Object Oriented Programming in Java

Introduction to Computer Science II

Christopher M. Bourke
cbourke@cse.unl.edu

Object Oriented Programming in Java

Introduction

Java is Object-Oriented
- Not pure object oriented
- Differs from other languages in how it achieves and implements OOP functionality
- OOP style programming requires practice

Objects in Java

Java is a class-based OOP language

Definition

A class is a construct that is used as a blueprint to create instances of itself.

- Objects are concepts; classes are Java’s realization of that concept
- Classes are a definition of all objects of a specific type (instances)
- Classes provide an explicit blueprint of its members and their visibility
- Instances of a class must be explicitly created

Objects in Java

Contrast with prototype based OOP languages

- Example: JavaScript
- No classes, only instances (of objects)
- Constructed from nothing (ex-nihilo)
- Inheritance through cloning of original prototype object
- Interface is build-able, mutable at runtime

Objects in Java

Paradigm: Singly-Rooted Hierarchy

- All classes are subclasses of Object
- Object is a java object, not an OOP object
- Makes garbage collection easier
- All instances have a type and that type can be determined
- Provides a common, universal minimum interface for objects
- Avoids complex lattice relations in other OOP languages: no multiple inheritance

Objects in Java I

Object methods

```java
//ignore:
protected Object clone();
protected void finalize();
Class<? extends Object> getClass();

//multithreaded applications:
public void notify();
public void notifyAll();
public void wait();
public void wait(long timeout);
public void wait(long timeout, int nanos);

//important ones:
public boolean equals(Object obj);
public int hashCode();
public String toString();
```
Objects in Java II
Object methods

- `equals` returns true/false whether or not `obj` is equal to `this`
- Default behavior: uses reference comparison
- `hashCode` returns an integer hash of this object
- Default behavior: typically the memory address of the object
- `toString` provides a human readable representation of the object
- Default behavior: qualified class name plus JVM memory address

Declaring Classes
Syntax

- Classes must be defined in file of the same name (`<filename>.java`)
- Class names should be upper-camel case, singular form

```
package foo.bar;
/* imports here */
public class MyClass {
    /* static field declarations */
    /* member field declarations */
    /* member method declarations */
}
```

Object Life Cycle I

Instantiation

- New objects created using the `new` operator:
  ```
  Student s = new Student();
  ```
- All classes have a default constructor if no constructor is defined
- If non-default constructor defined, default constructor is not provided
- Can have multiple constructors
- Good practice: constructor variants should call other constructors when appropriate
- Example: `this(...)`
- If calling another constructor, must be done first!

Object Life Cycle II

Persistence

- Instances persist until they are out of scope, no longer referenced, etc.
- Can be serialized (written to disk, database, or transmitted over a network)
- `Serializable` interface, using
  ```java
  writeObject(java.io.ObjectOutputStream out)
  readObject(java.io.ObjectInputStream in)
  ```

Object Life Cycle III

Destruction

- Can set a reference to null: `Integer a = 10; a = null;`
- When no reference to an object exists, eligible for garbage collection
- No guarantee of when it gets destroyed
- No destructors in Java
- Cannot rely on `finalize()`
- Cannot rely on `System.gc()`
- Good practice: Do all cleanup manually in a `finally` block

Inner Classes

- Java allows you to declare a class within a class
- Visibility: allowed to be public, private, protected
- Static or not
- Static fields must be final
- No static methods allowed unless inner class is static
- Example?
Anonymous Classes I

- An anonymous class is a class declared and defined inline
- Not bound to a (class) name or identifier
- Useful idiom: allows you to provide a component declared and defined inline as an argument
- Not just a convenience: prevents reuse of an ad-hoc class
- Example: providing a comparator to the collection’s sort method

Anonymous Classes II

```java
List<Integer> myNumbers = new ArrayList<Integer>();
...
Collections.sort(myNumbers, new Comparator<Integer>() {
    @Override
    public int compare(Integer arg0, Integer arg1) {
        if (arg0 == null) return -1;
        else if (arg1 == null) return 1;
        else return arg0.compareTo(arg1);
    }
});
```

Note: Normally, the sort method cannot handle null values, here we do!

Abstract Classes

- A class can be made `abstract`
- Such a class cannot be instantiated
- Instances must be created from non-abstract subclasses
- May contain abstract and non-abstract methods
- Non-abstract methods provide a “default” behavior for subclasses

Need to provide a default? Use an abstract class! Don’t or don’t want to? Use an interface!

Abstraction

- All `public` methods and fields available to client code
- Static methods/fields available without an instance
- Interfaces (revisited)
- Abstract classes (revisited)
- `final` keyword:
  - A `final` class cannot be subclassed
  - A `final` method cannot be overridden in a subclass
  - A `final` field cannot be reassigned (does not imply immutable)

Interfaces I

An `interface` is an abstract type that defines method signatures that must be implemented by a class

- Methods are always public (or default) and abstract
- May also define `final static` constants
- No default implementation can be defined
- A class can `implements` multiple interfaces
- Allows us to simulate multiple inheritance without being locked into a hierarchy

Interfaces II

Syntax:
```java
public interface InterfaceName {
    return-type methodName();
    ...
}
```

```java
public class Foo implements InterfaceA, InterfaceB {
    ...
}
```
Interface Example

```java
1 public interface Gradeable {
2     public double getScore();
3     public String getLabel();
4 }
5 ...
6 public class Exam implements Gradeable {
7     ...
8     public double getScore() {
9         return this.numPoints / this.totalPoints;
10     }
11     public String getLabel() {
12         return "Exam: " + this.examNumber;
13     }
14 }
```

Advantages of an interface I

Interfaces provide a means to:
- Simulate multiple inheritance
- Provide interface (an is-a relationship) without locking you into a hierarchy

Advantages of an interface II

Example:

![Inheritance through interfaces](image)

Other interface items

- Collections framework: `java.util.List`, `java.util.Set`
- Note: an `interface` can `extend` other interfaces (sub/super interface)
- Example: `java.util.List extends java.util.Collection`
- Though not a class, still provides the is-a relation
- You can refer to an implementing class as its interface:

```java
1 List l = null;
2 ArrayList al = new ArrayList();
3 l = al;
```

Encapsulation

- Encapsulation achieved through declaring member fields, methods
- Visibility provides protection
- Common idiom: Mutators and Accessors (getters and setters)

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>none (default)</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Table: Java Access Levels

Inheritance I

- Java supports inheritance through `subclassing`
- Syntax: `extends` keyword
- Superclass methods and fields are inherited (provided they are not private)
- Subclasses may override superclass methods
- Good practice: use the `@Override` annotation
Inheritance II

- Subclass methods can access super class methods and fields using `super`.
- Can be a pure substitution: no new methods introduced in a subclass, or
- New methods may be declared in a subclass
- Preventing subclassing: `final` (examples: `java.lang.Integer`, libraries)
- Recall: all methods in Java are virtual by default
- Note: a pure virtual method in Java is an abstract method

Composition

- Java supports composition through the same mechanism as encapsulation
- A Java class can have classes as members
- What’s responsible for constructing it?
  - May be completely internal, may be provided, etc.

```java
class Foo {
    private Integer a;
    private Graph g;
    protected List<Integer> myList;
}
```

Polymorphism

- Many mechanisms for polymorphism
- Many different types of polymorphism supported
- Some types not supported:
  - Operator overloading not allowed
  - Behavioral polymorphism
- Polymorphic behavior not always apparent

Automatic Type Casting I

- Numerical operators support mixed types (int, double, etc.)
- Simpler types are up-casted to compatible types and (may be) downcasted in the final result
- A form of coercion

```
int a = 10;
double b = 20.5, c = 10.5, f;
int d, e;
//d = a + b; <- compile error.
// requires explicit cast to int, works in C!
f = a + b;
e = (int) (b + c);
System.out.println("f = "+f);
System.out.println("e = "+e);
```

Autoboxing I

- Autoboxing/unboxing: mixing primitive numeric types with Java wrapper classes
- As needed, compiler replaces expressions with value methods
- May result in runtime `NullPointerException`
- Need to be careful when mixing types with comparison operators
- Form of coercion

```
int a = 10;
double b = 20.5, c = 10.5, f;
int d, e;
//d = a + b; <- compile error.
// requires explicit cast to int, works in C!
f = a + b;
e = (int) (b + c);
System.out.println("f = "+f);
System.out.println("e = "+e);
```
Autoboxing II

1 Integer a = new Integer(10);
2 Double pi = new Double(3.14);
3 int b = 20;
4 double c = a + b + pi;
5 //becomes:
6 //double c = a.doubleValue() + b + pi.doubleValue();

Autoboxing III

From the Java documentation:

So when should you use autoboxing and unboxing? Use them only when there is an impedance mismatch between reference types and primitives, for example, when you have to put numerical values into a collection. It is not appropriate to use autoboxing and unboxing for scientific computing, or other performance-sensitive numerical code. An Integer is not a substitute for an int; autoboxing and unboxing blur the distinction between primitive types and reference types, but they do not eliminate it.

Or: when null values are needed to distinguish between missing/invalid/unknown data (tri-valued logic)

String Casting I

- Plus operator (+) is overloaded
- Could mean addition or (string) concatenation
- Can mix Strings and any other Object
- Since Object has a public String toString() method

String Casting II

How this works:

- Rather than dynamic dispatch, Java replaces string concatenation code
- For efficiency, compiler replaces + with StringBuilder instances
- StringBuilder uses (overloaded) static String.valueOf() method
- Converts null instances to the string "null", otherwise calls object’s toString() method
- A form of operator/method overloading and coercion

String Casting III

Example 1:

1 String a = “hello “ + “world “ + “!”;
2 //would actually become:
3 String a = new StringBuilder().append(“hello “)
4 .append(“world “).append(“!”).toString();

String Casting IV

Example 2:

1 Integer a = null;
2 System.out.println("a = “+a);
3 a = new Integer(10);
4 System.out.println("a = “+a);
5 a = null;
6 System.out.println("a = “+String.valueOf(a));
7 System.out.println("a = “+a.toString());
Upcasting

- Upcasting is when an instance is treated as its superclass
- Form of subtype polymorphism

```java
public class Dog extends Animal {
    public void doSomething(Animal a) {
        // a is treated as an animal here
    }

    public class Dog extends Animal {
        public void doSomething(Animal a) {
            // a is treated as an animal here
        }
    }
```

Downcasting I

- Downcasting works in the opposite direction
- An instance is explicitly cast to a subclass:
  ```java
  Animal a; ...
  Dog d = (Dog) a;
  ```
- A subclass always has an is-a relationship
- Does not work in this case: may not be-a!
- May be problematic if `a` is not a Dog: ClassCastException
- Useful tool: `instanceof`
- Useful tool: `getClass()` method, `.class` static element

Downcasting II

```java
Animal a = ...
Dog d = null;
if (a instanceof Dog) {
    d = (Dog) a;
}
``` 

Downcasting III

```java
Integer c = 10;
if (c instanceof java.lang.Number) {
    System.out.println("Its a numeric: "+c.getClass());
    if (c.getClass() == java.lang.Integer.class) {
        ...
    }
    if (c.getClass().equals(java.lang.Integer.class)) {
        ...
    }
```

Method Overloading I

- A method’s signature depends on its name and parameters
- Java allows you to overload methods; that is:
- Define methods with the same names but different (number or type) parameters
- Compiler uses dynamic dispatch to call the appropriate method
- However: cannot define methods with same name, parameters, but different (covariant) return types (more later)
Method Overloading II

```
1 public void foo() {}
2 public int foo(Integer a) {return 0;}
3 public int foo(Integer a, Double b) {return 0;}
4 public String foo(Double a) {return "0.0";}
5  //not allowed:
6  //public double foo(Double a) {return 0.0;}
```

Method Overloading II

```
1 public class Animal {
2  ...
3  protected void eat() {
4    System.out.println("Eating...");
5  }
6  ...
7 }
```

Method Overloading III

```
1 public class Dog extends Animal {
2  ...
3  @Override
4  public void eat() {
5    System.out.println("Eating a milkbone...");
6  }
7  ...
8 }
```

Method Overriding I

- In a subclass (non-private, non-final) methods can be **overridden**
- Implementation can be changed
- Method signature and return type cannot change
- Visibility can be **increased** but not restricted further
- Good practice: use the `@Override` annotation
- Still have access to the superclass's method (`super.methodName()`)

Method Overriding I

**toString method**

```
1 public class Student {
2  private int id;
3  private String firstName;
4  private String lastName;
5  ...
6  public String toString() {
7    return this.lastName + ", " + this.firstName + 
8       " (" + this.id + ")";
9  }
```

Covariant Return Types I

- In general, an overridden method must have the same return type
- Since Java 1.5: allowed to return a covariant type
- Return type can be changed to another type as long as the new return type has an is-a relation to the original
- Cannot be an arbitrary type
Covariant Return Types II

```java
public class A {
    public Animal getOrganism() {
        ...
    }
}

public class B extends A {
    @Override
    public Dog getOrganism() {
        ...
    }
}
```

Parameterized Polymorphism I

- Generics introduced in Java 1.5
- Named parameter syntax: `<T>`
  ```java
  public class Foo<T>{ ...
  }
  ```
- Multiple parameter types: `<T, U>`
- Usage:
  ```java
  Foo<Integer> mc = new Foo<Integer>();
  ```
- Explicit parameterized types relieve us of the need to do explicit casting
- Prevents incompatible type mixing (compile time check)

Parameterized Polymorphism II

```java
//no compile error, no run time error
ArrayList myList = new ArrayList();
myList.add(new Integer(10));
myList.add(new String("ten"));

//okay:
Integer a = (Integer) myList.get(0);

//runtime error:
Integer b = (Integer) myList.get(1);
```

Parameterized Polymorphism III

- Not the same as C++ templates
- Generics can be applied to: classes, interfaces, methods (as arguments, return types, or static methods)

```java
ArrayList<Integer> myList2 = new ArrayList<Integer>();
...
```

Parameterized Polymorphism IV

```java
public static <T> T getMax(Collection<T> items) {
    T max = items.iterator().next();
    for(T t : items) {
        if([some criteria here]) {
            max = t;
        }
    }
    return max;
}
```

Parameterized Polymorphism V

```java
//unparameterized list:
ArrayList myList = new ArrayList();
//list parameterized as Integer:
ArrayList<Integer> myList2 = new ArrayList<Integer>();
...
```

```java
//warning and possible runtime exceptions:
Integer a = getMax(myList);
//best way:
Integer b = getMax(myList2);
//compiler error:
String c = getMax(myList2);
```
Bounded Polymorphism I

- Revisit the `getMax` example: unable to establish a criterion for comparison
- No information other than `<T>` was an `Object`
- Illustrates the need for bounded polymorphism
- By default, all named parameters are bounded by `Object`
- Wildcard: `<?`, equivalent to `<? extends Object>`

Bounded Polymorphism II

- Upper bound: `<? extends T>` – matches `T` and any subtype of `T`
- Lower bound: `<? super T>` – matches `T` and any super type of `T`
- Named upper bounded: `<T extends Foo>` - matches a particular type `T` which must be a subclass of type `Foo`
- You can also do multiple lower bounds on named parameters (but not wildcards): `<T extends ClassA & Interface1 & ... & InterfaceN>`
  (note: you cannot have multiple classes as there is no multiple inheritance!)

Bounded Polymorphism III

- Cannot specify upper and lower bound on the same type (for either named or wildcard)
- Named lower bounds do not exist
  - Hypothetically, `<T super Foo>` would match a particular type which is a superclass of `Foo`
  - Not that useful though: it only provides a restriction that doesn’t guarantee an interface!
  - In contrast, lower bounds provide a restriction that guarantees a minimal interface

Bounded Polymorphism IV

```java
public static <T extends Comparable<T>> T getMax(
    Collection<T> items)
{
    T max = items.iterator().next();
    for (T t : items) {
        if (t.compareTo(max) < 0) {
            max = t;
        }
    }
    return max;
}
```

Bounded Polymorphism V

Using the `Comparable` interface:

```java
public static <T extends Comparable<T>> T getMax(
    Collection<T> items)
{
    T max = items.iterator().next();
    for (T t : items) {
        if (t.compareTo(max) < 0) {
            max = t;
        }
    }
    return max;
}
```

More on Bounded Parameterization I

It can be somewhat confusing on when to use named bounded parameters vs. wildcard bounded parameters.

In general, if the type is not referenced elsewhere in a method, you can (and arguably should) use a wildcard.

The argument is that using a wildcard “advertises” the fact that the type is unused and unimportant.

However, very little is “lost” if a named parameter is used (with a very few limited exceptions).
More on Bounded Parameterization II

The only thing you know about the `stuff` is that they are `Object`s:

```java
public static void foo(List<?> stuff) {
    Object o = stuff.get(0);
}
```

More on Bounded Parameterization III

Same, but the parameter type is named and it can now be used:

```java
public static <T> void bar(List<T> stuff) {
    T item = stuff.get(0);
}
```

More on Bounded Parameterization IV

By providing a bound we know more about the type `T`: its a subclass of `BankAccount` and therefore is a `BankAccount`:

```java
public static <T extends BankAccount> double baz(List<T> accounts) {
    double total = 0.0;
    for(BankAccount acct : accounts) {
        total += acct.getBalance();
    }
    return total;
}
```

More on Bounded Parameterization V

But we didn’t really use the type `T`, so we could have instead used a wildcard, but still provide a bound on the argument, guaranteeing a minimal interface.

```java
public static double biz(List<? extends BankAccount> accounts) {
    double total = 0.0;
    for(BankAccount acct : accounts) {
        total += acct.getBalance();
    }
    return total;
}
```

More on Bounded Parameterization VI

However, if we do need to reference the type in the method, then we must use a named parameter.

```java
public static <T extends BankAccount> T buz(List<T> accounts, T account) {
    if(account.getBalance() > 0) {
        accounts.add(account);
        return account;
    } else {
        return null;
    }
}
```

More on Bounded Parameterization VII

Alternatively, we can use both named parameters and wildcards to make our code a bit more general (and thus flexible to those using it). Here we have the same advantages as before, but now the list can be generalized to any super type of `T` (say, for example `List<Object>`).

```java
public static <T extends BankAccount> T boz(List<? super T> accounts, T account) {
    if(account.getBalance() > 0) {
        accounts.add(account);
        return account;
    } else {
        return null;
    }
}
```
More on Bounded Parameterization VIII

More:

- Good post: [http://stackoverflow.com/questions/3486689/java-bounded-wildcards-or-bounded-type-parameter](http://stackoverflow.com/questions/3486689/java-bounded-wildcards-or-bounded-type-parameter)

Type Erasure & Raw Types I

- Generics introduced in Java 1.5
- But Collections framework introduced in Java 1.2
- Old interfaces (and many libraries) involve the `Object` type
- Example: `List.add(Object obj)`
- Upcasting eliminates any type-specific information

Type Erasure & Raw Types II

- Java designers made a choice: backwards compatibility over strict typing
- To support backwards compatibility, the Java compiler performs **Type Erasure**
- At compile time, generic type information is erased and treated as a **Raw Type**: an `Object`

Type Erasure & Raw Types III

```
List<Integer> intList = new ArrayList<Integer>();
//becomes
List intList = new ArrayList();
```

- You should not use Raw types yourself!
- Raw types result in runtime exceptions for incompatible types (see previous example)

Open Recursion

- Inside classes, we need a way to reference the object itself
- Open recursion (recursive because the object is referencing itself)
- `this()` allows you to access member methods/variables
- `this(...)` allows you to call another constructor
- `super` allows you to access member methods/variables of the superclass
- `super(...)` allows you to call a constructor of the superclass

Copy Constructors I

- Copy Constructors are preferred to using `Object.clone()` or the `Cloneable` interface; more info:
- All state should be copied
- Immutable objects can be reference-copied
- Mutable objects should be **deep copied**
- Useful methods: `Arrays.copyOf(...)`
Demonstration I

Design a **Student** object and a **Course** object and illustrate usage of:

▶ A copy constructor and deep vs. shallow copy
▶ Bi-directional association (roster and/or schedule)
▶ Potential pitfalls of bi-directional association
▶ Bi-directional association through an intermediate class