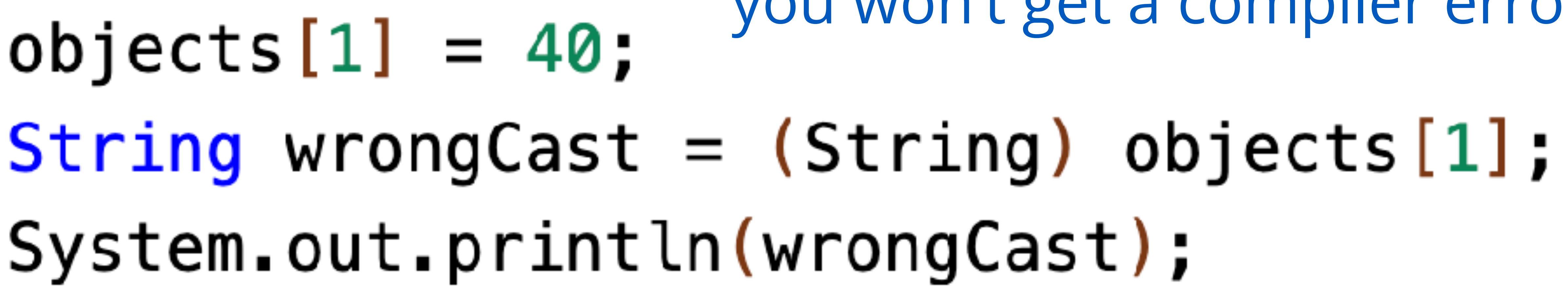
- We want our data structure to support any type of elements, as long as they are of the same type. We could use the class Object but this requires casting to the desired type:



```java
Object[] objects = new Object[2];
objects[0] = "hello";
String message = (String) objects[0];
System.out.println(message);
```

casting objects[0] to String, since it's type Object

but we might accidentally mix types! that's not OK! but you won't get a compiler error



```java
objects[1] = 40;
String wrongCast = (String) objects[1];
System.out.println(wrongCast);
```

results in runtime error: ClassCastException

Why generics help

- Generics are type parameters which are, well, generic.
- Let's say we want to create an interface that defines a new List data structure, but we don't know yet what type of object should be in the List. That's where we can us a *generic type*.

```
public interface List <E> {
    void add(E element);
    void add(int index, E element);
    E get(int index);
    boolean isEmpty();
}
```

We use <> to denote generics in an interface or class declaration

- Benefits:
  - Type safety (can't mix types in a list anymore)
  - No explicit casting needed anymore
  - Errors are caught at compile time instead of run time

Generics

```
public interface List <E> {
    void add(E element);
    void add(int index, E element);
    void clear();
    E get(int index);
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
    int size();
}
```

Formal type parameters

Kinds of formal type parameters:

E: element (common in data structures), T: type, K: key, V: value, N: number.

```
public class MyList<E> implements List<E>{…}
```

- In the invocation, all occurrences of the formal type parameters are replaced by the actual type argument

-
  ```
  MyList<String> list = new MyList<String>();
  list.add("hello");
  String s = list.get(0);   // no cast
  ```

ArrayLists

Limitations of Arrays

- Fixed-size.

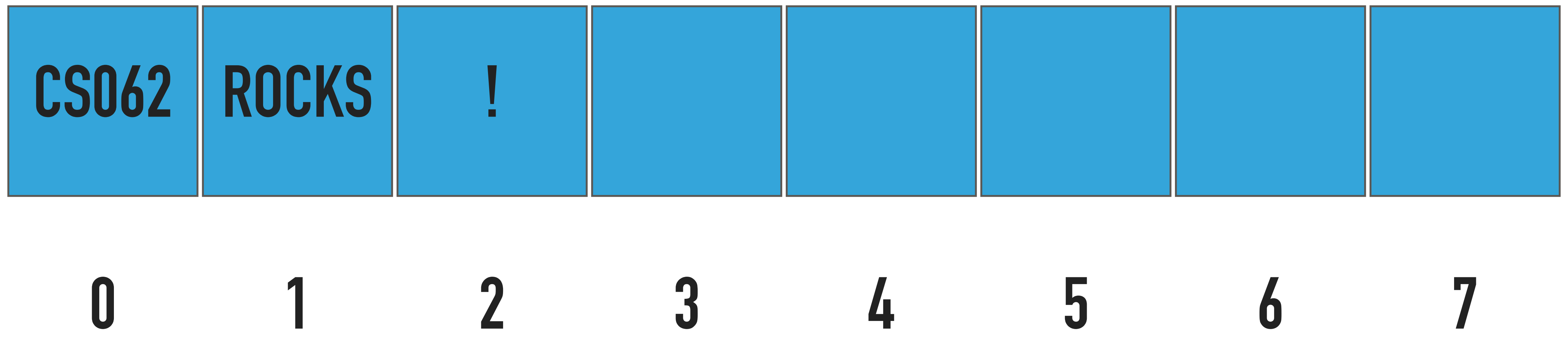
- Do not work well with generics.
  - `E[] myArray = (E[]) new Object[capacity];`

- Limited functionality (Java requires the use of Arrays class for printing contents and manipulating arrays, such as sorting and searching).

- We want resizable arrays that support any type of object.

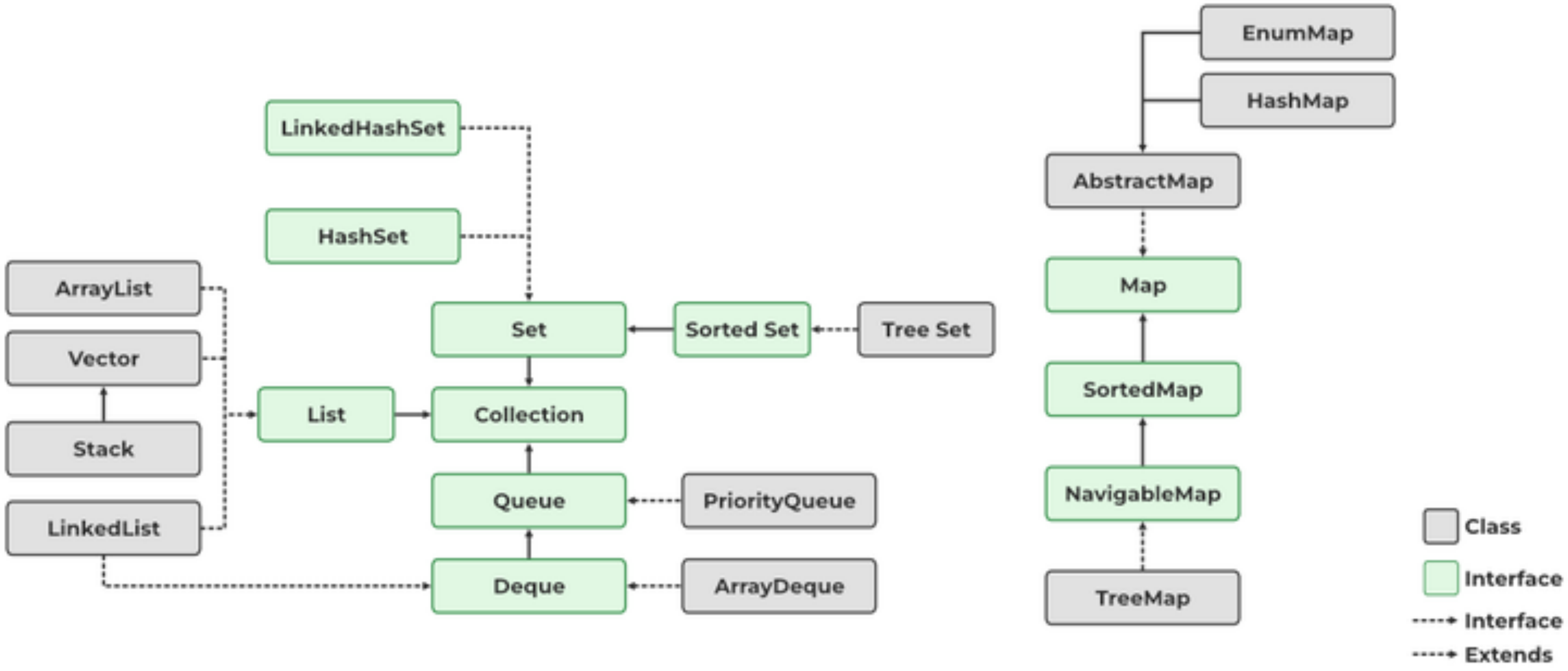
ArrayList (or dynamic/growable/resizable/mutable array)

- Dynamic linear data structure that is zero-indexed.
  - We will use the `List` interface to build it next class. This class, we’ll use a pre-built version that we can import.
- Sequential data structure that requires consecutive memory cells.
- Implemented with an underlying array of a specific capacity.
  - But the user does not see that!



CS062	ROCKS	!					
0	1	2	3	4	5	6	7

The Java Collections Framework

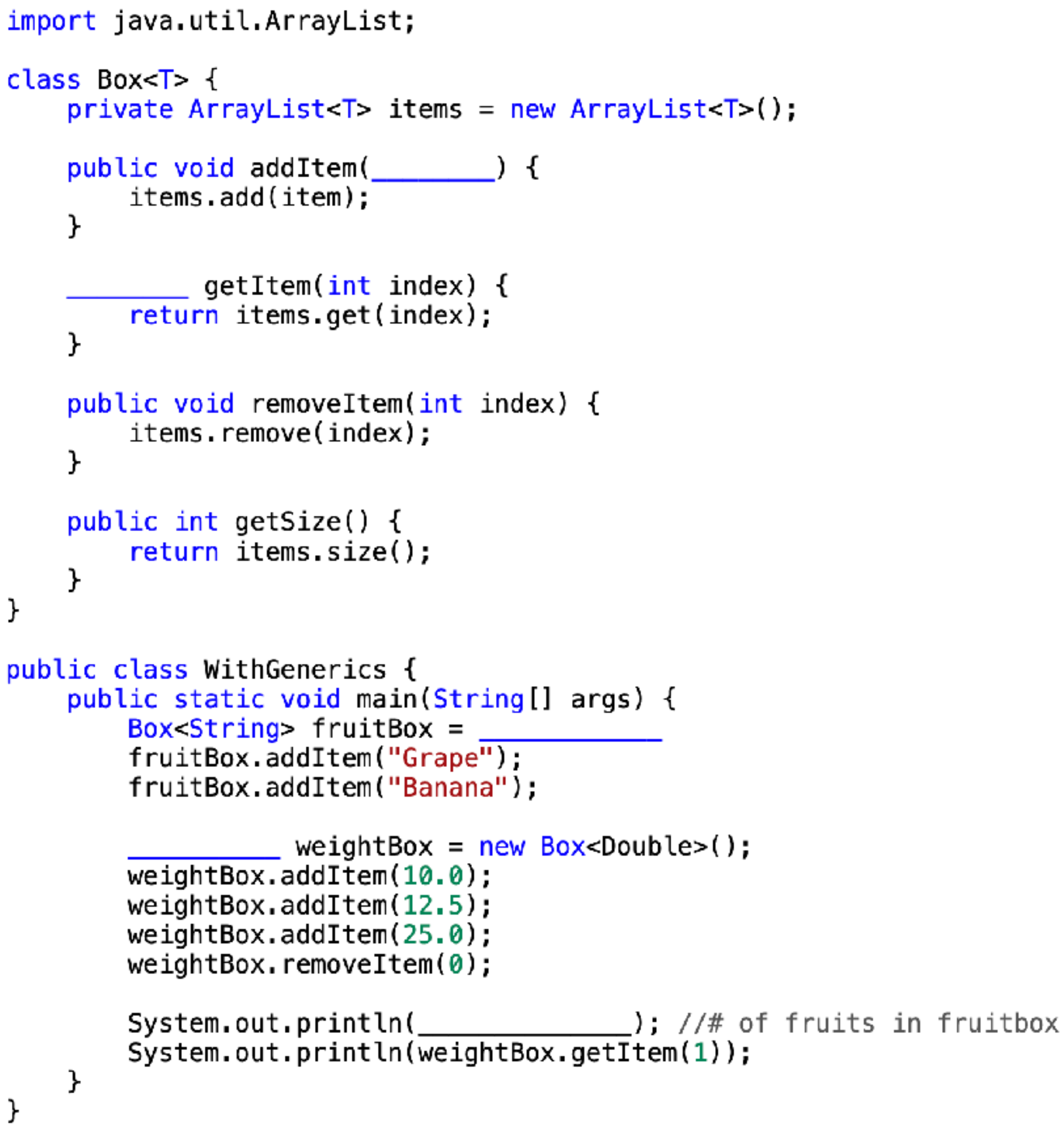


- Built in data structure classes that you can use in your code
- We'll be using it in this week's lab to practice *using* ArrayLists. Then, in Thursday's lecture, we'll practice writing our own ArrayList implementation.

```java
import java.util.ArrayList;
```

https://www.geeksforgeeks.org/collections-in-java-2/

Worksheet time!



```java
import java.util.ArrayList;

class Box<T> {
    private ArrayList<T> items = new ArrayList<T>();

    public void addItem(________) {
        items.add(item);
    }

    ________ getItem(int index) {
        return items.get(index);
    }

    public void removeItem(int index) {
        items.remove(index);
    }

    public int getSize() {
        return items.size();
    }
}

public class WithGenerics {
    public static void main(String[] args) {
        Box<String> fruitBox = ____________
        fruitBox.addItem("Grape");
        fruitBox.addItem("Banana");

        __________ weightBox = new Box<Double>();
        weightBox.addItem(10.0);
        weightBox.addItem(12.5);
        weightBox.addItem(25.0);
        weightBox.removeItem(0);

        System.out.println(______________); //# of fruits in fruitbox
        System.out.println(weightBox.getItem(1));
    }
}
```

1. Fill in the 5 blanks.

2. What gets printed?

3. If you called fruitBox.addItem(47), what would happen and why?

4. How do generics ensure type safety in this example?

Worksheet answers

```
import java.util.ArrayList;

class Box<T> {
    private ArrayList<T> items = new ArrayList<T>();

    public void addItem(T item) {
        items.add(item);
    }

  public T getItem(int index) {
        return items.get(index);
    }

    public void removeItem(int index) {
        items.remove(index);
    }

    public int getSize() {
        return items.size();
    }
}

public class WithGenerics {
    public static void main(String[] args) {
        Box<String> fruitBox = new Box<>(); // Box<String>() is OK too
        fruitBox.addItem("Grape");
        fruitBox.addItem("Banana");

     Box<double> weightBox = new Box<Double>();
        weightBox.addItem(10.0);
        weightBox.addItem(12.5);
        weightBox.addItem(25.0);
        weightBox.removeItem(0);

        System.out.println(fruitBox.getSize()); //# of fruits in fruitbox
        System.out.println(weightBox.getItem(1));
    }
}
```

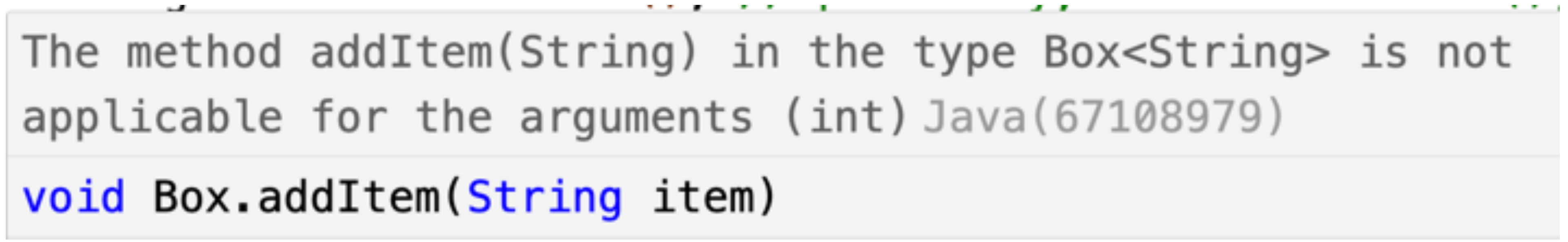
1. Fill in the 5 blanks.

2. What gets printed?

```
2
25.0
```

3. If you called fruitBox.addItem(47), what would happen and why?

Compiler error because types are mismatched
(can't add int to a String box)



```
The method addItem(String) in the type Box<String> is not
applicable for the arguments (int) Java(67108979)

void Box.addItem(String item)
```

4. How do generics ensure type safety in this example?

They are flexible enough to be any type, but they enforce that every item that's added to the ArrayList as to be the same type

Lecture 5 wrap-up

- HW2 due tonight 11:59pm
  - HW3 Darwin is already released. Pair programming, so find a partner (remember, we want you to code together in person, not I do this part, you do that part)
- Last retake for the quiz is tomorrow 4-5pm (remember, your lowest score is dropped)

Resources

- Interfaces: https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html
- Generics: https://docs.oracle.com/javase/tutorial/java/generics/index.html https://docs.oracle.com/javase/tutorial/extra/generics/intro.html
- Textbook: https://algs4.cs.princeton.edu/home/