Parameterization

John Ross Wallrabenstein
Assignments

• Quizzes are graded - handed back after class
  • Turnaround from now on ~1-3 days
• Project 1 - graded by next Friday
  • Project turnaround within 2 weeks
• Upcoming Test
  • I will stay after for test questions / review
Generics

- We can reuse code when the structure and operations are not dependent on the data represented.
- This is also referred to as Parameterized Classes, or Generics.
Motivation

- Consider the Queue example given in lecture
- Do we really need a queue class for every type, or is the structure sufficient?
Generics

• What are some examples of structures that do not depend* on the type of data they store?

• Are you already familiar with any generic classes?

*For the most part
Abstraction Overview

- Interface - a skeleton for a class design
- Inheritance - specialized classes use variables and methods from a general class
- Encapsulation - Provide access to necessary functionality, hiding the implementation
- Generics - Provide a general class independent of the data it operates on
Generics Construction

• When writing a generic class, we use a type parameter as a placeholder for any reference type

• We let the instantiating class define what reference type the type parameter actually represents
Type Parameter

- Example: public class Sample<T> {...}

- Here, T is the type parameter
  - Convention: Single Uppercase Letter
  - Uppercase Motivation: So are reference types
  - Single Letter: No one really knows
Using Generics

• When we *instantiate* a generic class, we need to define the reference type that the *type parameter* will represent

• Example:

```java
Sample<String> s = new Sample<String>();
```
Helpful Tip

• When writing code where Generics are involved, there are many caveats that may result in compile-time and run-time errors.

• To catch many of these, use the compilation argument -Xlint

• Ex: javac -Xlint Sample.java

• For an IDE, read the documentation as to how one sets compiler options

• Ignore “no def of serialVersionUID”
Generics Example

```java
public class Sample<T> {
    private T data;

    public void setData(T newData) {
        data = newData;
    }

    public T getData() {
        return data;
    }
}
```
public class Pair<T>
{
    private T first;
    private T second;

    public Pair()
    {
        first = null;
        second = null;
    }

    public Pair(T firstItem, T secondItem)
    {
        first = firstItem;
        second = secondItem;
    }

    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            Pair<T> otherPair = (Pair<T>)otherObject;
            return (first.equals(otherPair.first)
                    && second.equals(otherPair.second));
        }
    }
}
Using the Generic Pair

Pet male = new Pet();
Pet female = new Pet();

// Set data for male and female
Pair<Pet> breedingPair = new Pair<Pet>(male, female);
Parameter Pitfalls

• Generic constructors do not have a type parameter
  • Correct: public Pair(){...}
  • Incorrect: public Pair<T>(){...}

• Generic methods have a type parameter, but do not have angle brackets <>
  • Correct: public Pair(T one, T two)
  • Incorrect: public Pair(<T> one, <T> two)
Parameter Pitfalls

• Generic instantiations require angle brackets and a type parameter

  • Correct: `Pair<Pet> p = new Pair<Pet>()`

  • Incorrect: `Pair p = new Pair()`

• Why is this?

  • Hint: `Pair p = new Pair("a","b");`
  `Pair p = new Pair(1,2);`
Parameter Pitfalls

• Primitive types *cannot* be used for a type parameter
  • Why?

• Yet, I claim that we can have generics that operate on primitives
  • How?
Parameter Pitfalls

- Type Parameters are not legal everywhere a type name is legal

  - `T object = new T();`
  - `T[] a = new T[10];`

Illegal:
- We need a constructor name, not a type name!
- This is a constructor in disguise
Parameter Pitfalls

• An instantiation of a Generic cannot be an array base type

  • Incorrect:
    ```java
    Pair<String>[] a = new Pair<String>[10];
    ```

• Don’t worry about why - it is well beyond the scope of this course
Generics with Multiple Parameters

- Generics can have *more than one* type parameter
- Ex: `public class MyGeneric<T1, T2>{...}`
- Remember to use both when instantiating!
- Ex: `MyGeneric<T1, T2> g = new MyGeneric<T1, T2>();`
Generic Restrictions

- Consider a Generic class where we wish to *compare* the elements
- To use `compareTo`, we need to extend `Comparable`
- Ex:
  ```java
  public class Compare<T extends Comparable>{...}
  ```
Test Review
Abstraction

• Essence: Understand only what inputs a method takes, and what output it returns

• Use of a method should **not** depend on its implementation

• This allows project size to grow without decreasing the readability of the code
Roadmap

- Abstraction is an **essential** concept
- Abstraction is a building block for concepts covered later in the course
  - Example: Polymorphism (Many Forms)
    - Isosceles Triangle > Triangle > Shape
Lecture Review

- What must we know about a method’s input?
  - Number of Inputs
  - Data Type of each Input
  - Permissible Values
  - Meaning of Input Data
Lecture Review

• What must we know about a method’s output?
  • Number of Outputs
  • Were the Inputs Changed
  • Data Type of each Input
  • Permissible Values
  • Meaning of Input Data
Concurrency

• When tasks execute in **sequence**, they are executed one after the other.

• When tasks execute in **parallel**, they are executed simultaneously.
Concurrency

• By writing code that can execute multiple tasks in parallel, we gain a significant advantage by having multiple processors.

• Sequential tasks are unable to use the processing power available.
Concurrency

- A **thread** is a separate computation process
- Each parallel task should be given its own thread
public class MyThreadedClass extends Thread{

    //Optional Instance Variables

    //Optional Methods

    public void run(){
        //Begin Task Work Here
    }

}
Confusing Issues

• The class Thread has its own run() method, however we do not want to use that method as it is not specific to our class.

• Instead, we will define our own run() method within our threaded class.

• Since both Thread and our class have a method named run(), how does Java choose the correct one?
Using a Threaded Class

- Instantiate a new instance of your class
- Call the start() method
  - This calls the run() method in your class
  - Why does start() call run()?
  - Why not call run() directly()?
== versus .equals

• When comparing primitives (int, double, boolean) use ==
• When comparing classes, use .equals
• NEVER* use == with Strings
• You are not doing what you think you are
References

- Primitives - the **actual value** is held at the memory address of the variable

- Class Types - a **reference** to the class location in the heap is held at the memory address

- A reference is another name for a **pointer**

See [http://cslibrary.stanford.edu/104/](http://cslibrary.stanford.edu/104/) for more!
MyClass x = new MyClass("John Doe", 50);
MyClass y = x;
y.setName("Jane Doe");
y.setAge(45);
System.out.println("Name is: " + x.getName());
System.out.println("Age is: " + x.getAge());
Name is: Jane Doe
Age is: 45
Example

MyClass x = new MyClass("John Doe", 50);
MyClass y = new MyClass("Jane Doe", 45);
y = x;
System.out.println("Name is: " + x.getName());
System.out.println("Age is: " + x.getAge());

Name is: John Doe
Age is: 50
Overriding

• Override - to redefine a method in a derived class.

• What is a derived class? A base class?

• Characteristics:
  
  • Method definition in derived class has the same number and types of parameters.
Overloading

- Overloading - having two or more method definitions with the same *name*, but with differing *method signatures*.

- Method signature - the name, number and types of parameters of the method.
Interfaces in Java

- Define what methods a class should have, and their purpose.
- An interface is the “skeleton” of a class, but it is not a class!
- Interfaces are actually types

Not a “Java Skeleton”
Is This Legal?

- Assume that MyInterface is an Interface
- MyInterface x = new MyInterface();
- Why or why not?
public interface MyComparableInterface{

    //Compares two objects. Returns: <0 if this
    //precedes o, 0 if this equals o, >0 if this
    //follows o.

    public int compareTo(Object o);
}

How to Use Interfaces

• To use an interface, we define our class to implement the interface type.

• Example:

public class MyClass implements MyInterface{
   //Methods in MyInterface here
   public int myInterfaceMethod(int x){...}
}

Using Interfaces

• Unlike inheritance, where a class can only extend one other class, we can implement multiple interfaces.

• Why can’t we extend multiple classes?

• Why can we implement multiple classes?
Inheritance

• Inheritance is an OOP concept that allows a class to \textit{inherit} the instance variables and methods of another class.

• We say that a \textit{derived class} that \textit{extends} a \textit{base class} inherits the instance variables and methods of the base class.

• The derived class can \textit{override} methods from the base class to suit its purpose.
public class derived extends base {

    //Instance Vars for derived class
    //Omit I.V.‘s from base class

    //Overridden & New Methods
    //Omit Identical Methods from base

}
The `super` keyword

- Sometimes, we may wish to call methods from the base, or `superclass`, from the derived class
- This is done with the command `super`
- Ex: `super.baseClassMethod( ... )`
- Note: `super.super` is illegal
Constructors

• A derived class should always call super() as the first call in the constructor.

• This makes the call explicit and lets the base class set up instance variables.

• You can call either super or this as the first call in the constructor.

• BUT NOT BOTH! (Exclusive-Or)

• What does the this keyword mean?

CS 180 - John Ross Wallrabenstein
Calling other Base Class Methods

• You can call methods in the base class to avoid re-writing code in derived classes

• This is especially useful for equals methods

• Call the super.equals(Object) method, then add more specific constraints in the derived class
## Encapsulation Summary

<table>
<thead>
<tr>
<th></th>
<th>Public</th>
<th>Protected</th>
<th>Package</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class Itself</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Same Package</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Derived Class</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Other Classes</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

*Slide Author: Dr. Keith Frikken, Miami University*
Questions