

thefuture.build

Thursdays, 6:30 - 8:00 pm @ 105 North Gate Hall



Project 2

Some words of warning:

- Project 2 is going to be very hard.
- ... but you'll have a teammate to help.

Key resources:

- Lab 5 ([Link](#))
- Partnerships for project 2 ([Link](#))
- Getting started video for project 2 ([Link](#))
- Project 2 spec ([Link](#))

CS61B: 2018

Lecture 13: Generics, Conversion, Promotion

- Generic Basics, Autoboxing, Widening
- Immutability
- Generic Methods



Coming up Next: The Syntax Lectures

In the next three lectures, we'll build an Array based implementation of a Map, and along the way, learn some new syntax.

- Syntax1: Autoboxing, promotion, immutability, generics
- Syntax2: Exceptions, Iterables/Iterators
- Syntax3: Access control, equals, other loose ends
- Syntax4 (optional): Wildcards, type upper bounds, covariance (not in the scope of the class).

After that, we're done with Java language stuff.

Generics

For the most part, using generics is pretty straightforward.

- Generic classes require us to provide one or more ***actual type arguments***.

```
import java.util.ArrayList;

public class BasicArrayList {
    public static void main(String[] args) {
        ArrayList<String> L = new ArrayList<String>();
        L.add("potato");
        L.add("ketchup");
        String first = L.get(0);
    }
}
```

actual type argument: String.



In Java 8: No longer necessary at instantiation if also declaring a variable at the same time.

Primitives Cannot Be Used as Actual Type Arguments

We cannot use primitive types as actual type arguments.

- Code below causes a compile time error.

```
import java.util.ArrayList;

public class BasicArrayList {
    public static void main(String[] args) {
        ArrayList<int> L = new ArrayList<int>();
        L.add(5);
        L.add(6);
        int first = L.get(0);
    }
}
```

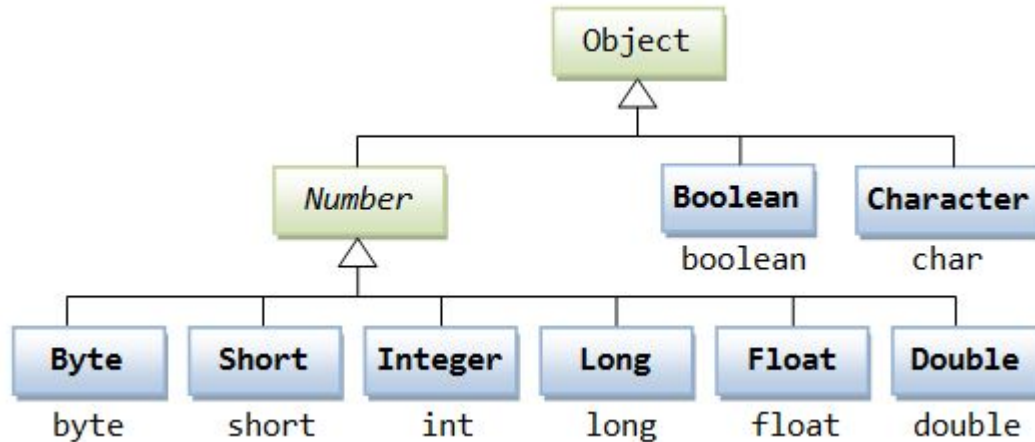
```
jug ~/temp
$ javac BasicArrayList.java
BasicArrayList.java:5: error: unexpected type
        ArrayList<int> L = new ArrayList<int>();
                        ^
required: reference
found:    int
```

Reference Types

Reminder: Java has 8 primitive types. All other types are reference types.

For each primitive type, there is a corresponding reference type called a wrapper class.

- For example, boolean's wrapper class is Boolean.



Reference Types as Actual Type Arguments

Solution: Use wrapper type as actual type parameter instead of primitive type.

```
import java.util.ArrayList;

public class BasicArrayList {
    public static void main(String[] args) {
        ArrayList<Integer> L = new ArrayList<Integer>();
        L.add(new Integer(5));
        L.add(new Integer(6));
        int first = L.get(0).valueOf();
    }
}
```

Conversion between int and Integer is annoying, so in Java 1.5 they also introduced...

Autoboxing

Autoboxing (auto-unboxing): Implicit conversions between wrapper/primitives.

```
import java.util.ArrayList;

public class BasicArrayList {
    public static void main(String[] args) {
        ArrayList<Integer> L = new ArrayList<Integer>();
        L.add(5);
        L.add(6);
        int first = L.get(0);
    }
}
```

Code above works even though we're passing an int into an Integer parameter, and assigning a return value of type Integer to an int.

Autoboxing and Unboxing

Wrapper types and primitives can be used almost interchangeably.

- If Java code expects a wrapper type and gets a primitive, it is autoboxed.

```
public static void blah(Integer x) {  
    System.out.println(x);  
}
```

```
int x = 20;  
blah(x);
```

- If the code expects a primitive and gets a wrapper, it is unboxed.

```
public static void blahPrimitive(int x) {  
    System.out.println(x);  
}
```

```
Integer x = new Integer(20);  
blahPrimitive(x);
```

Some notes:

- Arrays are never autoboxed/unboxed, e.g. an `Integer[]` cannot be used in place of an `int[]` (or vice versa).
- Autoboxing / unboxing incurs a measurable performance impact!
- Wrapper types use MUCH more memory than primitive types.

Wrapper Types Are (Mostly) Just Like Any Class

You can read the source code to all built-in Java libraries.

- e.g. google “grepcode java Integer” yields [this link](#).
- Integer has no magic powers except autoboxing/auto-unboxing.

```
public final class Integer
    extends Number implements Comparable<Integer> {

    private final int value;

    public Integer(int value) {
        this.value = value;
    }
    ...
}
```

Wrapper Type Memory: <http://shoutkey.com/appear>

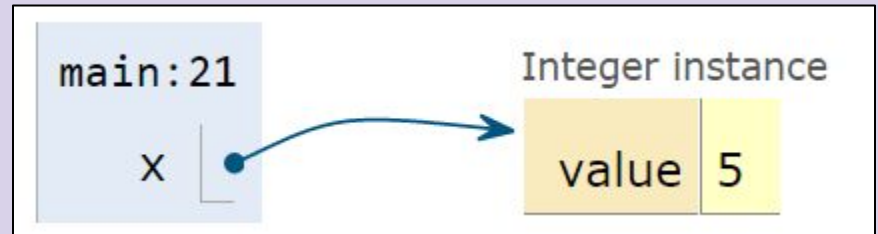
Assuming:

- Addresses are 64 bits.
- ints are 32 bits.
- All Java objects take 64 bits + their fields.

```
public static void bleepblorp() {  
    Integer x = new Integer(5);  
    System.out.println(x);  
}
```

How much total memory is used by bleepblorp to store its local variables?

- 32 bits.
- 64 bits.
- 96 bits.
- 128 bits.
- 160 bits.



Wrapper Type Memory: <http://shoutkey.com/appear>

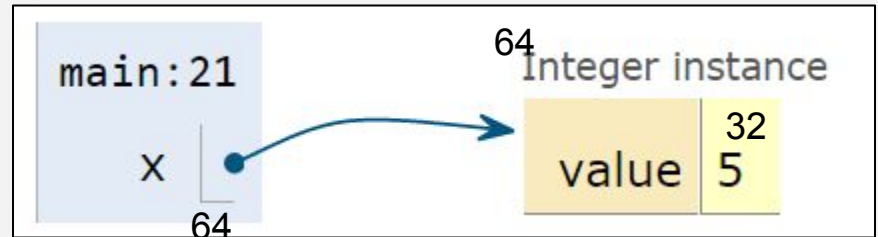
Assuming:

- Addresses are 64 bits.
- ints are 32 bits.
- All Java objects take 64 bits + their fields.

```
public static void bleepblorp() {  
    Integer x = new Integer(5);  
    System.out.println(x);  
}
```

How much total memory is used by bleepblorp to store its local variables?

- 32 bits.
- 64 bits.
- 96 bits.
- 128 bits.
- 160 bits: 64 + 96 for object**



Another Type of Conversion: Primitive Widening

A similar thing happens when moving from a primitive type with a narrower range to a wider range.

- In this case, we say the value is “widened”.
- Code below is fine since double is wider than int.

```
public static void blahDouble(double x) {  
    System.out.println("double: " + x);  
}
```

```
int x = 20;  
blahDouble(x);
```

To move from a wider type to a narrower type, must use casting:

```
public static void blahInt(int x) {  
    System.out.println("int: " + x);  
}
```

```
double x = 20;  
blahInt((int) x);
```

Full details here: <http://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html>

Immutability

Immutable Data Types

An immutable data type is one for which an instance cannot change in any observable way after instantiation.

Examples:

- Mutable: ArrayDeque, Planet.
- Immutable: Integer, String, Date.

```
public class Date {  
    public final int month;  
    public final int day;  
    public final int year;  
    private boolean contrived = true;  
    public Date(int m, int d, int y) {  
        month = m; day = d; year = y;  
    }  
}
```

The ***final*** keyword will help the compiler ensure immutability.

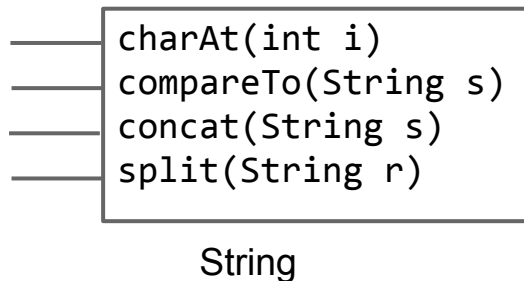
- final variable means you will assign a value once (either in constructor of class or in initializer).
- Not necessary to have final to be immutable (e.g. Dog with private variables).

Immutability

Advantage: Less to think about: Avoids bugs and makes debugging easier.

- Analogy: Immutable classes have some buttons you can press / windows you can look inside. Results are ALWAYS the same, no matter what.

Disadvantage: Must create a new object anytime anything changes.



Warning: Declaring a reference as **Final** does not make object immutable.

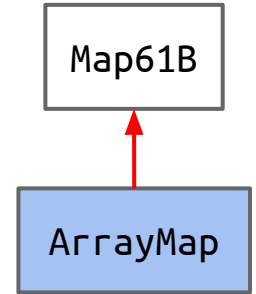
- Example: `public final ArrayDeque<String> d = new ArrayDeque<String>();`
- The d variable can never change, but the referenced deque can!

Defining Generic Classes

Goals

Goal 1: Create a class `ArrayMap` with the following methods:

- `put(key, value)`: Associate key with value.
- `containsKey(key)`: Checks to see if arraymap contains the key.
- `get(key)`: Returns value, assuming key exists..
- `keys()`: Returns a list of all keys.
- `size()`: Returns number of keys.



Ok to ignore resizing for this exercise.

- In lecture, I'll just show the answer, but you might find implementing it useful. See study guide for this lecture for starter code.

ArrayMap (Basic Implementation)

```
public class ArrayMap<K, V> {  
    private K[] keys;  
    private V[] values;  
    private int size;  
    public ArrayMap() {  
        keys = (K[]) new Object[100];  
        values = (V[]) new Object[100];  
        size = 0;  
    }  
    ...  
}
```

Array implementation of a Map:

- Use an array as the core data structure.
- `put(k, v)`: Finds the array index of `k`
 - If -1, adds `k` and `v` to the last position of the arrays.
 - If non-negative, sets the appropriate item in values array.

ArrayMap (Basic Implementation)

```
public void put(K key, V value) {
    int i = getKeyIndex(key);
    if (i > -1) {
        values[i] = value; return; }
    keys[size] = key;
    values[size] = value;
    size += 1;
}

public V get(K key) {
    return values[findKey(key)];
}
```

```
public boolean
    containsKey(K key) {
    int i = findKey(key);
    return (i > -1);
}

public List<Keys> keys() {
    ... /* See code */
}
```

Using An ArrayMap

Generic type variables

```
public class ArrayMap<K, V> {  
    private K[] keys;  
    private V[] values;  
    private int size;  
    public ArrayMap() {  
        keys = (K[]) new Object[100];  
        values = (V[]) new Object[100];  
        size = 0;  
    }  
    ...  
}
```

Actual type arguments

```
ArrayMap<Integer, String> ismap = new ArrayMap<Integer, String>();  
ismap.put(5, "hello");  
ismap.put(10, "ketchup");
```

A Mysterious Error Appears

```
@Test
public void test() {
    ArrayMap<Integer, Integer> am = new ArrayMap<Integer, Integer>();
    am.put(2, 5);
    int expected = 5;
    assertEquals(expected, am.get(2));
}
```

```
$ javac ArrayMapTest.java
```

```
ArrayMapTest.java:11: error: reference to assertEquals is ambiguous
    assertEquals(expected, am.get(2));
    ^
```

```
    both method assertEquals(long,long) in Assert and method
    assertEquals(Object,Object) in Assert match
```

The Issue:

- JUnit has many assertEquals functions including (int, int), (double, double), (Object, Object), etc.

Which automatic conversions are needed to call `assertEquals(long, long)`?

- A. Widen expected to long.
- B. Autobox expected as a Long.
- C. Autobox expected as an Long.
- D. Unbox `am.get(2)`.
- E. Widen the unboxed `am.get(2)` to long.

There may be more than one right answer.

Hint, the actual call is: `assertEquals(int, Integer)`

```
@Test
public void test() {
    ArrayMap<Integer, Integer> am = new ArrayMap<Integer, Integer>();
    am.put(2, 5);
    int expected = 5;
    assertEquals(expected, am.get(2));
}
```


Which automatic conversions are needed to call `assertEquals(long, long)`?

- A. **Widen expected to long.**
- B. Autobox expected as a Long.
- C. Autobox expected as an Long.
- D. **Unbox `am.get(2)`.**
- E. **Widen the unboxed `am.get(2)` to long.**

There may be more than one right answer.

Hint, the actual call is: `assertEquals(int, Integer)`

```
@Test
public void test() {
    ArrayMap<Integer, Integer> am = new ArrayMap<Integer, Integer>();
    am.put(2, 5);
    int expected = 5;
    assertEquals(expected, am.get(2));
}
```

Open Question

What automatic conversions are needed to call `assertEquals(Object, Object)`?

```
@Test
public void test() {
    ArrayMap<Integer, Integer> am = new ArrayMap<Integer, Integer>();
    am.put(2, 5);
    int expected = 5;
    assertEquals(expected, am.get(2)); }
```

Open Question

What automatic conversions are needed to call `assertEquals(Object, Object)`?

```
@Test
public void test() {
    ArrayMap<Integer, Integer> am = new ArrayMap<Integer, Integer>();
    am.put(2, 5);
    int expected = 5;
    assertEquals(expected, am.get(2)); }
```

Only one conversion needed (unless you count [Integer → Object](#))

- Autobox “expected” into an Integer.

Even though this is ‘easier’ than the 3-step process needed to get to `assertEquals(long, long)`, it’s still ambiguous and thus Java won’t let the code above compile.

Open Question

How do we get the code to compile, e.g. how do we resolve the ambiguity?

```
@Test
public void test() {
    ArrayMap<Integer, Integer> am = new ArrayMap<Integer, Integer>();
    am.put(2, 5);
    int expected = 5;
    assertEquals(expected, am.get(2)); }
```

Open Question

How do we get the code to compile, e.g. how do we resolve the ambiguity?

```
@Test
public void test() {
    ArrayMap<Integer, Integer> am = new ArrayMap<Integer, Integer>();
    am.put(2, 5);
    int expected = 5;
    assertEquals(expected, am.get(2)); }
```

Many possible answers, one of them is:

```
@Test
public void test() {
    ArrayMap<Integer, Integer> am = new ArrayMap<Integer, Integer>();
    am.put(2, 5);
    int expected = 5;
    assertEquals((Integer) expected, am.get(2)); }
```

Generic Methods

Goals

Goal: Create a class MapHelper with two methods:

- `get(Map61B, key)`: Returns the value corresponding to the given key in the map if it exists, otherwise null.
 - Unlike the `ArrayMap`'s `get` method, which crashes if the key doesn't exist.
- `maxKey(Map61B)`: Returns the maximum of all keys in the given `ArrayMap`. Works only if keys can be compared.

Goals

Goal: Create a class MapHelper with two methods:

- `get(key)`: Returns the item in the map if it exists, otherwise null.
- `maxKey()`: Returns the maximum of all keys. Works only if keys can be compared.

MapHelper.java



Generic Methods

Can create a method that operates on generic types by defining type parameters *before the return type* of the method:

Formal type parameter definitions.

```
public static <X, Zerp> Zerp get(ArrayMap<X, Zerp> am, X key) {  
    if (am.containsKey(key)) {  
        return am.get(key);  
    }  
    return null;  
}
```

Return type: Zerp (whatever that is)

In almost all circumstances, using a generic method requires no special syntax:

```
ArrayMap<Integer, String> ismap =  
    new ArrayMap<Integer, String>();  
System.out.println(MapHelper.get(ismap, 5));
```

It's that easy.

Goals

Goal: Create a class MapHelper with two methods:

- `get(key)`: Returns the item in the map if it exists, otherwise null.
- `maxKey()`: Returns the maximum of all keys. Works only if keys can be compared.

MapHelper.java



The Issue with Generic Methods

We had the code below with a major problem: Cannot compare Ks using >.

- Only numerical primitives can be compared with >.

... though due to auto-unboxing: numerical wrapper types can be compared with >

```
public static <K, V> K maxKey(ArrayMap<K, V> map) {  
    List<K> keylist = map.keys();  
    K largest = keylist.get(0);  
    for (K k : keylist) {  
        if (k > largest) {  
            largest = k;  
        }  
    }  
    return largest;  
}
```

The Issue with Generic Methods

New problem: K's don't necessarily have a compareTo method.

```
public static <K, V> K maxKey(ArrayMap<K, V> map) {  
    List<K> keylist = map.keys();  
    K largest = keylist.get(0);  
    for (K k : keylist) {  
        if (k.compareTo(largest) > 0) {  
            largest = k;  
        }  
    }  
    return largest;  
}
```

Issue with The compareTo Approach

```
public static <K, V> K maxKey(HashMap<K, V> map) {  
    List<K> keylist = map.keySet();  
    K largest = keylist.get(0);  
    for (K k : keylist) {  
        if (k.compareTo(largest) > 0) {  
            largest = k;  
        }  
    }  
    return largest;  
}
```

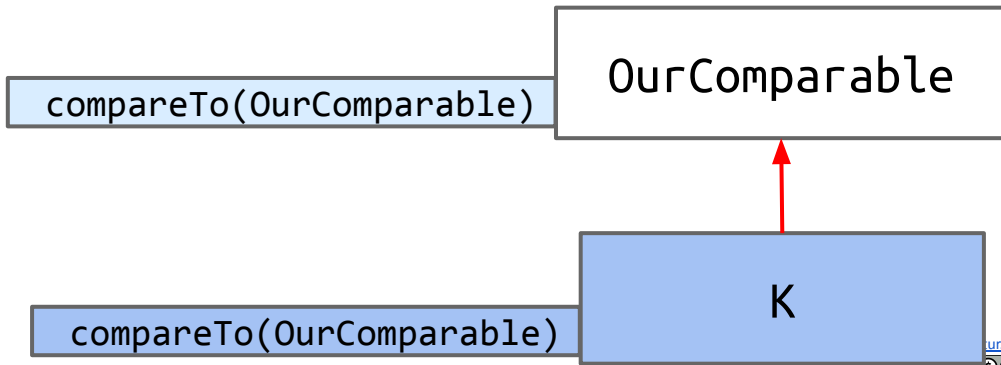
```
$ javac MapHelper.java  
MapHelper.java:14: error: cannot find symbol  
        int cmp = k.compareTo(largest);  
                           ^  
symbol:   method compareTo(K)  
location: class Object
```

Type Upper Bounds to The Rescue

Can use extends keyword as a **type upper bound**. Only allow use on ArrayMaps with OurComparable keys.

Meaning: Any ArrayMap you give me must have actual parameter type that is a subtype of OurComparable.

```
public static <K extends OurComparable, V> K maxKey(ArrayMap<K, V> am) {  
    ...  
    if (k.compareTo(largest) > 0) {  
    ...  
}
```



Note: Type lower bounds also exist, specified using the word super. Won't cover in 61B.

A Better Type Upper Bound: Comparable

Can use extends keyword as a **type upper bound**. Only allow use on ArrayMaps with Comparable keys.

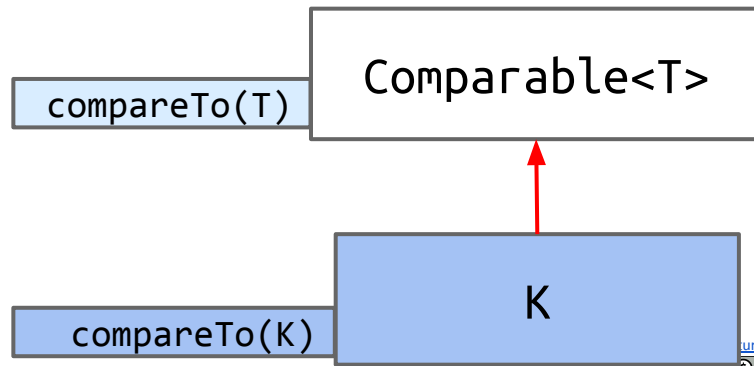
Meaning: Any ArrayMap you give me must have actual parameter type that is a subtype of Comparable<T>.

```
public static <K extends Comparable<K>, V> K maxKey(ArrayMap<K, V> am) {  
    ...  
    if (k.compareTo(largest) > 0) {  
    ...  
}
```

Built in Java interface: Comparable<T>

- Implemented by Integer, String, etc.

Note: Type lower bounds also exist, specified using the word super. Won't cover in 61B.



Generics Summary

We've now seen four new features of Java that make Generics more powerful:

- Autoboxing and auto-unboxing of primitive wrapper types.
- Promotion between primitive types.
- Specification of generic types for methods (before return type).
- Type upper bounds (e.g. `K` extends `Comparable<K>`)
- In `syntax4`, you can also see another feature called “wildcards”.

A true understand of Java generics takes a long time and lots of practice.

- You won't know all the details by the end of 61B.
- I promise not to ask questions about bounded wildcards, type erasure, or covariance (see bonus lecture entitled `syntax4`).

And yet there's still more, e.g. `public static <T> int binarySearch(T[] a, T key, Comparator<? super T> c) {`

Syntax4: Optional Lecture (coming later, but slides now for the curious)

A Quick Dip into Generic Hell

Second, we started building a MapHelper class hoping to provide the following:

- `get(key)`: Returns the item in the map if it exists.
- `maxKey()`: Returns the maximum of all keys. Works only if keys can be compared.
- `allBark()`: Makes all keys bark. Works only for keys of type Dog.

```
ArrayMap<Dog, Double> am2 = new ArrayMap<Dog, Double>();  
am2.put(new Dog("frank"), 10);  
am2.put(new FrenchDog("francis", 20);  
MapHelper.allBark(am2);
```

MapHelper.java




Problem #1: Dealing with Types We Don't Care about

Implementation below works, but only for ArrayMaps from Dog to Double.

```
public static void allBark(ArrayMap<Dog, Double> am) {  
    List<Dog> dogs = am.keys();  
    for (int i = 0; i < dogs.size(); i += 1) {  
        dogs.get(i).bark();  
    }  
}
```

```
ArrayMap<Dog, Integer> am2 = new ArrayMap<Dog, Integer>();  
am2.put(new Dog("frank"), 10);  
am2.put(new FrenchDog("francis"), 20);  
MapHelper.allBark(am2);
```

Value types mismatch!



```
$ javac MapHelper.java  
MapHelper.java:62: error: incompatible types: ArrayMap<Dog,Integer>  
cannot be converted to ArrayMap<Dog,Double>
```

Problem #1: Dealing with Types We Don't Care about

How could we fix the allBark method so that it works for any value type?

```
public static void allBark(HashMap<Dog, Double> am) {  
    List<Dog> dogs = am.keySet();  
    for (int i = 0; i < dogs.size(); i += 1) {  
        dogs.get(i).bark();  
    }  
}
```

```
HashMap<Dog, Integer> am2 = new HashMap<Dog, Integer>();  
am2.put(new Dog("frank"), 10);  
am2.put(new FrenchDog("francis"), 20);  
MapHelper.allBark(am2);
```

Value types mismatch!

```
$ javac MapHelper.java  
MapHelper.java:62: error: incompatible types: HashMap<Dog,Integer>  
cannot be converted to HashMap<Dog,Double>
```

Fix #1

Can add generic parameter to method to fix.

```
public static <V> void allBark(ArrayMap<Dog, V> am) {  
    List<Dog> dogs = am.keys();  
    for (int i = 0; i < dogs.size(); i += 1) {  
        dogs.get(i).bark();  
    }  
}
```

Alternate Fix #1

Fix #1 and Alternate Fix #1 are both perfectly acceptable!

Alternately: Use Wildcard character: ?

```
public static void allBark(HashMap<Dog, ?> am) {  
    List<Dog> dogs = am.keySet();  
    for (int i = 0; i < dogs.size(); i += 1) {  
        dogs.get(i).bark();  
    }  
}
```

Basic idea:

- We don't care about the actual type, since we never used V anywhere.
- This is a fairly advanced feature you're unlikely to use in 61B. Will only appear on a midterm or final if it ends up being on a HW/lab/project.

Quick Aside: Code Optimization

Lists in Java support for-each loop, sometimes called enhanced for loop.

```
public static void allBark(HashMap<Dog, ?> am) {  
    List<Dog> dogs = am.keys();  
    for (int i = 0; i < dogs.size(); i += 1) {  
        dogs.get(i).bark();  
    }  
}
```

Same
output.

```
public static void allBark(HashMap<Dog, ?> am) {  
    for (Dog d : am.keys()) {  
        d.bark();  
    }  
}
```

Avoids need to iterate through
list using indices.

Problem #2: Covariance

Surprisingly, cannot pass an ArrayMap of FrenchDog keys!

```
public static void allBark(ArrayMap<Dog, ?> am) {  
    for (Dog d : am.keys()) {  
        d.bark();  
    }  
}
```

```
ArrayMap<FrenchDog, Integer> am2 = new ArrayMap<FrenchDog, Integer>();  
am2.put(new FrenchDog("francis"), 10);  
am2.put(new FrenchDog("francis jr"), 20);  
allBark(am2);
```

```
$ javac MapHelper.java
```

```
MapHelper.java:62: error: incompatible types:
```

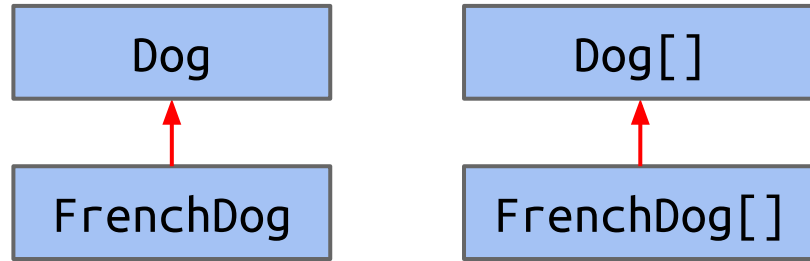
```
ArrayMap<FrenchDog,Integer> cannot be converted to ArrayMap<Dog,?>
```


Covariance

Arrays are **covariant** in Java, but generic types are invariant.

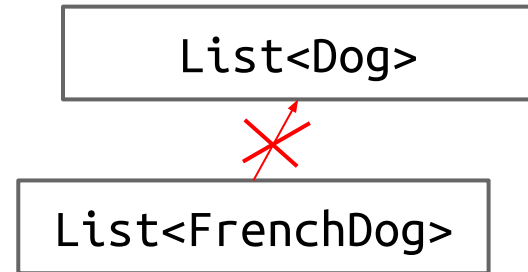
Arrays are covariant:

- A FrenchDog is-a Dog.
- An FrenchDog[] is-a Dog[].



Generic types are invariant:

- A List<FrenchDog> is NOT a List<Dog>.



This maddening feature is my least favorite part of Java.

- Leads to lots of syntactical contortions. See next slide.
- Why did Java designers do this to us? See extra slides.

Fixing Problem #2

Two equivalent fixes:

- Approach 1: Add a generic type to our method.
- Approach 2: Add a bounded-wildcard (unlikely to use in 61B).

```
public static <K extends Dog> void allBark(ArrayMap<K, ?> am) {  
    for (Dog d : am.keys()) {  
        d.bark();  
    }  
}
```

```
public static void allBark(ArrayMap<? extends Dog, ?> am) {  
    for (Dog d : am.keys()) {  
        d.bark();  
    }  
}
```

Code never uses K so need to actually specify a generic type.

Generics Summary

We've now seen four new features of Java that make Generics more powerful:

- Autoboxing and auto-unboxing of primitive wrapper types.
- Promotion between primitive types.
- Specification of generic types for methods (before return type).
- Type upper bounds (e.g. `K extends Comparable<K>`)
- Wildcards: ?

A true understand of Java generics takes a long time and lots of practice.

- You won't know all the details by the end of 61B.
- I promise not to ask questions about bounded wildcards or type erasure (see extra slides).

And yet there's still more, e.g. `public static <T> int binarySearch(T[] a, T key, Comparator<? super T> c) {`

Citations

Drink:

http://hilanddairy.com/image-library/sites/default/files/styles/large/public/Hiland_GrapeDrink_Gal.jpg

Wrapper class image from Nanyang Technological University:

https://www3.ntu.edu.sg/home/ehchua/programming/java/images/OOP_WrapperClass.png