GIUSEPPE TURINI

CS-102 COMPUTING AND ALGORITHMS 2
APPENDIX
ADVANCED JAVA AND OOP TOPICS

DATA ABSTRACTION AND PROBLEM SOLVING WITH JAVA, **CHAP 9, 455-504**.

HIGHLIGHTS

Polymorphism in Java

Static Binding, and Dynamic Binding

Violating Data Encapsulation

Encapsulation, Information Hiding, Immutable Objects, and Violation Example

Boxing, Autoboxing, and Unboxing

Abstract Classes, Abstract Methods, and Interfaces

Statics

Overriding and Hiding Methods

Method Signature

Generics

Generic Types and Methods, Bounded Type Parameters

Generics and Inheritance and Subtypes, and Type Inference

Wildcards, Type Erasure, and Restrictions on Generics

Intro to the Java Garbage Collector

POLYMORPHISM IN JAVA 1

Polymorphism: A state of having many shapes, the ability to take on different forms.

Polymorphism (OOP): The capacity of an OOP language (Java) to process objects of various types and classes through a single uniform interface.

Polymorphism in Java has 2 types:

- compile-time polymorphism (**static binding**); or
- runtime polymorphism (**dynamic binding**).

Example: The class **Animal** has a subclass **Cat**. So, any object of type **Cat**, is also of type **Animal**. Therefore, any object of type **Cat** is polymorphic.

Note: It is correct to say that (almost) every object in Java is polymorphic in nature, since all Java classes are derived from the **Object** class.

POLYMORPHISM IN JAVA 2

STATIC BINDING (COMPILE TIME POLYMORPHISM)

In Java, compile time polymorphism is obtained using **method overloading**.

Method Overloading: multiple methods in the same class having the same name but different types/order/number of formal parameters.

Static Binding: At compile time, the Java compiler knows which method to invoke by checking the **method signatures**.

```
class ExampleOverloading {
   public int add(int x, int y) { ... } // Method 1.
   public int add(int x, int y, int z) { ... } // Method 2.
   public int add(double x, int y) { ... } // Method 3.
   ... }
```

POLYMORPHISM IN JAVA 3

DYNAMIC BINDING (RUNTIME POLYMORPHISM)

In Java, runtime polymorphism is obtained using **method overriding**.

Method Overriding: A subclass can provide a specific implementation of a method that is already included in its superclass. When a subclass method has the same signature and return type as the superclass method, the subclass method is said to override the method in the superclass.

Dynamic Binding: At runtime, Java knows which method to invoke by checking **the type of the object** pointed by the reference (used to call a method).

```
Fruit f = new Apple(); f.print(); // Calls print method in Apple class (subclass).
f = new Fruit(); f.print(); // Calls print method in Fruit class (superclass).
// Note: The type or the reference variable f is always Fruit!
```

VIOLATING DATA ENCAPSULATION 1

ENCAPSULATION

"The process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; ***encapsulation*** *serves to separate the contractual interface of an abstraction and its implementation."*
Grady Booch. *Object-Oriented Analysis and Design with Applications. 2007.*

In programming languages, encapsulation refers to one of these distinct notions:

- A mechanism for restricting access to some of the object's components.
- A construct to bundle the data with the methods operating on that data.

See: en.wikipedia.org/wiki/encapsulation_(computer_programming)
See: en.wikipedia.org/wiki/information_hiding

VIOLATING DATA ENCAPSULATION 2

INFORMATION HIDING

Information hiding is the ability to prevent certain aspects of a software component from being accessible to its clients.

In OOP, information hiding reduces software development risk by shifting the code's dependency on an uncertain implementation onto a well-defined interface. Clients of the interface perform operations purely through it so if the implementation changes, the clients do not have to change.

See: en.wikipedia.org/wiki/encapsulation_(computer_programming)
See: en.wikipedia.org/wiki/information_hiding

VIOLATING DATA ENCAPSULATION 3

ENCAPSULATION VS INFORMATION HIDING

The term encapsulation is often used interchangeably with information hiding.

One may think of information hiding as being the principle and encapsulation being the technique. A software module hides information by encapsulating the information into a module or other construct which presents an interface.

See: en.wikipedia.org/wiki/encapsulation_(computer_programming)
See: en.wikipedia.org/wiki/information_hiding

VIOLATING DATA ENCAPSULATION 4

MUTABLE AND IMMUTABLE OBJECTS

An object is **immutable** if its state cannot change after it is constructed. Max reliance on immutable objects is accepted as a sound strategy for creating reliable code.

Immutable objects are useful in **concurrent applications**. They cannot change state, so they cannot be corrupted by thread interference or found in an inconsistent state.

Programmers are often reluctant to employ immutable objects, because they worry about the cost of creating a new object as opposed to updating an object in place. The **impact of object creation is often overestimated**, and can be offset by some of the efficiencies associated with immutable objects.

See: docs.oracle.com/javase/tutorial/essential/concurrency/immutable

VIOLATING DATA ENCAPSULATION 5

EXAMPLE VIOLATING DATA ENCAPSULATION A

```
import java.lang.StringBuilder;
public class ExampleViolatingDataEncapsulation {
   static public void main( String[] args ) {
      // TEST 1
      StringBuilder sb1 = new StringBuilder( "ab" );
      StringBuilder sb2 = new StringBuilder( "xyz" );
      // We print the SortedStringPair object to check if sorting is fine.
      SortedStringPair ssp = new SortedStringPair( sb1, sb2 );
      System.out.println( "1: " + ssp );
      // We try to violate the sorting using the class setter.
      ssp.setShorter( new StringBuilder( "abcd" ) );
      System.out.println( "2: " + ssp );
      // Warning: data encapsulation violated by external code using object references!
      sb1.append( 'c' ); sb1.append( 'd' );
      System.out.println( "3: " + ssp );
      // ...
```

VIOLATING DATA ENCAPSULATION 6

EXAMPLE VIOLATING DATA ENCAPSULATION B

```
        // TEST 2
        StringBuilder sb3 = new StringBuilder( "ab" );
        StringBuilder sb4 = new StringBuilder( "xyz" );
        // We print the SortedStringPair object to check if sorting is fine.
        SortedStringPairSafe ssps = new SortedStringPairSafe( sb3, sb4 );
        System.out.println( "4: " + ssps );
        // We try to violate the sorting using the class setter.
        ssps.setShorter( new StringBuilder( "abcd" ) );
        System.out.println( "5: " + ssps );
        // Warning: trying to violate data encapsulation using object references (1)!
        sb3.append( 'c' ); sb3.append( 'd' );
        System.out.println( "6: " + ssps );
        // Warning: trying to violate data encapsulation using object references (2)!
        StringBuilder sb5 = ssps.getShorter();
        sb5.append( 'c' ); sb5.append( 'd' );
        System.out.println( "7: " + ssps ); }
}
```

VIOLATING DATA ENCAPSULATION 7

SORTED STRING PAIR A

```
import java.lang.StringBuilder; // Note: String is immutable!
public class SortedStringPair {
   private StringBuilder shorter;
   private StringBuilder longer;
   public SortedStringPair( StringBuilder a, StringBuilder b ) {
      if( a.length() > b.length() ) { shorter = b; longer = a; }
      // Note: if both strings have equal length we don't sort.
      else { shorter = a; longer = b; } }

   // Note: we set shorter only if input argument does not violate the sorting.
   public void setShorter( StringBuilder s ) {
      if( s.length() < longer.length() ) { shorter = s; } }
   public StringBuilder getShorter() { return shorter; }

   // ...
```

VIOLATING DATA ENCAPSULATION 8

SORTED STRING PAIR B

```
   // Note: we set longer only if input argument does not violate the sorting.
   public void setLonger( StringBuilder s ) {
      if( s.length() > shorter.length() ) { longer = s; } }
   public StringBuilder getLonger() { return longer; }

   @Override
   public String toString() {
      return "'" + shorter + "' is shorter than '" + longer + "'"; }

}
```

VIOLATING DATA ENCAPSULATION 9

SORTED STRING PAIR SAFE A

```
import java.lang.StringBuilder; // Note: String is immutable!
public class SortedStringPairSafe {
   private StringBuilder shorter;
   private StringBuilder longer;
   public SortedStringPairSafe( StringBuilder a, StringBuilder b ) {
      if( a.length() > b.length() ) {
         shorter = new StringBuilder(b);
         longer = new StringBuilder(a); }
      // Note: if both strings have equal length we don't sort.
      else {
         shorter = new StringBuilder(a);
         longer = new StringBuilder(b); }
   }

   // ...
```

VIOLATING DATA ENCAPSULATION 10

SORTED STRING PAIR SAFE B

```
   // Note: we set shorter only if input argument does not violate the sorting.
   public void setShorter( StringBuilder s ) {
      if( s.length() < longer.length() ) { shorter = new StringBuilder(s); } }
   public StringBuilder getShorter() { return new StringBuilder(shorter); }

   // Note: we set longer only if input argument does not violate the sorting.
   public void setLonger( StringBuilder s ) {
      if( s.length() > shorter.length() ) { longer = new StringBuilder(s); } }
   public StringBuilder getLonger() { return new StringBuilder(longer); }

   @Override
   public String toString() {
      return "'" + shorter + "' is shorter than '" + longer + "'"; }

}
```

AUTOBOXING AND UNBOXING 1

BOXING A

Boxing (aka **wrapping**) is the process of placing a **primitive type** (i.e. a **value type**) within an object so that the primitive can be used as a **reference object**.

Example: In Java, we cannot create a **LinkedList** of **int**, since this class only lists **references** to objects. To circumvent this, an **int** can be boxed into an **Integer** object, and then added to a **LinkedList** of **Integer** (i.e. a **LinkedList<Integer>**).

See: en.wikipedia.org/wiki/object_type_(object-oriented_programming)
See: docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes
See: docs.oracle.com/javase/tutorial/reflect/class
See: docs.oracle.com/javase/8/docs/api/java/util/LinkedList
See: docs.oracle.com/javase/8/docs/api/java/lang/integer

AUTOBOXING AND UNBOXING 2

BOXING B

Note: The boxed object (usually **immutable**) is always a copy of the value object. Unboxing the object also returns a copy of the stored value.

Note: Repeated boxing/unboxing decreases performance: it dynamically allocates new objects (boxing) and then marks them for garbage collection (unboxing).

There is a direct equivalence between an unboxed primitive type and a reference to an immutable boxed object type.

See: en.wikipedia.org/wiki/object_type_(object-oriented_programming)
See: docs.oracle.com/javase/tutorial/java/data/autoboxing
See: docs.oracle.com/javase/tutorial/essential/concurrency/immutable

AUTOBOXING AND UNBOXING 3

BOXING C

Example: We can substitute all primitives in a program with boxed objects. Whereas:

- assignment from one primitive to another will copy its value,
- assignment from one reference to a boxed object to another will copy the reference value to refer to the same object as the first reference.

This is not an issue, because boxed objects are immutable, so there is no semantic difference between 2 references to the same object or to different objects. For all operations other than assignment: unbox the boxed type, perform the operation, and re-box the result as needed. Thus, **it is possible to not store primitive types at all**.

See: en.wikipedia.org/wiki/object_type_(object-oriented_programming)

AUTOBOXING AND UNBOXING 4

AUTOBOXING

Autoboxing is the term for **getting a reference type out of a value type via type conversion** (implicit or explicit). The Java compiler automatically supplies the extra source code which creates the object.

```
Integer i = new Integer(9); // Allocation of a new Integer object storing the value 9.
Integer i = 9; // Autoboxing: implicit type conversion of value type int (9) into
               //             an Integer object, and assignment of its reference to i.

List<Integer> li = new ArrayList<>();
for( int i = 1; i < 50; i += 2 ) {
   li.add(i); } // At runtime, compiler converts this into: li.add( Integer.valueOf(i) );
```

See: en.wikipedia.org/wiki/object_type_(object-oriented_programming)
See: docs.oracle.com/javase/tutorial/java/data/autoboxing

AUTOBOXING AND UNBOXING 5

UNBOXING

Unboxing refers to **getting the value which is associated to a given object via type conversion** (implicit or explicit). Java compiler automatically supplies extra code to retrieve the value out of that object (e.g. by invoking some method on that object).

```
Integer a = new Integer(1); Integer b = new Integer(2);
Integer c = a + b; // Unboxing: a and b are unboxed, their int values summed up,
                   //           and the result is autoboxed into a new Integer object
                   //           (autoboxing via implicit type conversion), and finally
                   //           a reference to this new Integer object is stored in c.
```

Note: Operator **==** cannot be used in this way, since it is already defined for reference types (i.e. equality of the references). To test for equality of values of boxed types: unbox boxed objects and compare their unboxed values, or use **equals** method.

AUTOBOXING AND UNBOXING 6

EXAMPLE AUTOBOXING A

```
// The 8 primitive data types in Java are:
//     byte: 8-bit signed integer in [ -128, 127 ].
//     short: 16-bit signed integer in [ -32768, 32767 ].
//     int: 32-bit signed integer in [ -2^31, 2^31-1 ].
//          In Java SE 8 and later, can be unsigned 32-bit integer in [ 0, 2^32-1 ].
//     long: 64-bit integer. The signed long is in [ -2^63, 2^63-1 ].
//           In Java SE 8 and later, can represent unsigned 64-bit long in [ 0, 2^64-1 ].
//     float: single-precision 32-bit IEEE 754 floating point.
//     double: double-precision 64-bit IEEE 754 floating point.
//     boolean: true or false, and its "size" is not precisely defined.
//     char: single 16-bit Unicode character in [ '\u0000' or 0, '\uffff' or 65535 ].

public class ExampleAutoboxing {
```

AUTOBOXING AND UNBOXING 7

EXAMPLE AUTOBOXING 8

```
static public void f( Object o ) {
   if( o instanceof Byte ) { System.out.println( "Autoboxing in Byte!" ); }
   else if( o instanceof Integer ) { System.out.println( "Autoboxing in Integer!" ); }
   else if( o instanceof Long ) { System.out.println( "Autoboxing in Long!" ); }
   else if( o instanceof Float ) { System.out.println( "Autoboxing in Float!" ); }
   else if( o instanceof Double ) { System.out.println( "Autoboxing in Double!" ); }
   else if( o instanceof Boolean ) { System.out.println( "Autoboxing in Boolean!" ); }
   else if( o instanceof Character ) {
      System.out.println( "Autoboxing in Character!" ); }
   else { System.out.println( "Autoboxing in?" ); } }
```

AUTOBOXING AND UNBOXING 8

EXAMPLE AUTOBOXING C

```
   static public void main( String[] args ) {
      f(-1); // Autoboxing in Integer.
      f(123); // Autoboxing in Integer.
      f(5000000000L); // Autoboxing in Long. Call f(5000000000) is autoboxed in Integer.
      f(123.456); // Autoboxing in Double.
      f(1.4f); // Autoboxing in Float. A call to f(1.4) mean autoboxing in Double...
      f(true); // Autoboxing in Boolean.
      f('a'); // Autoboxing in Character.
      byte b = -1;
      f(b); } // Autoboxing in Byte.

      int i = 42;
      Object o = i; // Autoboxing: value type i autoboxed into reference type Integer.
      int j = o; // Unboxing error! Incompatible types: Object can't be converted to int.
}
```

ABSTRACT CLASSES AND METHODS 1

Abstract Class: A class declared **abstract**. It may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

```
public abstract class MyAbstractClass { ... }
```

Abstract Method: A method declared abstract and providing no implementation.

```
public abstract void myAbstractMethod();
```

See: docs.oracle.com/javase/tutorial/java/iandi/abstract

ABSTRACT CLASSES AND METHODS 2

Note: A class including abstract methods must be declared **abstract**.

Note: When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared **abstract**.

Note: Methods in an interface that are not declared as **default** or **static** are implicitly **abstract**, so the **abstract** modifier is not used with interface methods (i.e. it can be used, but it is unnecessary).

See: docs.oracle.com/javase/tutorial/java/iandi/abstract

ABSTRACT CLASSES AND METHODS 3

ABSTRACT CLASSES COMPARED TO INTERFACES

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation.

However:

- **with abstract classes**, you can declare fields that are not static and not final, and define public/protected/private concrete (i.e. with complete definition) methods;
- **with interfaces**, all fields are automatically public, static, and final, and all methods that you declare/define are public.
- In addition, (in Java) **you can extend (derive) only 1 class**, whether or not it is abstract, whereas **you can implement any number of interfaces**.

See: docs.oracle.com/javase/tutorial/java/iandi/abstract

ABSTRACT CLASSES AND METHODS 4

Consider using abstract classes if:

- you want to share code among several closely related classes;
- classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private);
- you want to declare non-static or non-final fields (e.g. to define methods that can access and modify the state of the object to which they belong).

Consider using interfaces if:

- unrelated classes will implement the interface (e.g. interface **Comparable**);
- you want to specify the behavior of a data type, but not its implementation;
- you want to take advantage of multiple inheritance of type.

See: docs.oracle.com/javase/tutorial/java/iandi/abstract

INTRO TO THE JAVA GARBAGE COLLECTOR 1

The Java Memory Management

Java Memory Management has a built-in Garbage Collector (GC) that allows developers to create new objects without worrying about memory allocation-deallocation, since the GC automatically reclaims unused memory for reuse.

Thanks to the GC, most memory-management issues are solved, but often at the cost of creating serious performance problems. Making garbage collection adaptable to all kinds of situations has led to a complex and hard-to-optimize system.

In order to wrap your head around garbage collection, you need first to understand how memory management works in a Java Virtual Machine (JVM).

See: www.dynatrace.com/resources/ebooks/javabook/how-gc-works

INTRO TO THE JAVA GARBAGE COLLECTOR 2

HOW THE GARBAGE COLLECTION (IN JAVA) WORKS

You may think the GC collects and discards unused (aka dead) objects.
In reality, the GC does the opposite!
Used (aka live) objects are tracked and everything else is marked as garbage.

The Heap: is the area of memory used for dynamic allocation. In most configurations the Operating System (OS) allocates the heap in advance to be managed by the Java Virtual Machine (JVM) while the program is running.

See: www.dynatrace.com/resources/ebooks/javabook/how-gc-works

INTRO TO THE JAVA GARBAGE COLLECTOR 3

Using the Heap

Using the heap has a couple of important ramifications:

- Object creation is faster because global synchronization with the OS is not needed for every single object. **An allocation simply claims some portion of a memory array and moves the offset pointer forward**. The next allocation starts at this offset and claims the next portion of the array.

- When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means **there is no explicit deletion and no memory is given back to the OS**.

See: www.dynatrace.com/resources/ebooks/javabook/how-gc-works

INTRO TO THE JAVA GARBAGE COLLECTOR 4

All objects are allocated on the heap area managed by the JVM. Every item that the developer uses is treated this way, including: class objects, static variables, and even the code itself. As long as an object is being referenced, the JVM considers it alive.

Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory.

As simple as this is, it raises a question: **what is the first reference in the tree?**

To be continued...

See: www.dynatrace.com/resources/ebooks/javabook/how-gc-works