Announcements/Follow-ups

• Midterm #2 Friday
  – Everything up to and including today
  – Review section tomorrow
  – Study set # 6 online
    • answers posted later today

• P5 due next Tuesday
  – A good way to study
  – Style – omit “this” when unnecessary

• Lab 07 questions?
  – Returning a reference to the current object

• Immutability and shallow copies
When to use **this**

- In a constructor:
  ```java
class MyObj {
    int data = 0;

    public MyObj(int dat) { //Constructor
      data = dat;
    }

    // Another constructor
    public MyObj(int data) {
      this.data = data; // Use 'this' to refer to the current object
    }
  }
  ```

  VS

  ```java
class MyObj {
    int data = 0;

    public MyObj(int dat) { //Constructor
      // data = dat; // No need for 'this'
    }

    // Another constructor
    public MyObj(int data) {
      // data = data; // No need for 'this'
    }
  }
  ```
When to use `this`

- To return the object at the end.

  Compare:

  ```java
  public void grow(int more) {
      data = data + more;
  }
  ...
  
  //Somewhere else
  obj1.grow(3)
  System.out.println(obj1);
  ```
When to use `this`

- To return the object at the end.

With:

```java
public MyObj grow(int more) {
    data = data + more;
    return this;
}
```

...  

//Somewhere else
System.out.println(obj1.grow(3));
Immutables and shallow copy

Original array

Copy array

Can't be changed

Changes here

Don't Occur here

Changes here

Can't be changed

Don't Occur here

imm1

imm2

imm3
Built-in interfaces

• CharSequence
  – Declares charAt(...), length(), and more
  – Implemented by String, StringBuffer, and more

• Comparable
  – Declares compareTo()
  – Implemented by Integer, Boolean, String, and many more
  – Any array whose elements implement Comparable can be automatically sorted with Arrays.sort()
Built-in interfaces

• List
  – Declares get(), set(), indexOf(), and many more
  – Implemented by many “Collections” (next week)
  – “Optional” methods?? (e.g. set())
    • Must be implemented, but may throw “UnsupportedOperationException”
    • The other alternative would be no optional methods, but many more built-in interfaces
    • “Optional” methods not advised in your own interfaces
    • [Link](http://stackoverflow.com/questions/10572643/optional-methods-in-java-interface)
Purposes of Interfaces

• Organization
  – Enforces encapsulation at compile-time
  – Provides a contract for users (other programmers)
  – Interfaces should be designed carefully
    • Many classes may eventually implement the interface
    • Changes to the interface requires changes to all classes that implement it

• Polymorphism
Polymorphism

• “Poly-morphism” ↔ “Many-forms”
• Informally: The ability to type-cast non-primitive data
  ```java
  int i = 3;
  float f = (float) i;
  
  Circle c = new Circle(3);
  Shape s = (Shape) c;
  ```
• Inheritance also facilitates polymorphism (more later)
• Set theory basics
Set Theory

- **Set**: A collection of distinct elements (unordered)

![Rectangles](image-url)
Set Theory

- Set: A collection of distinct elements (unordered)

Integers (\(\mathbb{Z}\))

1, -2, 7, 100
Set Theory

- Set: A collection of distinct elements (unordered)

Rationals ($\mathbb{Q}$)

- $\frac{1}{3}$
- $-\frac{2}{9}$
- $\frac{7}{44}$
- $1.01$
Set Theory

• Set: A collection of distinct elements (unordered)

• Typically written as \{element1, element2,...\}
  – The positive, even numbers less than 10 (finite)
    • \{2, 4, 6, 8\}
    • \{2, 6, 4, 8\}
    • \{2, 4, 6, 8\}
    • \{8, 4, 6, 2\}
Set Theory

- Sets can overlap
Set operations

• Subset: One set entirely contained in another
Set operations

- Subset: One set entirely contained in another

$\mathbb{Q}$

$\mathbb{Z}$

$\mathbb{Z}$

Even $\mathbb{Z}$

{2, 4, 6, 8}
Set operations

• Subset: Mathematical symbol is $\subset$
  – Squares $\subset$ Rectangles (Four right angles)
  – Squares $\subset$ Rhombuses (Four equal sides)
  – $\mathbb{Z} \subset \mathbb{Q}$

• Set union: Everything in either set
  – Rectangles $\cup$ Rhombuses = Parallelograms(?)
  – $\mathbb{Z} \cup \mathbb{Q} = \mathbb{Q}$

• Set intersection: Everything in both sets
  – Rectangles $\cap$ Rhombuses = Squares
  – $\mathbb{Z} \cap \mathbb{Q} = \mathbb{Z}$
Explicit/implicit casting

The rules of type-casting depend on the “is-a” (subset) relationship

• \( \mathbb{Z} \subset \mathbb{Q} \):
  – An integer is a rational number
  – A rational number is not necessarily an integer

• Circles \( \subset \) Shapes
  – A circle is a shape
  – A shape is not necessarily a circle

• A String is never a shape

• Polymorphism examples
Explicit/implicit casting

Which type-casts are necessary?

- `int i = (int) 3.0;`
- `float f = (float) 3;`
- `Shape s = (Shape) new Circle(3);`
- `Circle c = (Circle) s;`
- `String str = (String) c;`
Explicit/implicit casting

Which type-casts are necessary?

- `int i = (int) 3.0; //Yes`
- `float f = (float) 3; //No`
- `Shape s = (Shape) new Circle(3); //No`
- `Circle c = (Circle) s; //Yes`
- `String str = (String) c; //Error`

“is-a” relationship gives the correct answer (“hiding/destroying data” does not)
Polymorphic references

• As long as the underlying object implements Shape, it can be referenced by a Shape variable.

    //Casts are not necessary here
    Shape s = (Shape) new Circle(3);
    s = (Shape) new Triangle(3);
    s = (Shape) new Square(3);
Polymorphic references

- A reference variable is a *memory address*
- An object is a data structure on the heap

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle c</td>
<td>Circle object</td>
</tr>
<tr>
<td>Triangle t</td>
<td>double r</td>
</tr>
<tr>
<td>Shape s</td>
<td>Triangle object</td>
</tr>
<tr>
<td></td>
<td>double a</td>
</tr>
<tr>
<td></td>
<td>double b</td>
</tr>
<tr>
<td></td>
<td>double c</td>
</tr>
</tbody>
</table>
Polymorphic references

- A reference variable is a *memory address*
- An object is a data structure on the heap

Stack
- Circle c
- Triangle t
- Shape s

Heap
- Circle object
  - double r 3
- Triangle object
  - double a 3
  - double b 3
  - double c 3
Polymorphic references

• The Shape variable can call all of (and only) the methods in the Shape interface
  
  s.area();  //Valid
  s.dilate(3.0);  //Valid
  s.toString();  //Valid
  s.circumference();  //Invalid

• But the underlying object determines which method gets called:
  
  s.area();  //Circle or triangle area?  //Depends on the underlying object
Dynamic binding

• Different classes can have methods with the same name, so method calls must be “bound” to objects before they are executed
  – “Early”/ “Static” binding: at compile time
  – “Late”/ “Dynamic” binding: at run time

• Java uses late binding
  – With interfaces and polymorphic references, the class of the underlying object is not necessarily known until run-time
  – Late binding allows this sort of polymorphism, but at the cost of efficiency
Heterogenous arrays

Stack
Shape[] shapes

Heap
Circle object
double r 3

Circle object
double r 4

Triangle object
double a 3
double b 3
double c 3
Heterogenous Arrays

• Polymorphism allows us to create “heterogenous” arrays
  – Different classes in a single array

• If we stick to interface methods, we can manipulate heterogenous arrays the same way as usual:
  
  ```java
  Shape[] shapes = new Shape[6];
  shapes[3] = new Circle(3);
  shapes[4] = new Triangle(3,4,5);
  System.out.println(shapes[3].area());
  System.out.println(shapes[4].area());
  ```

• Array example
The Object class

• “Object” is an umbrella class that contains all other classes
• Any object can be stored in a variable of type Object
  – `Object obj = new Integer(3);`
  – `Object obj = new Scanner(System.in);`
  – ObjectExample
The Object class

• Object has its own set of methods:
  – Equals
  – toString
  – hashCode
  – getClass
  – And more:
    http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

• Every class has access to these methods by default (unless they get overridden)
  – MyObject example
The getClass() method

• GetClassExample
  – Shape is an interface
  – Shape shape is a reference variable
    • It points to an object with a specific class at run time
  – How do we determine the runtime class?
    • Calling the getClass() method.
  – “Dummy” objects: allocated simply for calling getClass()
  – buggingMe method
• Class, get getClass()?
Using `getClass()`

- The return type of `getClass()` is `Class`
  - “Class” is a class that represents classes (ugh...)
- You can declare variables of type `Class`:
  - `Class c = obj1.getClass();`
  - Warnings related to “generics” (next week)
- But typically you compare the Classes of two objects:
  - `obj1.getClass().equals(obj2.getClass())`
The standard equals method

- There is a standard way of defining equals that is widely accepted and expected

<table>
<thead>
<tr>
<th>Warning</th>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComplexNumber defines equals(ComplexNumber) method and uses Object.equals(Object)</td>
<td></td>
</tr>
</tbody>
</table>
The standard equals method

• What if we want to compare our object to *any* other object, not just others of the same type?
  – public boolean equals(Square other);
  – public boolean equals(Shape other);
  – public boolean equals(Object other);

• Square example – standard equals
Standard equals

• Public boolean equals(Object other) should contain:
  – Null check (is other null?)
  – Type check (is other the correct run-time type?)
  – Your checks
  • Typically, check that all the fields are the same. This will require a type-cast: (this.getClass()) other.
  • Application may have different requirements – for example, two equal “Set” objects may contain arrays with the same elements, but in different orders.