School of Computing

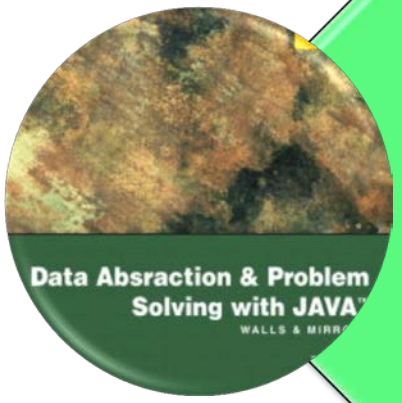
CS1020 Data Structures and Algorithms I
Lecture Note #5

Generics

Objective

Generics: Allowing operations not be to tied to a specific data type.

References



Book

- **Generics:** Chapter 9, Section 9.4, pages 499 to 507

CS1020 website → Resources → Lectures

- http://www.comp.nus.edu.sg/~cs1020/2_resources/lectures.html

Outline

0. Recapitulation
1. Generics
   - 1.1 Motivation
   - 1.2 Example: The **IntPair** Class (non-generic)
   - 1.3 The Generic **Pair** Class
   - 1.4 Autoboxing/unboxing
   - 1.5 The Generic **NewPair** Class
   - 1.6 Summary

0. Recapitulation

- We explored OOP concepts learned in week 2 in more details (constructors, overloading methods, class and instance methods).
- In week 3, we learned some new OOP concepts (encapsulation, accessors, mutators, “this” reference, overriding methods)
- UML was introduced to represent OO components

1 Generics

Allowing operation on objects of various types

1.1 Motivation

- There are programming solutions that are applicable to a wide range of different data types
  - The code is exactly the same other than the data type declarations
- In C, there is no easy way to exploit the similarity:
  - You need a separate implementation for each data type
- In Java, you can make use of **generic programming**:
  - A mechanism to specify solution without tying it down to a specific data type

1.2 Eg: The IntPair Class (non-generic)

- Let's define a class to:
  - Store a pair of integers, e.g. (**`74, -123`**)
  - Many usages, can represent 2D coordinates, range (min to max), height and weight, etc.

IntPair.java

```
class IntPair {

  private int first, second;

  public IntPair(int a, int b) {
    first = a;
    second = b;
  }

  public int getFirst() { return first; }
  public int getSecond() { return second; }
}
```

1.2 Using the IntPair Class (non-generic)

```
// This program uses the IntPair class to create an object
// containing the lower and upper limits of a range.
// We then use it to check that the input data fall within
// that range.
import java.util.Scanner;
public class TestIntPair {

  public static void main(String[] args) {

    IntPair range = new IntPair(-5, 20);
    Scanner sc = new Scanner(System.in);
    int input;

    do {
      System.out.printf("Enter a number in (%d to %d): ",
                        range.getFirst(), range.getSecond());

      input = sc.nextInt();

    } while( input < range.getFirst() ||
             input > range.getSecond() );
  }
}
```

```
Enter a number in (-5 to 20): -10
Enter a number in (-5 to 20): 21
Enter a number in (-5 to 20): 12
```

TestIntPair.java

1.2 Observation

- The **`IntPair`** class idea can be easily extended to other data types:
  - **`double`**, **`String`**, etc.
- The resultant code would be almost the same!

```
class StringPair {
   private String first, second;

   public StringPair( String a, String b ) {
     first = a;
     second = b;
   }

   public String getFirst() { return first; }
   public String getSecond() { return second; }

}
```

Only differences are the data type declarations

1.3 The Generic Pair Class

```
class Pair <T> {

  private T first, second;

  public Pair(T a, T b) {
    first = a;
    second = b;
  }

  public T getFirst() { return first; }
  public T getSecond() { return second; }

}
```

Pair.java

- Important restriction:
  - The generic type can be substituted by reference data type only
  - Hence, primitive data types are NOT allowed
  - Need to use wrapper class for primitive data type

1.3 Using the Generic Pair Class

TestGenericPair.java

```
public class TestGenericPair {

  public static void main(String[] args) {

    Pair<Integer> twoInt = new Pair<Integer>(-5, 20);
    Pair<String> twoStr = new Pair<String>("Turing", "Alan");

    // You can have pair of any reference data types!
     // Print out the integer pair
    System.out.println("Integer pair: (" + twoInt.getFirst()
                       + ", " + twoInt.getSecond() + ")";
     // Print out the String pair
    System.out.println("String pair: (" + twoStr.getFirst()
                       + ", " + twoStr.getSecond() + ")";
  }
}
```

- The formal generic type **<T>** is substituted with the actual data type supplied by the user:
  - The effect is similar to generating a new version of the **Pair** class, where **T** is substituted

1.4 Autoboxing/unboxing (1/2)

- The following statement invokes autoboxing

```
Pair<Integer> twoInt = new Pair<Integer>(-5, 20);
```

- **`Integer`** objects are expected for the constructor, but -5 and 20, of primitive type **`int`**, are accepted.
- ***Autoboxing*** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes
  - The primitive values -5 and 20 are converted to objects of **`Integer`**
- The Java compiler applies autoboxing when a primitive value is:
  - Passed as a parameter to a method that expects an object of the corresponding wrapper class
  - Assigned to a variable of the correspond wrapper class

1.4 Autoboxing/unboxing (2/2)

- Converting an object of a wrapper type (e.g.: **`Integer`**) to its corresponding primitive (e.g: **`int`**) value is called unboxing.
- The Java compiler applies unboxing when an object of a wrapper class is:
  - Passed as a parameter to a method that expects a value of the corresponding primitive type
  - Assigned to a variable of the corresponding primitive type

```
int i = new Integer(5); // unboxing
Integer intObj = 7;     // autoboxing
System.out.println("i = " + i);
System.out.println("intObj = " + intObj);
```

```
i = 5
intObj = 7
```

```
int a = 10;
Integer b = 10;     // autoboxing
System.out.println(a == b);
```

```
true
```

1.5 The Generic NewPair Class

- We can have more than one generic type in a generic class
- Let's modify the generic pair class such that:
  - Each pair can have two values of **different data types**

```
class NewPair <S,T> {

  private S first;
  private T second;

  public NewPair(S a, T b) {
    first = a;
    second = b;
  }

  public S getFirst() { return first; }
  public T getSecond() { return second; }

}
```

You can have multiple generic data types.

Convention: Use single uppercase letters for generic data types.

NewPair.java

1.5 Using the Generic NewPair Class

TestNewGenericPair.java

```
public class TestNewGenericPair {

  public static void main(String[] args) {

    NewPair<String, Integer> someone =
      new NewPair<String, Integer>("James Gosling", 55);

    System.out.println("Name: " + someone.getFirst());
    System.out.println("Age: " + someone.getSecond());
  }
}
```

```
Name: James Gosling
Age: 55
```

- This NewPair class is now very flexible!
  - Can be used in many ways

1.6 Summary

- Caution:
  - Generics are useful when the code remains unchanged other than differences in data types
  - When you declare a generic class/method, make sure that the code is valid for all possible data types
- Additional Java Generics topics (not covered):
  - Generic methods
  - Bounded generic data types
  - Wildcard generic data types

End of file