CS 2420

Introduction to Algorithms & Data Structures

LECTURE 4

Generic programming

Java generics

Function objects

Review: Lec3

- What is method overriding and how is it used?
- What is an example of polymorphism at work?
- What are the visibility rules for inheritance?
- What is an `abstract` method? an `abstract` class?
- How is an `interface` different from any `abstract` class?

Generic Programming

- A style of computer programming where algorithms
  - are written in an extended grammar
  - are made adaptable by specifying variable parts
  - these parts are then somehow instantiated later with respect to the base grammar
- A *grammar* is simply the rules of a language.
- *Example*: `Shape A` is “bigger” than `Shape B`
  - because `A`'s area is larger than `B`'s
  - because `A`'s perimeter is larger than `B`'s
  - ...

Genericity

- The *generic* mechanism supports code reuse.
- If the implementation is identical except for the basic type of the object, a *generic implementation* can be used to describe basic functionality. (*example*: sorting)
- Generic classes and methods in Java (since 5.0) are similar to templates in C++.
- Inheritance is used (even pre-Java 5) to implement generic programs.

Using `Object` for Genericity

- A generic class can be implemented using the appropriate superclass, such as `Object`.
- Every class has `Object` as a superclass.

```
public class SimpleArrayList {
  public int size() {
    return theSize;
  }
  public Object get(int index) {
    if(index < 0 || index >= theSize)
      throw new ArrayIndexOutOfBoundsException();
    return theItems[index];
  }
  public boolean add(Object x) {...}
  private int theSize = 0;
  private Object[] theItems = new Object[10];
}
```

Wrapper Classes

- One problem—primitive types are not `Object`-derived.
- Will this work?

```
SimpleArrayList a = new SimpleArrayList();
if(a.add(53))
  int i = a.get(0);
```

- A *wrapper class* stores an entity (the wrapee) and adds operations that the wrapee's type does not support.
- Java provides wrapper classes for each primitive type (`Integer` for `int`).
- Wrapper classes are compatible with `Object`.

Auto-Boxing and Auto-Unboxing

- In Java, if an `int` is passed in a place where an `Integer` is required, the compiler will insert a call to the `Integer` constructor behind the scenes.
  - Similarly, if an `Integer` is passed where an `int` is required, the compiler will insert a call to `intValue`.
- The former is *auto-boxing*, the latter *auto-unboxing*.
  - Also works for other seven primitive/wrapper pairs.
- Yes, this will work:

```
SimpleArrayList a = new SimpleArrayList();
if(a.add(53))
  int i = a.get(0);
```

Using Interface Types for Genericity

- Another problem—using `Object` as a generic type works only if method needed is in `Object` class.
- Be more specific and use an interface type.
- *For example*: Comparisons can be made using the `compareTo` method in classes implementing the `Comparable` interface.
- If there is an attempt to compare two incompatible objects, the `compareTo` method will throw a `ClassCastException`.

Generic Classes

- Java 5 provides generic classes, such as `ArrayList`.
- To use, simply put the desired reference type in `<>`.

```
ArrayList<Point> a = new ArrayList<Point>();
```

- To write a generic class, include one or more type parameters in `<>` after the class name.

```
public class GenericClass<AnyType> {
  public AnyType getData() {
    return data;
  }
  public void setData(AnyType x) {
    data = x;
  }
  private AnyType data;
}
```

Generic `static` Methods

- `static` methods can have their own type parameter list.
- Such types do not apply to the rest of the class.
- To write a generic method, include one or more type parameters in `<>` just before the return type.

```
public static <T> boolean contains(T[] a, T x) {
  for(T val : a)
    if(x.equals(val)
      return true;
  return false;
}
```

- To use a generic method, no need to specify type in `<>`.

Type Bounds

- Suppose we have

  ```
  public void m(ArrayList<Shape> a) {...}
  ```

  What happens if we pass an `ArrayList<Circle>`?

- In using a generic class or method, we can be more general about the actual type to be used.

- `... m(ArrayList<? extends Shape> a) {...`

  allows passing `ArrayList` of anything that is a `Shape`.

- `... m(ArrayList<? super Circle> a) {...`

  allows what?

Java's Comparable Interface

```
public interface Comparable<Type> {
   int compareTo(Type o);
}

public class Shape implements Comparable<Shape> {
  ...
  public int compareTo(Shape o) {
    return area() - o.area();
  }
}

public class ShapeTest {
  public static void main(String[] args) {
    Circle a; Square b;
    ...
    if (a.compareTo(b) < 0)
      System.out.println(“Shape b is bigger than a”);
...
```

Function Objects

- What should `findMax` do for `Shape`?
- Should it compare areas, perimeters, ...?
- One solution would be to pass a function to `findMax` that performs the desired comparison.
- Java does not allow functions as parameters, but we can embed a function in an object and pass a reference to it.
- Such an object is known as a *function object* (or a *functor*).
- A function object contains just one method and no data.

Java's Comparator Interface

For each comparison, a new class contains a different implementation of the agreed-up single method. An interface declares the signature of the method.

```
public interface Comparator<Type> {
  int compare(Type lhs, Type rhs);
}

public class OrderByArea implements Comparator<Shape> {
  public int compare(Shape s1, Shape s2) {
    return s1.area() - s2.area();
}}

public class OrderByPerimeter implements Comparator<Shape> {...}

public class Util {
  public static Shape findMax(Shape[] a, Comparator<Shape> c) {
    int maxIndex = 0;
    for(int i = 1; i < a.length; i++)
      if(c.compare(a[i], a[maxIndex]) > 0)
        maxIndex = i;
    return a[maxIndex];
}}
```