LOYOLA UNIVERSITY CHICAGO

COMP 313 / 413: Intermediate Object-Oriented Programming

Dr. Robert Yacobellis
Advanced Lecturer
Department of Computer Science

Week 4 Topics

- **Generic Types**
- Reverse Engineering Unit Tests
- Introduction to Design Patterns
  - Factory, Strategy, Visitor
  - Examples
- Android Application Development – continued Week 5
  - Setting up Android Studio
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps
- Project 3 and Agile Development – as time permits

LOYOLA UNIVERSITY CHICAGO

Generics in Java

- **Generic programming** is a style of computer programming in which algorithms are written in terms of *to-be-specified-later* types that are instantiated when needed for specific types provided as parameters
- **Generics** are a facility of generic programming that was added to Java in J2SE 5.0. They allow a type (ie, a class or interface) or method (including a constructor) to operate on objects of various types while providing compile-time type safety.
- Java generic hierarchy and classification
  - A type variable is an unqualified identifier (like **T**). Type variables are introduced by generic class declarations, generic interface declarations, generic method declarations, and generic constructor declarations.
  - Classes, interfaces, methods, and constructors are generic if they declare one or more type variables. These type variables are also known as *formal type parameters*.

Source: Wikipedia, Generics_in_Java

LOYOLA UNIVERSITY CHICAGO

A Key Problem Generics Solve

- Consider the following usage of container types from the Collection hierarchy:

```
List myIntList = new LinkedList();                          // 1 – non-generic
myIntList.add(new Integer(0));                              // 2
Integer x = (Integer) myIntList.iterator().next();         // 3 – required cast
```

- The required cast in line 3 not only introduces clutter, it also introduces the possibility of a run time error. **Why?**
- Using generics, the program fragment becomes:

```
List<Integer> myIntList = new LinkedList<Integer>();    // 1' – generic
// List<Integer> myIntList = new LinkedList<>();        // Java 7!
myIntList.add(0);                      // 2' – autoboxing
Integer x = myIntList.iterator().next();          // 3' – no cast
```

- This approach allows <u>compile time</u> type checking of the type of myIntList, so run time usage errors are no longer possible.

Source: J2SE 1.5 Generics Tutorial

Defining Simple Generics

- Excerpts from the Java API List and Iterator interface definitions:

```
public interface List<E> { // E stands for the Elements in the List
    void add(E x);
   Iterator<E> iterator();
}
public interface Iterator<E> {
    E next();
   boolean hasNext();
}
```

- The E's in angle brackets are the declarations of the *formal type parameters* of interfaces List and Iterator.
  - By convention (only), single capital letters are used for formal type parameters:
    T for type, E for element, K for Key, V for value, …
- An *invocation* of the generic type declaration List looks like List<Integer>; in this invocation or *parameterized type*, all occurrences of the formal type parameter E are replaced by the actual *type argument* Integer.
- A generic type declaration is compiled once and turned into a single class file, just like an ordinary class or interface declaration. When a generic declaration is invoked, the actual type arguments are substituted for the formal type parameters.
- Example usage: Set<Lark> exaltation = new HashSet<Lark>();

Generics and Subtyping

- Is the following code snippet legal?

```
List<String>  ls = new ArrayList<String>(); // 1
List<Object> lo = ls;              // 2
```

- Consider:

```
lo.add(new Object());    // 3: attempts to add an arbitrary Object into a String
   List
String s = ls.get(0);    // 4: attempts to assign that arbitrary Object to a String
```

- If statement 2 were legal, we would be able to use statement 3 to insert arbitrary objects into ls, a list of Strings, through its alias lo. Then statement 4 would try to assign an arbitrary object to the String s, which would throw a ClassCastException.
- The Java compiler prevents this from happening because statement 2 will cause a compile time error.
- The general case is that if **Abc** is a subtype (a subclass or subinterface) of **Def**, and **G<E>** is some generic type definition (like List<E> above), then **G<Abc>** is **not** a subtype of **G<Def>** – they are unrelated types.
  - In particular, List<String>, List<Integer>, etc are **not** subtypes of List<Object>.

Wildcards

- If we want to print out all the elements in an arbitrary collection, we might try, using generics and the new enhanced for loop syntax …

```
void printCollection(Collection<Object> c) {
  for (Object e : c) {
        System.out.println(e);
  }
}
```

- The problem with this approach is that it only takes Collection<Object> as a parameter, which is *not* a supertype of all kinds of collections.
- We need a way to specify a collection whose element type matches anything. This is written in the following way, using a *wildcard type*:

```
void printCollection(Collection<?> c) { // the type of collection is unspecified
  for (Object e : c) {      // regardless of the type, it must be an Object
     System.out.println(e);   // invokes possibly overridden toString method
  }
}
```

- We can now call this method with any type of collection!

Generic Methods

- How would we write a method that takes an array of objects and a collection and puts all the array's objects into the collection? To do this we need a *generic method*, one that is parameterized by one or more type parameters. Generic methods can appear in regular or generic classes.

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
  for (T o : a) { // OK if T is the type of a or any of its supertypes (polymorphism)
          c.add(o); // correct →   constrains T to be the "lowest" supertype of a
   and c
  }
} // note: the parameterized type name must be placed just before the return type
```

- We can now call this method with any kind of collection whose element type is a supertype of the element type of the array *(inferred upcasting):*

```
Object[] oa = new Object[100];           // definition 1
Collection<Object> co = new ArrayList<Object>();    // definition 2
fromArrayToCollection(oa, co);    // T inferred to be Object
String[] sa = new String[100];           // definition 3
Collection<String> cs = new ArrayList<String>();    // definition 4
fromArrayToCollection(sa, cs);    // T inferred to be String (definitions 3 & 4)
fromArrayToCollection(sa, co);    // T inferred to be Object (definitions 3 and
    2)
// fromArrayToCollection(oa, cs); // illegal – cannot put Objects from a into
```

LOYOLA UNIVERSITY CHICAGO

Week 4 Topics

- Generic Types
- **Reverse Engineering Unit Tests**
- Introduction to Design Patterns
  - Factory, Strategy, Visitor
  - Examples
- Android Application Development
  - Setting up Android Studio
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps
- Project 3 and Agile Development

How to Reverse Engineer Unit Tests

- Suppose we are given the following JUnit test fragment:

```
final IntComplex c1 = new DefaultIntComplex(1, 0); // note: these must be immutable
final IntComplex c2 = new DefaultIntComplex(0, 1);

assertEquals(1, c1.real());
assertEquals(0, c1.imag());
assertEquals("1+0i", c1.toString());
assertTrue(c1.equals(c1));
assertFalse(c2.equals(c1));
assertFalse(c1.equals(null));
assertTrue(c1.equals(c2.switch()));
```

- How could we reverse-engineer an IntComplex interface that is consistent with this test sequence, leaving out Object methods?
  - That interface must declare non-Object methods real, imag, and switch:

```
interface IntComplex {
  int real();
  int imag();
  IntComplex switch(); // switch can return any type that implements IntComplex
} // we could then proceed to fully define an implementing class DefaultIntComplex
```

How to Reverse Engineer Unit Tests

```
final IntComplex c1 = new DefaultIntComplex(1, 0);
final IntComplex c2 = new DefaultIntComplex(0, 1);

assertEquals(1, c1.real());
assertEquals(0, c1.imag());
assertEquals("1+0i", c1.toString());
assertTrue(c1.equals(c1));
assertFalse(c2.equals(c1));
assertFalse(c1.equals(null));
assertTrue(c1.equals(c2.switch()));
```

```
interface IntComplex {
   int real();
   int imag();
   IntComplex switch();
}
```

```
// changes: c1 and c2 declarations, first two asserts:
final Complex<Integer> c1 = new
DefaultComplex<Integer>(1, 0);
final Complex<Integer> c2 = new
DefaultComplex<Integer>(0, 1);
assertEquals(1, c1.real().intValue());
assertEquals(0, c1.imag().intValue());
```

- How can we generalize the IntComplex interface by making it *generic* in the real and imaginary types (eg, they could be float)?
  - Let's call that generic interface just Complex:

```
interface Complex<T> {
   T real();
   T imag();
   Complex<T> switch(); // switch can return any type that implements Complex<T>
} // we could then proceed to fully define an implementing class DefaultComplex
```

- We can now redefine c1 and c2 in the test fragment using the generic types Complex and Default Complex – that's almost all that needs to change in the test fragment (see above) – **Why??**

LOYOLA UNIVERSITY CHICAGO

Week 4 Topics

- Generic Types
- Reverse Engineering Unit Tests
- **Introduction to Design Patterns**
  - Factory, Strategy, Visitor
  - Examples
- Android Application Development
  - Setting up Android Studio
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps
- Project 3 and Agile Development

Introduction to Design Patterns

- Design Pattern "Bob Tarr" slides are on Sakai in Week 4
  - For more information on patterns in Java, also see:
    http://www.patterndepot.com/put/8/JavaPatterns.htm
    http://home.earthlink.net/~huston2/dp/strategy.html
- A suggested Design Pattern book is Design Patterns in Java (DPiJ) by Metsker and Wake
  - DPiJ shows how to refactor programs to use patterns

Note: there are some differences between the Bob Tarr slide materials, which match the "Gang of Four" original Design Patterns book, and the DPiJ book – there's a summary of that plus more design pattern info at the end of this set of slides

Design Pattern Overview

- A Design Pattern is "a solution to a problem in a context" (Tarr); it is also "a way of doing something: a way of pursuing an intent, a technique" and "[a] distillation of accumulated wisdom" (DPiJ)
- The Design Patterns we'll cover in this course provide solutions to general design problems in particular contexts, which is a middle level of abstraction for solutions (Tarr)
  - They do <u>not</u> provide designs/solutions for either an entire application/subsystem or for a specific class such as linked list or hash table (like the Java Collections Framework)
- "A design pattern names, abstracts, and identifies key aspects of a common design structure that makes it useful for creating a reusable object-oriented design." (Tarr / GoF). It provides "[a] class- and method-level solution to common problems in object-oriented design" (DPiJ)

  http://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm

We'll review problems each pattern addresses as we go.

Design Pattern Classification

- The Bob Tarr material is based on the landmark "Gang of Four" book Design Patterns (DP or GoF) by Gamma, Helm, Johnson, and Vlissides, which classifies patterns based on their *purpose*:
  - **Creational patterns** – relate to the process of object creation
  - **Structural patterns** – deal with composition of classes/objects
  - **Behavioral patterns** – deal with interactions of classes/objects

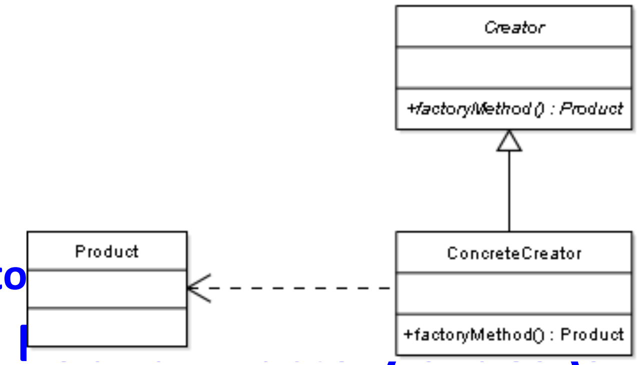
Week 4 Topics

- Generic Types
- Reverse Engineering Unit Tests
- **Introduction to Design Patterns**
  - **Factory**, Strategy, Visitor
  - Examples
- Android Application Development
  - Setting up Android Studio
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps
- Project 3 and Agile Development

Factory Method (Creational)

From Wikipedia, the free encyclopedia

- The **factory method pattern** is an object-oriented design pattern. Like other creational patterns, it deals with the problem of **creating objects (products) without specifying the exact class of object that will be created.** The factory method design pattern handles this problem by defining a separate method for creating the objects, whose subclasses can be overridden to specify the derived type of product that will be created. More generally, the term *factory method* is often used to refer to any method whose main purpose is creation of objects.
- The essence of the Factory Pattern is to **"Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses."**



http://www.tuto ctory_pattern.htm

→ es 1-9)

Week 4 Topics

- Generic Types
- Reverse Engineering Unit Tests
- **Introduction to Design Patterns**
  - Factory, **Strategy**, Visitor
  - Examples
- Android Application Development
  - Setting up Android Studio
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps
- Project 3 and Agile Development

The Strategy Pattern (Behavioral)

From Wikipedia, the free encyclopedia

- In computer programming, the **strategy pattern** (also known as the **policy pattern**) is a particular software design pattern, whereby **algorithms can be selected at runtime.**
- In some programming languages, such as those without polymorphism, the issues addressed by this pattern are handled through forms of reflection, such as the native function pointer or function delegate syntax.
- <u>The strategy pattern is useful for situations where it is necessary to dynamically swap the algorithms used in an application.</u> The strategy pattern is intended to:

1) provide a means to define a family of algorithms,
2) encapsulate each one as an object, and
3) make them interchangeable.

- **The strategy pattern lets the algorithms vary independently from clients that use them.**

DPiJ Operations Patterns (Ch. 21)

- DPiJ says that operations patterns like Strategy "implement an operation in methods across several classes", and goes on to say that (based on UML concepts) …
  - An **operation** is a <u>specification of a service</u> that can be requested from an instance of a class, eg, "stringify yourself"
  - A **method** like **toString()** is an <u>implementation of an operation</u>
- An operation is thus a level of abstraction up from the concept of a method
- The concept of an *operation* relates to the idea that <u>methods in different classes can have the same interface</u> but implement that interface in different ways
  - Operations patterns address contexts where you need more than one method with the same interface as part of a design

The Strategy Pattern

- If you intend to encapsulate an operation, making implementations interchangeable, use the Strategy Pattern
  - A strategy is a plan or approach for achieving an aim, given certain input conditions, similar in some ways to an algorithm; if multiple strategies appear in a program, the code may become complex
  - In the Strategy Pattern alternative approaches or strategies are encapsulated in separate classes that each implement a common operation or *interface*

http://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

→ **Bob Tarr Strategy slides (State & Strategy slides 37-42)**

Week 4 Topics

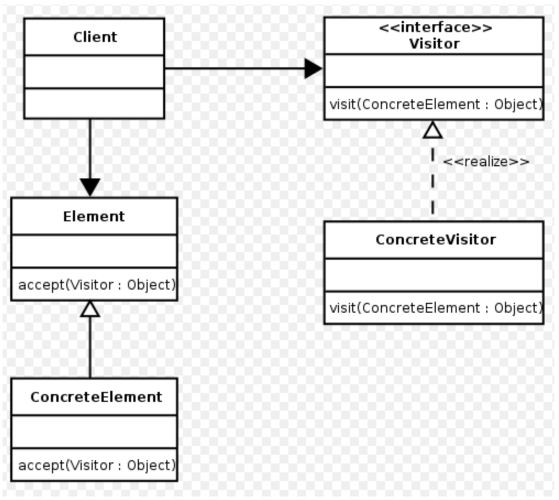
- Generic Types
- Reverse Engineering Unit Tests
- **Introduction to Design Patterns**
  - Factory, Strategy, **Visitor**
  - Examples
- Android Application Development
  - Setting up Android Studio
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps
- Project 3 and Agile Development

LOYOLA UNIVERSITY CHICAGO

Visitor Design Pattern (Behavioral)

From Wikipedia, the free encyclopedia

- In object-oriented programming and software engineering, **the visitor design pattern is a way of separating an algorithm from an object structure upon which it operates.** A practical result of this separation is **the ability to add new operations to existing object structures without modifying those structures**. Thus, using the visitor pattern helps conformance with the open/closed principle.
- In essence, **the visitor allows one to add new virtual functions (abstract methods in Java) to a family of classes without modifying the classes themselves**; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.
- While powerful, the visitor pattern is more limited than conventional virtual functions. It is not possible to create visitors for objects in Java without adding a small callback method inside each class and the callback method in each of the classes is not inheritable.



DPiJ Extension Patterns (Ch. 26) and the Visitor Pattern (Ch. 29)

- DPiJ says Extension patterns like Visitor allow you to add new behavior to existing or legacy classes
- In particular, **the intent of the Visitor pattern is to let you define a new operation for an existing class hierarchy without changing the hierarchy classes**
  - Visitor is often layered over a Composite, which we covered
- Mechanics of the Visitor pattern:
  - **Add an accept() operation to some or all of the classes in the class hierarchy**; every implementation of this method will accept an argument whose type is an interface that you create.
  - **Define that interface with a set of operations that share a common name, usually visit(),** but that have different argument types; declare one such operation for each class in the hierarchy for which you will allow extensions (that is, each of those classes is an argument type).

http://www.tutorialspoint.com/design_pattern/visitor_pattern.htm

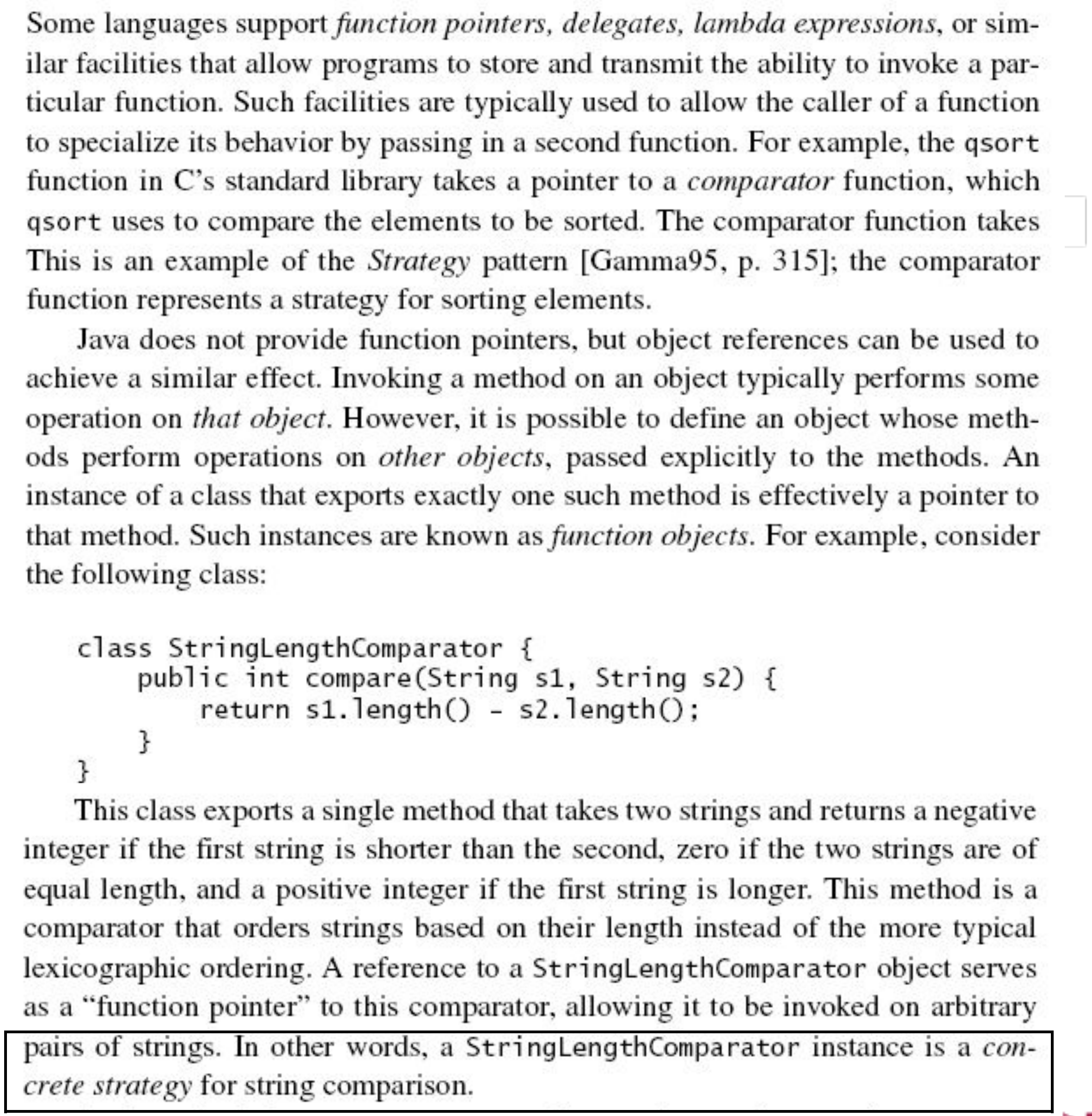
→ **Bob Tarr Visitor slides (1-12)**

Week 4 Topics

- Generic Types
- Reverse Engineering Unit Tests
- Introduction to Design Patterns
    - Factory, Strategy, Visitor
    - Examples
- **Android Application Development – continued Week 5**
    - Setting up Android Studio
    - Running some simple apps
    - Android framework & activity life cycle
    - More complex apps
- Project 3 and Agile Development

Supplemental Design Pattern Slides

Bloch's View of the Strategy Pattern



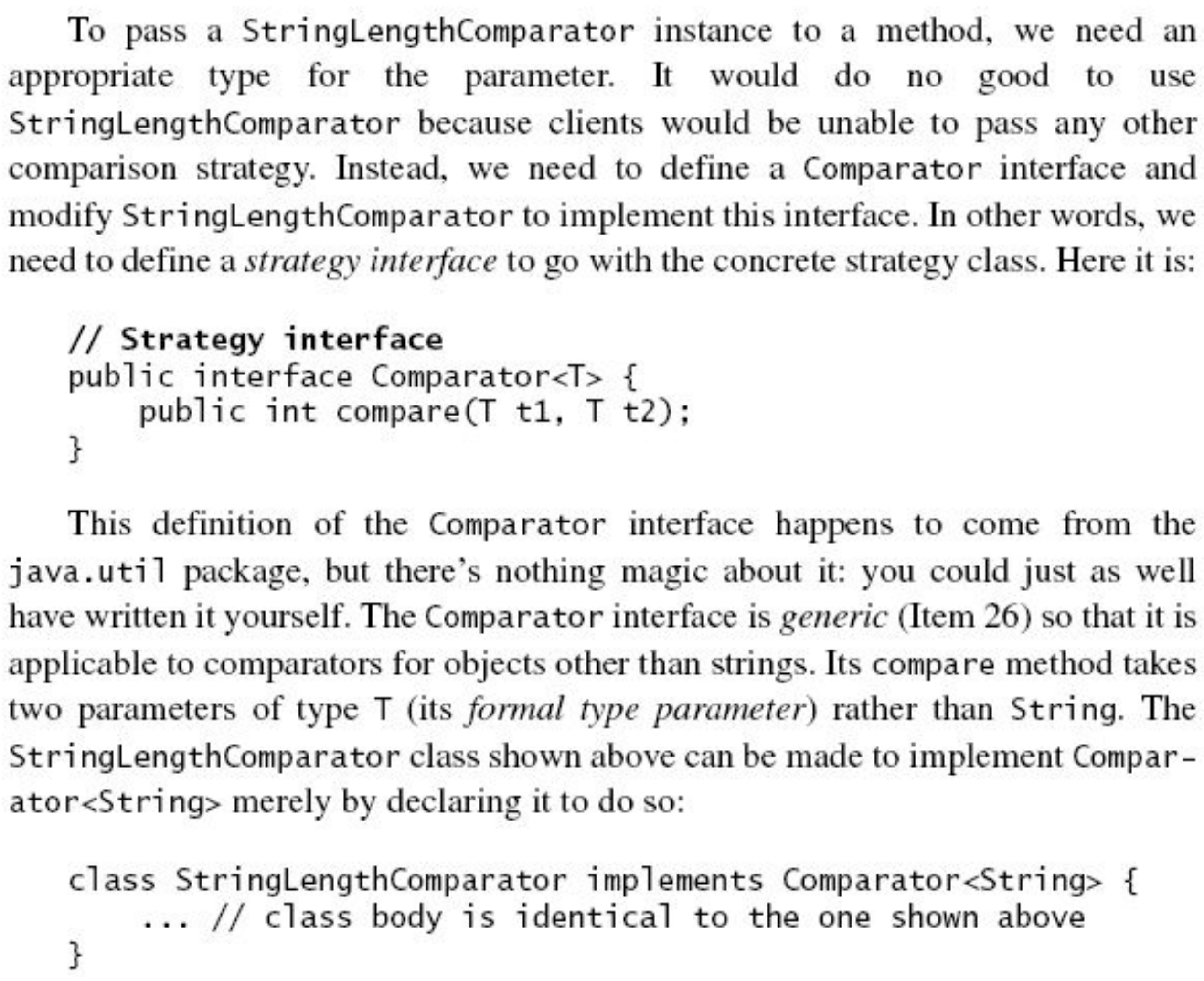
Some languages support *function pointers, delegates, lambda expressions*, or similar facilities that allow programs to store and transmit the ability to invoke a particular function. Such facilities are typically used to allow the caller of a function to specialize its behavior by passing in a second function. For example, the `qsort` function in C's standard library takes a pointer to a *comparator* function, which `qsort` uses to compare the elements to be sorted. The comparator function takes This is an example of the *Strategy* pattern [Gamma95, p. 315]; the comparator function represents a strategy for sorting elements.

Java does not provide function pointers, but object references can be used to achieve a similar effect. Invoking a method on an object typically performs some operation on *that object*. However, it is possible to define an object whose methods perform operations on *other objects*, passed explicitly to the methods. An instance of a class that exports exactly one such method is effectively a pointer to that method. Such instances are known as *function objects*. For example, consider the following class:

```
class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

This class exports a single method that takes two strings and returns a negative integer if the first string is shorter than the second, zero if the two strings are of equal length, and a positive integer if the first string is longer. This method is a comparator that orders strings based on their length instead of the more typical lexicographic ordering. A reference to a `StringLengthComparator` object serves as a "function pointer" to this comparator, allowing it to be invoked on arbitrary pairs of strings. In other words, a `StringLengthComparator` instance is a *concrete strategy* for string comparison.

Bloch’s View of the Strategy Pattern



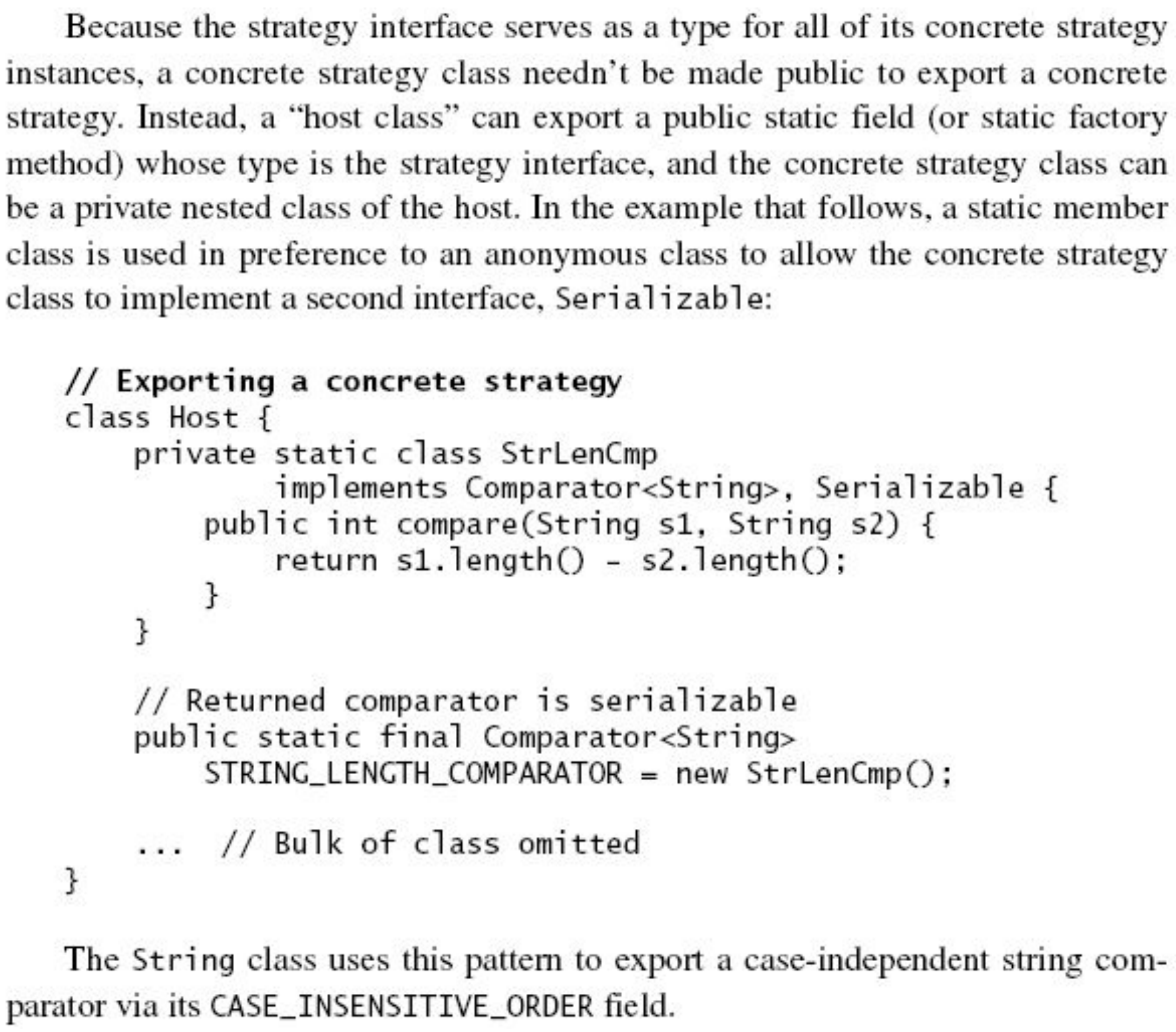
To pass a `StringLengthComparator` instance to a method, we need an appropriate type for the parameter. It would do no good to use `StringLengthComparator` because clients would be unable to pass any other comparison strategy. Instead, we need to define a `Comparator` interface and modify `StringLengthComparator` to implement this interface. In other words, we need to define a *strategy interface* to go with the concrete strategy class. Here it is:

```
// Strategy interface
public interface Comparator<T> {
    public int compare(T t1, T t2);
}
```

This definition of the `Comparator` interface happens to come from the `java.util` package, but there’s nothing magic about it: you could just as well have written it yourself. The `Comparator` interface is *generic* (Item 26) so that it is applicable to comparators for objects other than strings. Its `compare` method takes two parameters of type `T` (its *formal type parameter*) rather than `String`. The `StringLengthComparator` class shown above can be made to implement `Comparator<String>` merely by declaring it to do so:

```
class StringLengthComparator implements Comparator<String> {
    ... // class body is identical to the one shown above
}
```

Bloch’s View of the Strategy Pattern



Because the strategy interface serves as a type for all of its concrete strategy instances, a concrete strategy class needn’t be made public to export a concrete strategy. Instead, a “host class” can export a public static field (or static factory method) whose type is the strategy interface, and the concrete strategy class can be a private nested class of the host. In the example that follows, a static member class is used in preference to an anonymous class to allow the concrete strategy class to implement a second interface, `Serializable`:

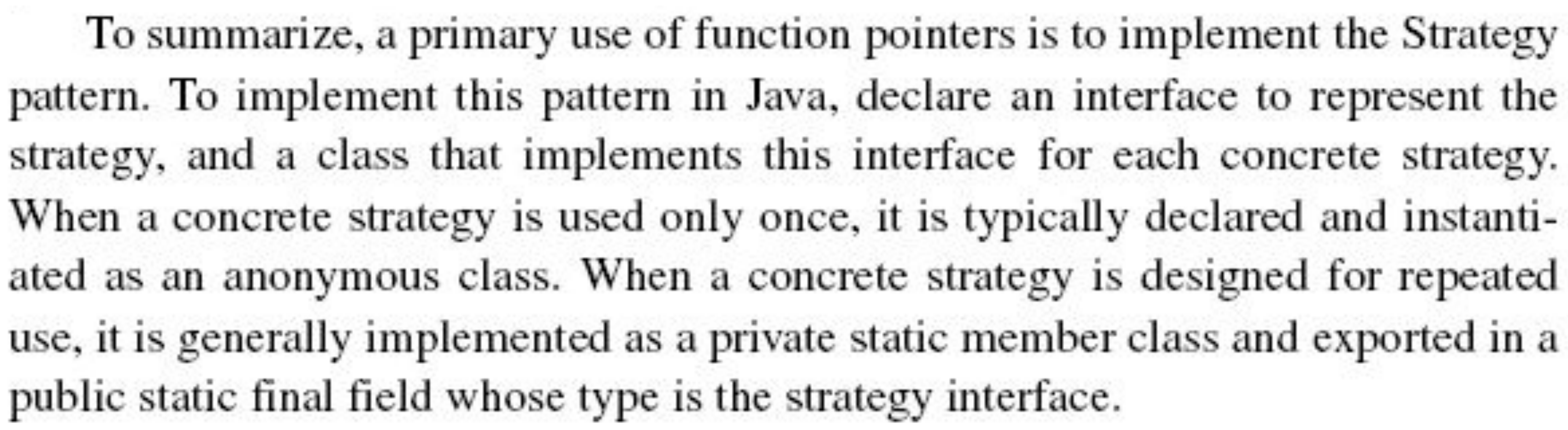
```
// Exporting a concrete strategy
class Host {
    private static class StrLenCmp
            implements Comparator<String>, Serializable {
        public int compare(String s1, String s2) {
            return s1.length() - s2.length();
        }
    }

    // Returned comparator is serializable
    public static final Comparator<String>
        STRING_LENGTH_COMPARATOR = new StrLenCmp();

    ...  // Bulk of class omitted
}
```

The `String` class uses this pattern to export a case-independent string comparator via its `CASE_INSENSITIVE_ORDER` field.

Bloch’s View of the Strategy Pattern



To summarize, a primary use of function pointers is to implement the Strategy pattern. To implement this pattern in Java, declare an interface to represent the strategy, and a class that implements this interface for each concrete strategy. When a concrete strategy is used only once, it is typically declared and instantiated as an anonymous class. When a concrete strategy is designed for repeated use, it is generally implemented as a private static member class and exported in a public static final field whose type is the strategy interface.

Note: we’ll talk about anonymous classes in a future lecture

Visitor Pattern “Double Dispatch”

- Consider the children’s game Rock, Paper, Scissors:
  - Rock breaks Scissors
  - Paper covers Rock
  - Scissors cut Paper
- Let’s look at an implementation of this game using “double dispatch” from the Visitor design pattern (thanks to Bruce Eckel, *Thinking in Java*, 4th Ed.)
  - Java only performs “single dispatch” – it can only invoke dynamic method binding on one type at a time
  - To get *n*-fold dispatch, you have to have *n* method calls (eg, 2 method calls for double dispatch)
  - In the RPS game, these 2 methods are compete() and eval() (in Visitor: accept() and visit())

Rock, Paper, Scissors (RPS)

Outcome.java

```
public enum Outcome { WIN, LOSE, DRAW }
```

RPS.java;

```
import java.util.*;
import static Outcome.*; // enum values are static

interface Item { // Note: could be an abstract class*
 Outcome compete(Item it);   // type #1 resolution
 Outcome eval(Paper p);      // type #2 resolution
 Outcome eval(Scissors s);   // type #2 resolution
 Outcome eval(Rock r);       // type #2 resolution
}

class Paper implements Item { // one of the competitors
❶     public Outcome compete(Item it) { return it.eval(this); }
❶, ❷  public Outcome eval(Paper p) { return DRAW; }
       public Outcome eval(Scissors s) { return WIN; }
       public Outcome eval(Rock r) { return LOSE; }
       public String toString() { return "Paper"; }
} // * compete() definitions could be factored out

class Scissors implements Item { // the second competitor
 public Outcome compete(Item it) { return it.eval(this); }
 public Outcome eval(Paper p) { return LOSE; } // etc …
 public String toString() { return "Scissors"; }
}

❷ class Rock implements Item { // the third competitor
   public Outcome compete(Item it) { return it.eval(this); }
❷, ❸ public Outcome eval(Paper p) { return WIN; } // etc …
   public String toString() { return "Rock"; }
}

public class RPS { // the Rock-Paper-Scissors game
 static final int SIZE = 20;
 private static Random rand = new Random(47);
 public static Item newItem() { // create a random item
   switch(rand.nextInt(3)) {
     case 0: return new Scissors(); // upcast to Item
     case 1: return new Paper();    // upcast to Item
     case 2: return new Rock();     // upcast to Item
   }
 } // this is a static factory method to create an Item

public static void match(Item a, Item b) {
   System.out.println( a + " vs. " + b + ": " + a.compete(b) ); ❶
 } // dynamic type #1 "compete()" resolution at run time
   // dynamic type #2 "eval()" call is resolved in type b
                                                    ❷
 public static void main(String[] args) {
   for(int i = 0; i < SIZE; i++)    ❶          ❷          ❸
     match(newItem(), newItem()); // eg, Paper vs. Rock: WIN
 }
}
```

Design Pattern Classification

- The Bob Tarr material is based on the landmark "Gang of Four" book Design Patterns (DP or GoF) by Gamma, Helm, Johnson, and Vlissides, which classifies patterns based on their *purpose*:
  - **Creational patterns** – relate to the process of object creation
  - **Structural patterns** – deal with composition of classes/objects
  - **Behavioral patterns** – deal with interactions of classes/objects
- The DPiJ instead classifies patterns into *intent*-oriented categories:
  - **Construction patterns** – essentially the same as Creational in DP/Bob Tarr
  - **Interface patterns** – work with class interfaces or interfaces to objects – covers most of the DP/Bob Tarr Structural patterns
  - **Extension patterns** – alter the way objects, collections, and class hierarchies operate – overlaps with Structural & Behavioral in DP/Bob Tarr
  - **Operation patterns** – provide services in different ways – overlaps with Behavioral in DP/Bob Tarr
  - **Responsibility patterns** – move responsibility to a central object/intermediary

LOYOLA UNIVERSITY CHICAGO

Design Pattern Classification

Pattern	Bob Tarr/GoF	DPiJ
Iterator	Behavioral	Extension
Observer	Behavioral	Responsibility
State	Behavioral	Operation
Strategy	Behavioral	Operation
Visitor	Behavioral	Extension
Abstract Factory	Creational	Construction
Builder	Creational	Construction
Factory Method	Creational	Construction
Prototype	Creational	Construction
Singleton	Creational	Responsibility
Adapter	Structural	Interface
Composite	Structural	Interface
Decorator	Structural	Extension
Façade	Structural	Interface

We'll cover many of these in class

LOYOLA UNIVERSITY CHICAGO

Additional DPiJ Patterns

Pattern	DPiJ
Bridge	Interface
Mediator	Responsibility
Proxy	Responsibility
Chain of Responsibility	Responsibility
Flyweight	Responsibility
Memento	Construction
Template Method	Operation
Command	Operation
Interpreter	Operation

We may cover some of these in class as appropriate

Tarr/GoF vs. DPiJ Design Patterns

Tarr/GoF Purpose	Patterns
Behavioral	Iterator, Observer, State, Strategy, Visitor
Creational	Abstract Factory, Builder, Factory Method, Iterator, Prototype, Singleton
Structural	Adapter, Composite, Decorator, Façade

DPiJ Intent	Patterns
Interfaces	Adapter, Façade, Composite, Bridge
Responsibility	Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight
Construction	Builder, Factory Method, Abstract Factory, Prototype, Memento
Operations	Template Method, State, Strategy, Command, Interpreter
Extensions	Decorator, Iterator, Visitor

LOYOLA UNIVERSITY CHICAGO

Week 4 Topics

- Generic Types
- Reverse Engineering Unit Tests
- Introduction to Design Patterns
  - Factory, Strategy, Visitor
  - Examples
- Android Application Development
  - Setting up Android Studio
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps
- Project 3 and Agile Development

LOYOLA UNIVERSITY CHICAGO