BASIC JAVA NOTES

Yingjie Xu

REFERENCE: CS61B / COMP 250

Table of Contents

Inheritance (with implements keyword) and Subtype Polymorphism

Method Overloading:

Java allows multiple methods with same name, but different parameters.

```
public static String longest(AList<String> list) {
    ...
}

public static String longest(SLList<String> list) {
    ...
}
```

Interface:

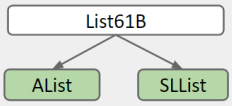
Step1: define an **interface**

Interface is a specification of what a List can do, not how to do it.

```
public interface List61B<Item> {
   public void addFirst(Item x);
   public void addLast(Item y);
   public Item getFirst();
   public Item getLast();
   public Item removeLast();
   public Item get(int i);
   public void insert(Item x, int position);
   public int  size();
}
```

Step2: Implements the interface with **implements** keyword (override the method with the @Override tag)

```
public class AList<Item> implements List61B<Item>{
   ...

   @Override
   public void addLast(Item x) {
      ...
```



Method Overriding:

If a "**subclass**" has a method with the exact same signature as in the "**superclass**", we say the subclass **overrides** the method.

@Override tag:

we'll always mark every overriding method with the **@Override** annotation. The only effect of this tag is that the code won't compile if it is not actually an overriding method.

If a subclass has a method with the exact same signature as in the superclass, we say the subclass overrides the method.

- Even if you don't write @Override, subclass still overrides the method.
- @Override is just an **optional reminder** that you're overriding.

Method Overriding vs. Overloading

- `Animal`'s subclass `Pig` overrides the `makeNoise()` method.
- Methods with the same name but different signatures are **overloaded**.

```
public interface Animal {
   public void makeNoise();
}
```

```
public class Pig implements Animal {
  public void makeNoise() {
    System.out.print("oink");
   }
}
```

`Pig` overrides `makeNoise()`

```
public class Dog implements Animal {
   public void makeNoise(Dog x) {
      ...
```

`makeNoise` is overloaded

```
public class Math {
   public int    abs(int a)
   public double abs(double a)
```

`abs` is overloaded

Interface Inheritance (implements)

Specifying the capabilities of a subclass using the **implements** keyword is known as **interface inheritance.**

- **Interface**: The list of all method signatures.
- **Inheritance**: The subclass "inherits" the interface from a superclass.
- Specifies what the subclass can do, but not how.
- Subclasses **must** override **all** these methods!
    - Will fail to compile otherwise.

Copying the Bits (e.g. storing ArrayList inside List)

If X is a superclass of Y, then memory boxes for X may contain Y.

- An AList is-a List.
- Therefore, List variables can hold ALList addresses.

Implementation Inheritance: Default Methods

Interface inheritance:

- Subclass inherits **signatures**, but **NOT implementation**.

Subclasses can inherit **signatures AND implementation**.

- Use the **default** keyword to specify a method that subclasses should inherit from an interface.

We can Override default methods in each class.

Summary: (Interface inheritance vs. Implementation Inheritance)

Two types of inheritance:

- Interface inheritance (inherit signatures only)
- Implementation inheritance (inherit both signatures and implementation)

In both cases we could use @Override to override the original method

Interface Inheritance (a.k.a. what):

- Allows you to generalize code in a powerful, simple way.

Implementation Inheritance (a.k.a. how):

- Allows code-reuse: Subclasses can rely on superclass or interfaces.

Important: In both cases, we specify **"is-a" relationships**, not "has-a".

- Good: Dog implements Animal, SLList implements List61B.

- Bad: Cat implements Claw, Set implements SLList.

Subtype Polymorphism:

Polymorphism: “providing a single interface to entities of different types”

- When you call deque.addFirst(), the actual behavior is based on the dynamic type. (deque is an interface in java)
- Java automatically selects the right behavior using what is sometimes called “dynamic method selection”.

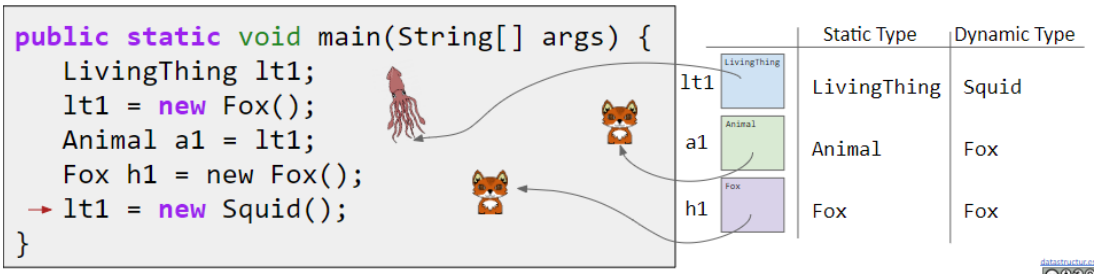
Static and Dynamic Type, Dynamic Method Selection and Type Casting

Every variable in Java has a **"compile-time type", a.k.a. "static type"**.

- This is the type specified at declaration. **Never changes**!

Variables also have a **"run-time type", a.k.a. "dynamic type"**.

- This is the type specified at **instantiation** (e.g. when using **new**).
- Equal to the type of the object being pointed at.



Dynamic Method Selection for Overridden Methods:

Suppose we call a method of an object using a variable with:

- compile-time type X
- run-time type Y

Then if Y overrides the method, Y's method is used instead.

- This is known as "dynamic method selection".

Dynamic Method Selection Puzzle

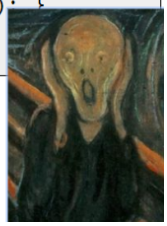
Suppose we have classes defined below. Try to predict the results.

```
public interface Animal {
  default void greet(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void flatter(Animal a)
    print("u r cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("dog sniff animal"); }
  void flatter(Dog a) {
    print("u r cool dog"); }
```

```
Animal a = new Dog();
Dog d = new Dog();
a.greet(d);   // "hello animal"
a.sniff(d);   // "dog sniff animal"
d.flatter(d); // "u r cool dog"
a.flatter(d); // "u r cool animal"
```

flatter is overloaded, not overridden!



If flatter is overridden, then the more specific method will be called. i.e. the method in the Dog class will be called.

```
public class Bird {
    public void gulgate(Bird b) {
        System.out.println("BiGulBi"); }}
```

```
public class Falcon extends Bird {
    public void gulgate(Falcon f) {
        System.out.println("FaGulFa");}}
```

```
Bird bird = new Falcon();
Falcon falcon = (Falcon) bird;
bird.gulgate(falcon);
```

Remember: The compiler chooses the most specific matching method signature from the static type of the invoking class.

- Falcon is overloading the gulgate method, not overriding.
- Compiler basically thinks "does Bird class have a gulgate method? Yes! I'll use that". Since there is no overloading, no dynamic method selection occurs.

datastructure.e

The rules for Dynamic Method Selection:

- Compiler allows **memory box** to hold any subtype.
- Compiler allows **calls** based on static type.
- **Overridden** non-static methods are selected at run time based on dynamic type.
- Everything else is based on static type, including **overloaded** methods.

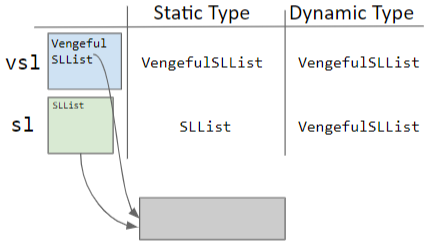
Type Checking:

Compile-Time Type Checking

Also called static type.

Compiler allows method calls based on **compile-time** type of variable.

- sl's runtime type: VengefulSLList.
- But cannot call printLostItems.




```
public static void main(String[] args) {
    VengefulSLList<Integer> vsl =
          new VengefulSLList<Integer>(9);
    SLList<Integer> sl = vsl;

    sl.addLast(50);
    sl.removeLast();

    sl.printLostItems();
    VengefulSLList<Integer> vsl2 = sl;
}
```

Compilation errors!

Compiler also allows assignments based on compile-time types.

- Even though sl's runtime-type is VengefulSLList, cannot assign to vsl2.
- Compiler plays it as safe as possible with type checking.

datastructures

Compile-Time Types and Expressions

Expressions have compile-time types:

- Method calls have compile-time type equal to their declared type.

```
public static Dog maxDog(Dog d1, Dog d2) { … }
```

- **Any call to maxDog will have compile-time type Dog!**

Example:

```
Poodle frank  = new Poodle("Frank", 5);
Poodle frankJr = new Poodle("Frank Jr.", 15);

Dog largerDog = maxDog(frank, frankJr);
Poodle largerPoodle = maxDog(frank, frankJr);
```

Compilation error!

RHS has compile-time type Dog.

Casting:

Java has a special syntax for specifying the compile-time type of any expression. Put desired type in parenthesis before the expression.

- Examples:
  - Compile-time type Dog: `maxDog(frank, frankJr);`
  - Compile-time type Poodle: `(Poodle) maxDog(frank, frankJr);`

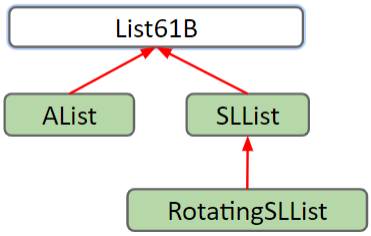
Casting is a powerful but dangerous tool. Tells Java to treat an expression as having a different compile-time type. **ClassCastException** is thrown if the run-time type is incorrect.

Inheritance (with extends keyword) and Encapsulation

Implementation Inheritance: Extends

We'd like to build RotatingSLList that can perform any SLList operation as well as:

- rotateRight(): Moves back item the front.



Example: Suppose we have [5, 9, 15, 22].

- After rotateRight: [22, 5, 9, 15]

Because of **extends**, RotatingSLList inherits all members of SLList:

- All instance and static variables.
- All methods.
- All nested classes.
- members may be **private** and thus inaccessible!

Constructors are not inherited.

Use extends for "is-a" relationship!

Super keyword:

```
public class VengefulSLList<Item> extends SLList<Item> {
    private SLList<Item> deletedItems;
    public VengefulSLList() {
       deletedItems = new SLList<Item>();
    }

    @Override
    public Item removeLast() {
        Item oldBack = super.removeLast();
        deletedItems.addLast(oldBack);
        return oldBack;
    }

    public void printLostItems() {
        deletedItems.print();
    }
}
```

calls Superclass's version of removeLast()

Inheritance and Constructor

Constructors are **not inherited**. However, the rules of Java say that all constructors must **start** with a **call to one of the super class's constructors**.

- You can explicitly call the constructor with the keyword super (no dot).
- If you don't explicitly call the constructor, Java will automatically do it for you.

```
public VengefulSLList() {
   deletedItems = new SLList<Item>();
}
```

These constructors are exactly equivalent.

```
public VengefulSLList() {
   super(); <-- must come first!
   deletedItems = new SLList<Item>();
}
```

Calling Other Constructors

If you want to use a super constructor other than the no-argument constructor, can give parameters to super.

```
public VengefulSLList(Item x) {
   super(x); <-- calls SLList(Item x)
   deletedItems = new SLList<Item>();
}
```

Not equivalent! Code to the right makes implicit call to super(), not super(x).

```
public VengefulSLList(Item x) {
   deletedItems = new SLList<Item>();
}
```

Object Class

As it happens, every type in Java is a descendant of the Object class.

Including methods: equals(), toString(), hashCode()

Modules and Encapsulation

Module: A set of methods that work together as a whole to perform some task or set of related tasks.

A module is said to be **encapsulated** if its implementation is **completely hidden**, and it can be accessed only through a documented interface.

- Instance variables private. Methods like resize private.

Even when writing tests, you don't (usually) want to peer inside.

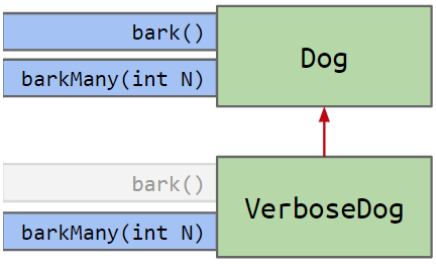
Implementation Inheritance breaks Encapsulation

What would vd.barkMany(3) output?

a. **As a dog, I say: bark bark bark**
b. bark bark bark
c. Something else.

(assuming vd is a Verbose Dog)

```
public void bark() {
   System.out.println("bark");
}
public void barkMany(int N) {
    for (int i = 0; i < N; i += 1) {
        bark();
    }
}
```




```
@Override
public void barkMany(int N) {
    System.out.println("As a dog, I say: ");
    for (int i = 0; i < N; i += 1) {
        bark(); ← calls inherited bark method
    }
}
```

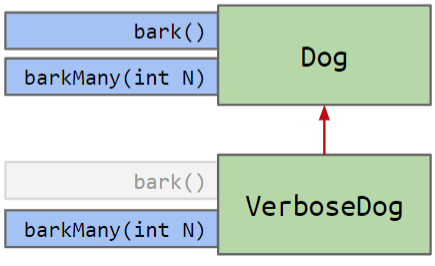
What would vd.barkMany(3) output?

c. **Something else.**

- Gets caught in an infinite loop!

(assuming vd is a Verbose Dog)

```
public void bark() {
    barkMany(1);
}
public void barkMany(int N) {
    for (int i = 0; i < N; i += 1) {
        System.out.println("bark");
    }
}
```




```
@Override
public void barkMany(int N) {
    System.out.println("As a dog, I say: ");
    for (int i = 0; i < N; i += 1) {
        bark(); ← calls inherited bark method
    }
}
```

Higher Order Functions:

Higher Order Function: A function that treats another function as data.

- e.g. takes a function as input.

Example: Higher Order Functions Using Interfaces in Java

```
public interface IntUnaryFunction {
    int apply(int x);
}
```

```
public class TenX implements IntUnaryFunction {
    public int apply(int x) {
        return 10 * x;
    }
}
```

```
public class HoFDemo {
    public static int do_twice(IntUnaryFunction f, int x) {
        return f.apply(f.apply(x));
    }
    public static void main(String[] args) {
        System.out.println(do_twice(new TenX(), 2));
    }
}
```

```
def tenX(x):
    return 10*x

def do_twice(f, x):
    return f(f(x))

print(do_twice(tenX, 2))
```

datastructur.es

Summary: (Implementation Inheritance)

VengefulSLList extends SLList means a VenglefulSLList is-an SLList. Inherits all members!

- Variables, methods, nested classes.
- Not constructors.
- Subclass constructor must invoke superclass constructor first.
- Use super to invoke overridden superclass methods and constructors.

Invocation of overridden methods follows two simple rules:

- Compiler plays it safe and only lets us do things allowed by **static** type.

  Compiler chooses to:

- For overridden methods the actual method invoked is based on dynamic type of invoking expression, e.g. Dog.maxDog(d1, d2).bark();
- Can use casting to overrule compiler type checking.

Things we should not do with inheritance:

- a subclass has **variables** with the same name as a superclass.
- subclass has a **static method** with the same signature as a superclass method

Interfaces and Abstract Classes

Inheritance:

- **Interface** inheritance: What (the class can do). → **abstract** method
- **Implementation** inheritance: How (the class does it). → **default** method

Interfaces may combine a mix of abstract and default methods.

- **Abstract** methods are what. And must be overridden by subclass.
- **Default** methods are how.

Not explicitly mentioned on a slide before:

- Unless you use the keyword **default**, a method will be **abstract**.
  (no default → abstract method)
- Unless you specify an access modifier, a method will be public.
  (no modifier → public)

Interface can provide **variables**, but they are **public static final**.

- final means the value can never change. Use for constants: G=6.67e-11

A class can implement multiple interfaces.

Interface Summary:

- Cannot be instantiated.
- Can provide either **abstract** or **concrete** methods.
  - Use no keyword for abstract methods.
  - Use **default** keyword for concrete methods.
- Can provide only public static final variables.

Abstract Class:

Implementations must override ALL abstract methods.

Abstract classes are often used as partial implementations as interfaces.

Introducing: Abstract Classes

Abstract classes are an intermediate level between interfaces and classes.

- Cannot be instantiated.
- Can provide either **abstract** or **concrete** methods.
  - Use **abstract** keyword for abstract methods.
  - Use no keyword for concrete methods.
- Can provide variables (any kind).
- Can provide protected and package private methods [after mt1].

Similarities

Differences

opposite of interfaces

```
public abstract class GraphicObject {
    public int x, y;
    ...
    public void moveTo(int newX, int newY) {  ... }
    public abstract void draw();
    public abstract void resize();
}
```

GraphicObject

Summary: Abstract Classes vs. Interfaces

Interfaces:

- Primarily for interface inheritance. Limited implementation inheritance.
- Classes can ***implement multiple interfaces***.

Abstract classes:

- Can do anything an interface can do, and more.
- Subclasses only extend ***one*** abstract class.

you should generally prefer interfaces whenever possible.

- More powerful programming language constructs introduce complexity.

Comparable and Comparator

Comparable (java.lang → no need to import)

The industrial strength approach: Use the built-in Comparable interface.

- Already defined and used by tons of libraries. Uses generics.

```
public interface Comparable<T> {
     public int compareTo(T obj);
}
```

```
public interface OurComparable {
     public int compareTo(Object obj);
}
```

Comparable Advantages

- Lots of built in classes implement Comparable (e.g. String).
- Lots of libraries use the Comparable interface (e.g. Arrays.sort)
- Avoids need for casts.

```
public class Dog implements Comparable<Dog> {
    public int compareTo(Dog uddaDog) {
        return this.size - uddaDog.size;
    }
```

← Much better!

```
public class Dog implements OurComparable {
    public int compareTo(Object obj) {
        Dog uddaDog = (Dog) obj;
       return this.size - uddaDog.size;
  } ...
```

Implementing Comparable allows library functions to compare custom types (e.g. finding max).

```
Dog[] dogs = new Dog[]{d1, d2, d3};
Dog largest = Collections.max(Arrays.asList(dogs));
```

Comparable interface has the method **"int compareTo(Object)"**

- Returns: a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Comparator (java.util → have to import)

Parameters:

- o1 - the first object to be compared.
- o2 - the second object to be compared.

Returns:

- a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Additional Orders in Java

In some languages, we'd write two comparison functions and simply pass the one we want :

- sizeCompare()
- nameCompare()

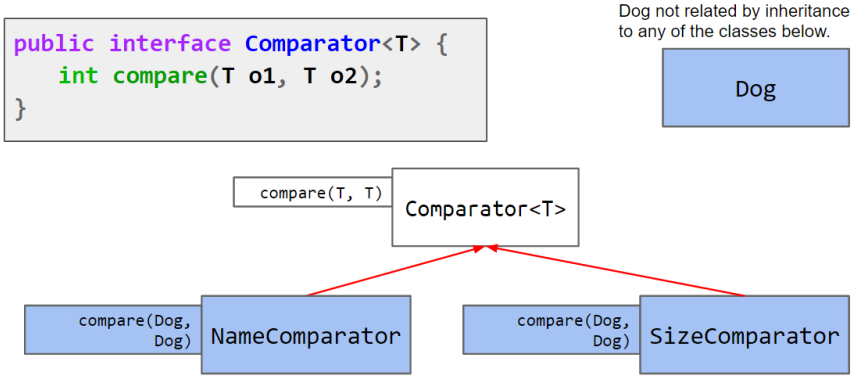
The standard Java approach: Create `sizeComparator` and `nameComparator` classes that implement the `Comparator` interface.

- Requires methods that also take Comparator arguments (see project 1B).

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

Dog not related by inheritance to any of the classes below.



Example: NameComparator

```
public class Dog implements Comparable<Dog> {
  private String name;
  private int size;

  public static class NameComparator implements Comparator<Dog> {
    public int compare(Dog d1, Dog d2) {
      return d1.name.compareTo(d2.name);
    }
  }
  ...
}
```

```
Comparator<Dog> cd = new Dog.NameComparator();
if (cd.compare(d1, d3) > 0) {
    d1.bark();
} else {
    d3.bark();
}
```

Result: If d1 has a name that comes later in the alphabet than d3, d1 barks.

Comparable and Comparator Summary (provide us with the ability to make call-backs)

- Some languages handle this using explicit function passing.

- In Java, we do this by wrapping up the needed function in an interface (e.g. Arrays.sort needs compare which lives inside the comparator interface)
- Arrays.sort “calls back” whenever it needs a comparison.

Java Libraries and Packages

Abstract Data Types

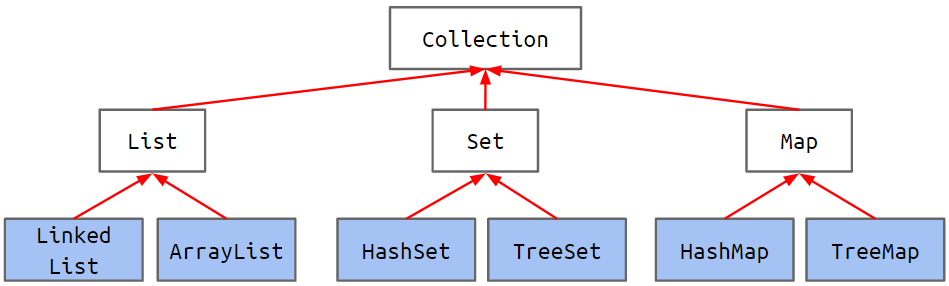
An Abstract Data Type (ADT) is defined only by its ***operations***, **not** by its **implementation**.

Collections

Among the most important **interfaces** in the **java.util library** are those that extend the Collection interface (btw interfaces can extend other interfaces).

The built-in java.util package provides a number of useful:

- Interfaces: ADTs (lists, sets, maps, priority queues, etc.) and other stuff.
- Implementations: Concrete classes you can use.



Packages

Canonicalization

- ***Canonical representation***: A unique representation for a thing. Cannot be reused or changed

In Java, we (attempt to) provide canonicity through by giving every class a “package name”.

- A package is a **namespace** that organizes classes and interfaces.

To address the fact that classes might share names:

We won't follow this rule. Our code isn't intended for distribution.

- A package is a ***namespace*** that organizes classes and interfaces.
- Naming convention: Package name starts with website address (backwards).

```
package ug.joshh.animal;

public class Dog {
    private String name;
    private String breed;
    private double size;
```

Dog.java

If used from the outside, use entire ***canonical name***.

```
ug.joshh.animal.Dog d =
    new ug.joshh.animal.Dog(...);
```

```
org.junit.Assert.assertEquals(5, 5);
```

If used from another class in same package (e.g. ug.joshh.animal.DogLauncher), can just use ***simple name***.

Importing Classes

Typing out the entire name can be annoying.

- Entire name:

```
ug.joshh.animal.Dog d =
    new ug.joshh.animal.Dog(...);
```

- Can use import statement to provide shorthand notation for usage of a single class in a package.

```
import ug.joshh.animal.Dog;
Dog d = new Dog(...);
```

- Wildcard import: Also possible to import multiple classes, but this is often a bad idea!
  - Use sparingly.

```
import ug.joshh.animal.*;
Dog d = new Dog(...);
```

Dangerous! Will cause compilation error if another * imported class contains Dog.

Importing Static Members

On the previous slide we saw how to import classes.

- Example:

```
import org.junit.Assert;
Assert.assertEquals(5, 5);
```

import static lets us import static members of a class.

- Example:

```
import static org.junit.Assert.assertEquals;
assertEquals(5, 5);
```

We've done this already. This is probably the only wildcard import that you should do in this course.

```
import static org.junit.Assert.*;
assertEquals(5, 5);
```

Unlikely that any other library will have any static members with same name as members in org.junit.Assert class

Generics

Basic Generics (array/primitive types)

- To create a **generic array:** E[] arr = (E[])new Object[LENGTH];
- Generic classes require us to provide one or more **actual type arguments**

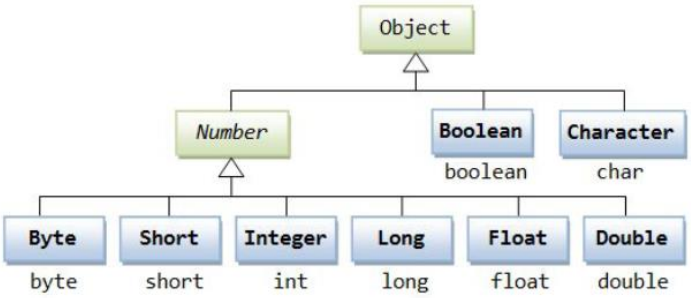
```java
import java.util.ArrayList;

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.

e.g. initialize ArrayList: ArrayList<String> L = new ArrayList<>();

- **Primitives Cannot** Be Used as Actual Type Arguments
- For each primitive type, there is a corresponding reference type called a wrapper class.



- Use wrapper type as actual type parameter instead of primitive type.

Autoboxing

Conversion between int and Integer is annoying, so autoboxing and unboxing is introduced.

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

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

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.

Primitive Widening

Widening: char → int → double

- 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);
```

- From string to int: Integer.parseInt

Immutability

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.

Final and Immuatblility

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).

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!

pros and cons

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.

Defining Generic Classes

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

A Mysterious Error with Autoboxing and Junit Test

The Issue:

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

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

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

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

Formal type parameter definitions.

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.

Type Upper Bound: Comparable

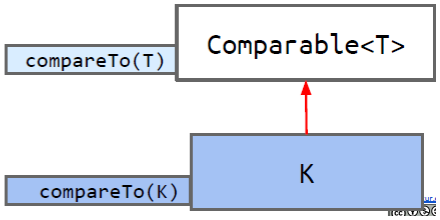
Can use extends keyword as a ***type upper bound***. Only allow use on ArrayMaps with Comparable keys.

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

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

Exceptions

Explicit Exceptions

We can also throw our own exceptions using the **throw** keyword.

- Can provide more informative message to a user.
- Can provide more information to some sort of error handling code.

Java code can also throw exceptions explicitly using ***throw*** keyword:

```
public static void main(String[] args) {
    System.out.println("ayyy lmao");
    throw new RuntimeException("For no reason.");
}
```

Creates new object of type RuntimeException!

```
$ java Alien
ayyy lmao
Exception in thread "main"
java.lang.RuntimeException: For no
reason.
    at Alien.main(Alien.java:4)
```

What has been Thrown, can be Caught

Can 'catch' exceptions instead, preventing program from crashing. Use keywords **try** and **catch** to break normal flow.

Catch blocks can execute arbitrary code.

- May include corrective action.

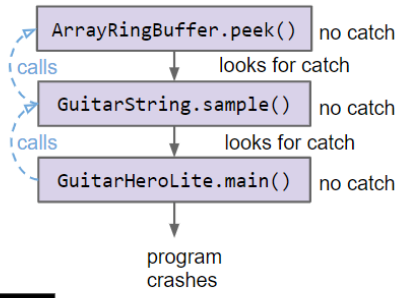
```
Dog d = new Dog("Lucy", "Retriever", 80);
d.becomeAngry();

try {
    d.receivePat();
} catch (Exception e) {
    System.out.println(
        "Tried to pat: " + e);
    d.eatTreat("banana");
}
d.receivePat();
System.out.println(d);
```

```
$ java ExceptionDemo
Tried to pat: java.lang.RuntimeException:
grrr... snarl snarl
Lucy munches the banana

Lucy enjoys the pat.

Lucy is a happy Retriever weighing 80.0
standard lb units.
```

Exceptions and the Call Stack

When an exception is thrown, it descends the call stack.

- If exceptions reaches the bottom of the stack, the program crashes and Java provides a message for the user.
  - Ideally the user is a programmer with the power to do something about it.




```
java.lang.RuntimeException in thread "main":
    at ArrayRingBuffer.peek:63
    at GuitarString.sample:48
    at GuitarHeroLite.java:110
```

Checked Exceptions

Compiler requires that all checked exceptions be **caught or specified**.

Two ways to satisfy compiler:

- **Catch**: Use a catch block after potential exception.
- **Specify** method as dangerous with **throws keyword**.
  - Defers to someone else to handle exception.

Two ways to satisfy compiler: *Catch* or *specify* exception.

```
public static void gulgate() throws IOException {
   ... throw new IOException("hi"); ...
}
```

```
public static void main(String[] args) {
  try {
    gulgate();
  } catch(IOException e) {
    System.out.println("Averted!");
  }
}
```

Catch an Exception:
Keeps it from getting out.

Use when you can handle the problem.

```
public static void main(String[] args)
    throws IOException {
      gulgate();
}
```

Specify that you might throw an exception.

Use when someone else should handle.

Nested Classes

Can combine two classes into one file pretty simply.

```
public class SLList {
   public class IntNode {
      public int item;
      public IntNode next;
      public IntNode(int i, IntNode n) {
         item = i;
         next = n;
      }
   }

   private IntNode first;
   public SLList(int x) {
      first = new IntNode(x, null);
   } ...
```

Nested class definition.

Could have made `IntNode` a private nested class if we wanted.

Instance variables, constructors, and methods of `SLList` typically go below nested class definition.

Nested Classes are useful when a class doesn't stand on its own and is obviously subordinate to another class.

- Make the nested class private if other classes should never use the nested class.

Nested classes are divided into two categories:

- **static nested class:** Nested classes that are declared static are called static nested classes.
    - And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.
- **inner class:** An inner class is a non-static nested class.
    - To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object

Static Nested Classes

If the nested class never uses any instance variables or methods of the outer class, declare it static.

- Static classes cannot access outer class's instance variables or methods.
- Results in a minor savings of memory. See book for more details / exercise.

```
public class SLList {
   private static class IntNode {
      public int item;
      public IntNode next;
      public IntNode(int i, IntNode n) {
         item = i;
         next = n;
      }
   ...
```

We can declare IntNode static, since it never uses any of SLList's instance variables or methods.

Analogy: Static methods had no way to access "my" instance variables. Static classes cannot access "my" outer class's instance variables.

Unimportant note: For private nested classes, access modifiers are irrelevant.

Difference between static and inner (non-static nested) classes

- **Static nested classes** do not directly have access to other members (non-static variables and methods) of the enclosing class because as it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.
- **Non-static nested classes (inner classes)** has access to all members (static and non-static variables and methods, including private) of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Citation and more examples: https://www.geeksforgeeks.org/nested-classes-java/

Iteration

The Enhanced For Loop

- public interface **Iterable**<T> { public Iterator<T> iterator(); }
- public interface **Iterator**<T> { boolean **hasNext**(); T **next**(); → returns curr and advance to next }
- Implementing the Iterable interface allows an object to make use of the for-each loop. It does that by internally calling the iterator() method on the object!
- ListIterator: built in interface in java (could move forward and backward)

The Secret of the Enhanced For Loop

For code on the right to work:

- Compiler checks that Lists have a method called iterator() that returns an Iterator<Integer>.
- Then, compiler checks that Iterators have:
  - hasNext()
  - next()

```
List<Integer> friends = new ArrayList<Integer>();
```

```
for (int x : friends) {
   System.out.println(x);
}
```

```
Iterator<Integer> seer
     = friends.iterator();

while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

SLinkedList

- `iterator()` returns an object of type `Iterator` that points to the head of the provided list.
- `next()` returns the element of the list that the `Iterator` is currently referencing, and then moves to the next node.

```
public class SLinkedList<E> implements Iterable<E> {
    private SNode<E> head;
    public SLLIterator iterator() {
        return new SLLIterator(this);
    }
    private class SLLIterator implements Iterator<E> {
        SNode<E> cur;
        SLLIterator(SLinkedList<E>  list) {
            cur = list.head;
        }
        public boolean hasNext() {
            return (cur != null);
        }
        public E next() {
            SNode<E> tmp = cur;
            cur = cur.next;
            return tmp.element;
        }
    }
}
```

Access Control

Access Levels

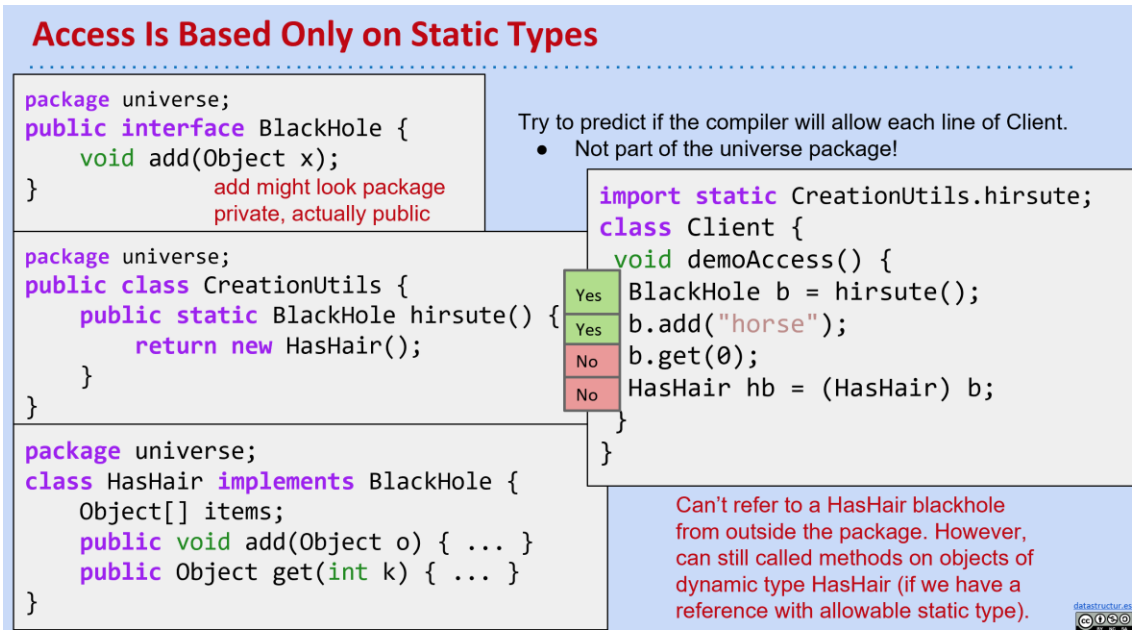
- ***Public*** declarations are declarations of the specification for the package, i.e. what clients of the package can rely on. Once deployed, these should not change.
- ***Private*** declarations are parts of the implementation of a class that only that class needs.
- ***Protected*** declarations are things that subtypes might need, but subtype clients will not.
- ***Package-private*** declarations are parts of the implementation of a package that other members of the package will need to complete the implementation.

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
	Y	Y	N	N
private	Y	N	N	N

The Default Package

Code without a package label is part of an unnamed package, a.k.a. the "**default package**".

- Everything is package private.
- Everything is part of the same (unnamed) default package.



Object Methods: Equals and toString()

toString()

toString()

If you want a custom String representation of an object, create a toString() method.

- Nothing particularly interesting about it, except that you can rely on it to generate a (nice?) String representation.

```
public String toString() {
        String mood;
        if (angry == true) {
           mood = "displeased";
        } else {
           mood = "happy";
        }
        return name + " is a " + mood + " " + breed +
               " weighing " + size + " standard lb units.";
}
```

```
Dog d = new Dog("Lucy",
                "Dachshund", 25);
System.out.println(d);
```

Rules for Equals in Java

Java convention is that equals must be an equivalence relation:

- Reflexive: x.equals(x) is true.
- Symmetric: x.equals(y) is true iff y.equals(x)
- Transitive: x.equals(y) and y.equals(z) implies x.equals(z).

Must also:

- Take an Object argument.
- Be consistent: If x.equals(y), then x must continue to equal y as long as neither changes.
- **Never true for null**, i.e. x.equals(null) must be false.

Difference between Set, List and Map in Java

Duplicate Object:

- **List**: allow **duplicate** objects (ArrayList/LinkedList)
- **Set**: **doesn't** allow duplicate objects (HashSet)
- **Map**: a key and a value and It may contain duplicate values, but **keys are always unique**. (HashMap/Hashtable)

Citation(Read more): http://www.java67.com/2013/01/difference-between-set-list-and-map-in-java.html#ixzz5nawAyp5H