CSE 219
COMPUTER SCIENCE III

Object Oriented Programming
What is memory?

- It’s basically a giant array of bytes!
- How do we access this memory?
  - in Java, we don’t!
  - the JVM handles it for us
    - using memory addresses
- The Java program uses object IDs
  - You must have seen these while debugging!
Memory “segments”

- **Text segment**
  - stores program instructions
Memory “segments”

- Global segment
  - data that can be reserved at compile time
  - global data like static methods and variables
Memory “segments”

- **(System) Stack segment**
  - *temporary variables inside methods*
  - *method arguments*
  - This data is removed from the stack once the method returns

- **Heap segment**
  - *dynamic data*
    - *instantiation of new objects*
    - This data is “persistent”:
      - As long as there is a reference to an object in this segment, it will not be removed.
That’s all very fine, but …

- The JVM handles all this automatically, so why should the end programmer care?
  - To understand the use (and abuse) of fundamental Java concepts like
    - Generics and Type Abstraction
    - Actual vs Apparent types
    - Call-by-value
    - Static vs non-static
Designing a good framework

■ It’s complicated …

■ Extensive use of
  – static and non-static members
  – inheritance
  – abstraction
    ■ abstraction is extremely important to make sure the framework is extensible
      – which is kind of the whole point of making a framework in the first place!
What is this “abstraction”?

- Ignoring some low-level details, and
  - Focusing on the big picture!
  - Allows us to get a simpler, more streamlined solution to the overall problem being addressed

- Abstraction is
  - the very first logical step in any design
  - The main question is this:
    - “What part of the problem can be abstracted out?”
      - e.g., An `add()` method should not be designed for lists or sets separately, but abstracted out to a higher-level, more general, solution!
Type Abstraction

- Abstract from a data type to families of related types
  - a many-to-one mapping
  - e.g., `public void equals(Object o)`

- How can we do this?
  - Inheritance and Polymorphism

- To understand type abstraction
  - it helps to first know how objects are managed by Java
Types

■ **Every** variable has a **type**.
  - The type defines how the variable is treated
  - How much memory to allocate for it

■ **Primitive data type**
  - `int, short, long, byte, float, double, char, boolean`

■ **Class data type**
  - Any object of a class *(i.e., not a primitive)* is said to have the type “class”.
  - This is also called “Object type” or “Reference type”.

■ **Generic type**
  - This is a generic class/interface that is *parameterized* over types
  - The parameter can be replaced by a concrete “class type”
Strong Typing

- **Java is a *strongly-typed* language**
  - This term has slightly different definitions (depending on who you’re asking)
  - But in general, this means that the type of every variable is a syntactic property
    - The compiler can determine the type without actually running the program.
    - A variable can only be used in a way that respects the restrictions of its type.
      - *e.g.*, it is impossible to perform a Double operation on an Integer

Hence the need for typecasting!
Runtime Safety

- Runtime safety in Java is designed to ensure that no variable can be interpreted as something it is not.
  - *e.g.*, for a String s, \texttt{s.charAt(i)} will throw an error if \texttt{i} is not in the range \(0, \ldots, \text{s.length()} - 1\).
  - Some languages (*e.g.*, C and C++) do not provide such safety nets!

- Advantages of runtime safety and strong typing:
  - Find errors early at compile time
  - Improves “fool proof” programming
  - Improves safety against malicious users (*applets* are generally web-accessible)
A student is a person

- Casting:
  
  ```java
  Student s = new Student();
  Person p = (Person) s;
  ```

```java
public class Person {
    public String firstName;
    public String lastName;
    public String toString() {
        return firstName + " " + lastName;
    }
}

public class Student extends Person {
    public double GPA;
    public String toString() {
        return "" + GPA;
    }
}
```
Typecasting (again)

■ Remember Java’s type hierarchy?
  - and in particular, how it restricts narrowing conversions?

■ Casting is a narrowing conversion.
  - An object can always be cast to one of its ancestor types

```java
Person p = new Person();
Student s = new Student();
p = new Student();
s = new Person();
p = (Person)new Student();
p = (Student)new Student();
s = (Person)new Person();
s = (Student)new Person();
```
Typecasting (again)

■ Remember Java’s type hierarchy?
  - and in particular, how it restricts narrowing conversions?
■ Casting is a narrowing conversion.
  - An object can always be cast to one of its ancestor types

```
Person p = new Person();
Student s = new Student();
p = new Student();
s = new Person();
```

Which lines would produce compiler errors?

```
p = (Person)new Student();
p = (Student)new Student();
```

Which lines would produce run-time errors?

```
s = (Person)new Person();
s = (Student)new Person();
```
Little boxes (of data)

- We have spoken about any object being conceptualized as a box before
  - When we instantiate an object by calling `new`, the assigned variable is a reference to that box
    - Actually, it is an ID
  - Multiple references to the same box can exist
    - But after calling `new`, more variables cannot be added to the same box
We have spoken about any object being conceptualized as a box before

- When we instantiate an object by calling new, the assigned variable is a reference to that box
- Actually, it is an ID
- Multiple references to the same box can exist
- But after calling new, more variables cannot be added to the same box

```java
Person p = new Student();

firstName: null
lastName: null
GPA: 0.0
```

```java
Student s = new Person();

firstName: null
lastName: null
```
Java Collections

■ A very abstracted framework.
  - Perhaps the most used abstracted code in Java

■ Examples:
  - `ArrayList` implements `List`
    ■ i.e., an `ArrayList` can be passed to any method that takes a `List`.
    ■ similarly, so can a `LinkedList`.
  - Abstracted methods for comparison
    ■ Extremely useful for sorting!
    ■ `Comparable` and `Comparator`
Sortable students

■ A very practical abstraction
  - `Collections.sort`
  - `overloaded` method

■ `public static <T extends Comparable<? super T>> void sort(List<T> list)`
■ `public static <T> void sort(List<T> list, Comparator<? super T> c)`

■ Then we will look at (again!)
  - `Comparable` versus `Comparator`
Using the *natural* ordering

- The comparable interface imposes a “natural” ordering
  - a natural order is a **total order**

- So what is a “total order” ?
  - a binary relation \( R \) on a set ( let’s call it \( S \) )
    - **transitive**
      - \( \forall x, y, z \in S : xRy \land yRz \Rightarrow xRz \) e.g., \( >, \geq, = \) are transitive (on the set of numbers)
    - **anti-symmetric**
      - \( \forall x, y \in S : xRy \land yRx \Rightarrow x = y \) e.g., \( \geq \) is an anti-symmetric relation
    - **total**
      - \( \forall x, y \in S : xRy \lor yRx \) e.g., \( \geq \) is a total ordering
Using the *natural ordering*

```java
public class Student extends Person implements Comparable<Student> {

    public double GPA;

    public String toString() { return "" + GPA; }

    public int compareTo(Student s) {
        if (GPA > s.GPA) return 1;
        else if (GPA < s.GPA) return -1;
        else return 0;
    }
}
```
public static void main(String[] args) {

    ArrayList<Student> students = new ArrayList<>();
    students.add(new Student("Bob", "Marley", 3.9));
    students.add(new Student("Joe", "Satriani", 2.5));

    // this is our code. we know what's happening here
    System.out.println(students.get(0).compareTo(students.get(1)));

    // but this depends on the list implementation
    System.out.println(Collections.sort(students));

    This is the 1st of the two overloaded Collections.sort() methods.
Suppose we have

- `public class Person implements Comparable<Person>

Then, can we have this

- `public class Student extends Person implements Comparable<Student>

No 😞

- ‘java.lang.Comparable’ cannot be inherited with different type arguments: ‘comparisons.Person’ and ‘comparisons.Student’
To compare or not to compare …

■ Suppose that were allowed
  - `public class Person implements Comparable<Person>`
  - `public class Student extends Person implements Comparable<Student>`

■ What do the corresponding `compareTo()` methods look like?

```java
@Override
public int compareTo(Person that) {
    return this.toString().compareTo(that.toString());
}

@Override
public int compareTo(Student that) {
    // the GPA comparison code we saw earlier
}
```
To compare or not to compare …

■ Then, effectively, we have one class implementing two comparable interfaces
  – one by itself
  – one inherited from its superclass

■ If these two implementations have different parameter types (e.g., Student and Person)
  – what do you think happens after type erasure?
To compare or not to compare ...

■ We *could* override `compareTo()` in the extended class

```java
@override
public int compareTo(Person p) {
    if (p instanceof Student) {
        Student that = (Student) p;
        if (this.gpa > that.gpa) return 1;
        if (this.gpa < that.gpa) return -1;
        return 0;
    } else 
        throw new IllegalArgumentException("Impostor!");
}
```

■ *Defeats the whole idea of abstraction* !!

■ It’s much better to simply fall back on the Comparator
public class StudentComparator implements Comparator<Student> {

    @Override
    public int compare(Student s1, Student s2) {
        if (s1.GPA > s2.GPA)
            return -1;
        else if (s1.GPA < s2.GPA)
            return -1;
        else
            return 0;
    }
}
public class StudentComparator implements Comparator<Student> {

    @Override
    public int compare(Student s1, Student s2) {
        if (s1.GPA > s2.GPA)
            return -1;
        else if (s1.GPA < s2.GPA)
            return -1;
        else
            return 0;
    }
}

This gets used by the 2\textsuperscript{nd} of the two overloaded \texttt{Collections.sort()} methods.
JAVA: apparent vs actual types

■ Apparent type
  - what the variable declaration said
  - Person p; // apparent type is Person
  - this is what the compiler cares about
  - can tell simply by looking at the declaration

■ Actual type
  - what it really is
  - Person p = new Student(); // actual type is Student
  - this is what the JVM cares about
  - can tell only by tracing through the code at runtime
JAVA: apparent vs actual types

■ Apparent type
  - determines what methods can be invoked on an object
    ■ what if the method is overridden again and again?

■ Actual type
  - determines *which* implementation of the method is called
    ■ *e.g.*, if the apparent type is `Person` but the actual type is `Student`,
      which implementation of the `toString()` method will be invoked?
  - the JVM knows the actual type
    ■ and calls the implementation in that class
Pass-By-Value (a.k.a. call-by-value)

- The actual argument is fully evaluated
- The resulting value is copied into a location
  - This is the location used to hold the argument’s value during method execution
    - typically a chunk of memory on the runtime stack for the application
      - in JAVA (some other programming languages do things differently)

- JAVA is **strictly** pass-by-value!
  - stay away from people who claim otherwise
Pass-By-Value

- Often, you may hear statements like
  - “In Java, objects are passed by reference, and primitives are passed by value.”
    - This is a half-truth!
- Primitives are, of course, passed by value
- But objects are **not passed by reference**!
  - _references_ to objects are passed by value

- This may look like we are splitting hairs
  - but it’s a very important hair
Pass-By-Value: an example

```java
public void foo(Dog d) {
    d = new Dog("Fifi"); // bow wow
}

public static void main(String[] args) {
    Dog aDog = new Dog("Max"); // woof woof
    foo(aDog);
    // aDog is still Max
}
```
Pass-By-Value: an example

```java
public void foo(Dog d) {
    d = new Dog("Fifi"); // bow wow
}

public static void main(String[] args) {
    Dog aDog = new Dog("Max"); // woof woof
    foo(aDog);
    // aDog is still Max
}
```

The value of `aDog` in `main()` is not overwritten in `foo()`
How is my dog still Max?

- Primitives are passed by value
- Object references are passed by value
  - An object itself is never passed to a method
  - The object is always in the heap
    - Only a reference to the object is passed to a method
- References
  - this is Java nomenclature
  - everywhere else in the world of programming languages, they’re called pointers, and Java couldn’t avoid it entirely …
  - NullPointerException
Think of balloons instead of dogs

- Imagine a balloon
- Calling a function is like tying a 2\textsuperscript{nd} string to the balloon and handing the line to the function parameter
- "= new Balloon()" will cut that string and create a new balloon
  - but this has no effect on the original balloon!
- Java is pass by value, but the value passed is not deep, it is at the highest level, i.e. a primitive or a pointer.
  - Don't confuse that with a deep pass-by-value where the object is entirely cloned and passed.
Interfaces

- “Contracts” of sorts
  - define how communication happens (through methods)
  - without getting into the implementation details

- In JAVA, an interface
  - is a reference type (just like classes)
  - can only contain
    - constants
    - method signatures
    - default methods
    - static methods
    - nested types
An interface is a reference data type

- You can use an interface name anywhere you can use any other data type name

- A variable $v$ whose (apparent) type is an interface
  - any assignment to $v$ must be an instance of a class that implements that interface
Designing an interface

- You have an interface with two method signatures
  - doThis(int x, String y)
  - doThat(String y)

- But in future, you realize that it’d be nice to have another functionality
  - so you want to add a 3rd signature doMore(String y, String z)

- Then, all classes implementing this interface will break!

- Anticipating all future uses is difficult
  - So, interfaces can also be extended 😊
Abstract classes and methods

- **Abstract classes**
  - declared using the keyword `abstract`
    - `public abstract class Klass { ... }`
  - may or may not have abstract methods
    - `abstract void m();`
  - if a class contains an abstract method
    - it **must** be declared `abstract`
  - cannot be instantiated (just like interfaces)
  - but they can be extended by subclasses

- **Highly recommended**
  - [http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html](http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html)