Java Review II

Idaho State University | Computer Science

Dr. Isaac Griffith

CS 2263
Department of Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will:

- Understand how to define the different member types
  - Fields including Constants and Enum Literals
  - Methods including Constructors and Destructors
  - Initializers
  - Contained Types
- Understand the basic statements types
  - Basic I/O
  - Flow Control
    - Selection: If/Switch
    - Iteration: For, ForEach, While, Do..While
- Understand the basics of Functional Programming in Java

What's in it for Me?

Before we start this lecture, I would like you to:

1. Take out a sheet of paper
2. On this piece of paper, write the following:
   - One thing you feel you know well from the list of outcomes
   - Two things you want to know more about from the list of outcomes.
3. Pause the lecture and complete the list.
4. When you are finished, resume the lecture.

Type Members

CS 2263

ROAR

Type Members

- All types contain zero or more of the following members:
  - Fields
    - Constants
    - Enum Literals
  - Methods
    - Constructors
    - Destructors
  - Initializers
  - Contained Types

Fields

- Provide the ability to store data local to a specific class and shared across all methods of the class, there are two forms:
  - **Instance Fields:** variables whose values are local to a specific object instance of the class
  - **Class Fields:** variables whose values are local to a specific class but not to any specific instance

Examples:

```
public class A {
  // instance fields
  private int width;
  private int height;

  // class fields
  private static int x;
  private static int y;
}
```

Constants

- Constants provide the ability to repeatedly reference values that never change
- Useful, so that we can write them once and reference them many times
  - If the value changes, we need only change it once.
- Though, I would suggest probably reading constants in rather than programming them in
- Constants are defined as variables with the following modifiers:
  - `public` can be accessed by anyone (though you could use any access modifier)
  - `static` belong to the class, not an object instance, in which they are declared
  - `final` cannot be modified
- Typically have all caps names
- Note: all variables defined in Interfaces are constants

Examples

```
public static final double PI = 3.14159265359;
public static final double E = 2.71828182845;
public static final double G = 9.8;
```

Methods

- Methods define the behavior, or functionality, of a class
- There are several types of methods that can be defined
  - Instance Methods defined within a Class, Enum, or Record
    - Accessors and Mutators defined within a Class or Enum
  - Static Methods defined within a Class, Enum, or Record
  - Constructors and Destructors within a Class or Enum
  - Interface Methods defined within an Interface
    - Default Methods defined within an Interface

Instance Methods

- These are the basic building blocks of programming in Java
- Provide all functionality (or behavior) for all programs
- Have four parts:
  - Return type - The type of value to be returned (Java only allows one)
  - Method Name - The name of the method (not necessarily unique)
  - Parameters - The inputs to the method
  - Body - the section containing the code (if the method is not abstract)

Example

```
public abstract int multiply(int a, int b);

public void printArray(int[] array) {
  // do something
}
```

Method Overloading/Overriding

Overloading: having multiple methods with the same name, but with

- A different set of parameters, and/or
- A different return type
- **Note:** It is better to have one method overloaded than two methods that do the same thing

Example:

```
public class Test {
  int myMethod(int x) {}
  float myMethod(float x) {}
  double myMethod(double x, double y) {}
}
```

Overriding: defining a method in a subclass with the same signature as a method in an ancestor class

- Typically done to provide additional functionality
- If you override a method, you should call the `super` form to utilize the functionality from the ancestor class
  - otherwise, you are violating good OOP

Example:

```
public class A {

  int myMethod(int x) {}
}

public class B extends A {

  int myMethod(int x) {
    super.myMethod(x);
  }
}
```

Final Methods

- Final methods are those declared with the `final` modifier
- These methods cannot be overridden by subclasses
- Represent the final form of the method
  - Thus you cannot have a method that is both `abstract` and `final`

Example

```
public final int multiply(int a, int b) {

}
```

Default Methods

- A special form of a method used only in interfaces
- This provides a default implementation of an interface method
  - Though it may be overridden by implementing classes
  - Provides binary compatibility with systems written for older versions of the interface

Example:

```
import java.time.*;

public interface TimeClient {
  void setTime(int hour, int minute, int second);
  void setDate(int day, int month, int year);
  void setDateAndTime(int day, int month, int year,
                      int hour, int minute, int second);
  ZoneId getZoneId (String zoneString);
  String getLocalDateTime();

  default ZonedDateTime getZonedDateTime(String zoneString) {
    return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
  }
}
```

Constructors

- A special method used to setup a new instance of a class
- Java provides a default no-parameter, empty constructor.
  - If you implement a constructor then the default is no longer provided
  - If you implement one or more constructors, all subclasses must call at least one from their own constructor
    - Called using `super(...)` which must be the first line of the child constructor
  - You can also call other constructors from the same class
    - Called using `this(...)` and must be the first line of the constructor
- Must be named the same as the class containing it
- Has no return type

Example:

```
public abstract class Vehicle {
  protected int year;

  public Vehicle(int year) {
    this.year = year;
  }
}

public class Car extends Vehicle {

  private String make, model;

  public Car() {
    this("Ford", "F150", "2021")
  }

  public Car(String make, String model, int year) {
    super(year)
    this.make = make;
    this.model = model;
  }
}
```

Destructors

- Destructors are similar to constructors, but rather than setup an instance, they tear them down
- In Java, this functionality is provided in the superclass `java.lang.Object` in the method `finalize()`
- `finalize()` is called by the Garbage Collector when an object is removed from the heap
- It is useful if you have any resources that need to be removed, when an instance is removed

Example:

```
public class TestA {

  @Override
  public void finalize() {
    // do something
  }
}
```

Contained Types

Example:

```
public class TestA {

  enum ItemType {}

  class ContainedItem {}
}
```

Initializers

- Provides the ability to execute code upon the creation of an object or upon the class loading (static form).
- **Note:** if there are multiple initializers, they are executed in the order in which they occur in the class

Example:

```
public class Test {
  int x, y;
  long id;
  static long ID;

  {
    x = 0;
    y = 1;
    id = ID++;
  }

  static {
    ID = 0;
  }
}
```

Statements

CS 2263

ROAR

Basic Input and Output

Basic Input:

1. `BufferedReader` class
   - This may throw an exception

```
BufferedReader reader = new BufferedReader(
    new InputStreamReader(System.in));
String name = reader.readLine();
```

2. Using the `Scanner` class

```
Scanner in = new Scanner(System.in);
String s = in.nextLine();
int a = in.nextInt();
```

3. Using the `Console` class
   - `System.console()` will return null if there is no console.

```
String name = System.console().readLine();
```

4. Command Line Args

```
public static void main(String[] args) {
  if (args.length > 0) {
    for (String val : args)
      System.out.println(val);
  } else {
    System.out.println("No args!");
  }
}
```

Basic Output:

1. The simplest form of output is to write to the standard out:

```
System.out.println("Hello World!")
System.out.printf("%s\n", "Hello World!");
System.out.print("Hello World!\n");
```

2. A better approach (if on a terminal) is to use the System Console:
   - However is a console is not available `System.console()` will return null.

```
System.console().writer().println("Hello World!");
System.console().writer().printf("%s\n", "Hello World!");
System.console().writer().print("Hello World!\n");
```

Flow Control

- There are two main ways in which we can adjust the logical flow of a program
  - *Selection*: which allows us to conditionally execute blocks of code
    - Achieved using either `if` or `switch` statements
  - *Iteration*: which allows us to conditionally repeat the execution of blocks of code
    - Achieved using one of `for`, `for each`, `while`, or `do..while` loops

Selection

Original Switch Example:

```
String s = "";
//...
switch(s) {
  case "val1":
    System.out.println("ans2");
    break;
  case "val2":
    System.out.println("ans1");
    break;
  default:
    System.out.println("default");
}
```

New Switch Example:

```
String s = "";
//...
switch(s) {
  case "val1" -> System.out.println("ans1");
  case "val2" -> System.out.println("ans2");
  default -> System.out.println("default");
}
```

If Statements

```
if (x < 3) {
  // do something
} else if (x > 4) {
  // do something
} else {
  // do something
}
```

Loops

For Loop

```
for (int i = 0; i < 1000; i++) {
  System.out.println(i);
}
```

For Each Loop

```
int[] array
List<Integer> list

for(int i : array) System.out.println(i);
for(int i : list) System.out.println(i);

list.forEach(i -> System.out.println(i););
list.forEach(System.out::println);
```

While Loops

```
int i = 0;
while(i < 1000) {
  System.out.println(i);
  i++;
}
```

Do..While Loops

```
int i = 0;
do {
  System.out.println(i);
  i++;
} while(i < 0)
```

Arrays

- Provide the ability to store multiple values of a single type in a single variable
- Arrays are useful when you know (or can calculate) the exact number of items you need
- Arrays are useful when you need to access each item in O(1) time
- Arrays are a type of object, but can hold primitive or object types

Examples

```java
// Initialization
int[] x = { 1, 2, 3, 4 };

int[] x = new int[4];
Arrays.fill(x, 0);

// Accessing items
int y = x[0]; // 0 indexed

// Iteration
for (int i = 0; i < x.length; i++)
  System.out.println(x[i]);

for (int item : x)
  System.out.println(item);
```

Text Blocks (aka Multiline Strings)

- Often times we have a need to have a string span across multiple lines
- We can achieve this with text blocks

Example:

```
String html = """
              <html>
                  <body>
                      <p>Hello, world</p>
                  </body>
              </html>
              """;
```

Functional Programming

CS 2263

ROAR

Lambda Expressions

- Provide a short block of code that takes parameters and returns a value
- Similar to methods but do not need a name and can be implemented inside a method

Syntax:

- Simplest form takes a single parameter and an expression

  ```
  parameter -> expression
  ```

- For more than one parameter, wrap them in parentheses

  ```
  (parameter1, parameter2) -> expression
  ```

- For more advanced code you can wrap the code in curly braces

  ```
  (parameter1, parameter2) -> { code block }
  ```

Example

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers =
          new ArrayList<>(5,9,8,1);
    numbers.forEach( (n) -> {
      System.out.println(n);
    });
  }
}
```

Referencing Lambdas

- You can store a lambda expression in a variable for repeated use later

Example

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<>(5,9,8,1);
    Consumer<Integer> method = (n) -> { System.out.println(n); };
    numbers.forEach( method );
  }
}
```

Method References

- Provides a shorthand for executing a method
- Can only be used to replace a single method of a lambda expression
- Does not allow for passing arguments to the method
- Works for both Static and non-static methods

Example

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<>(5,9,8,1);
    numbers.forEach( System.out::println );
  }
}
```

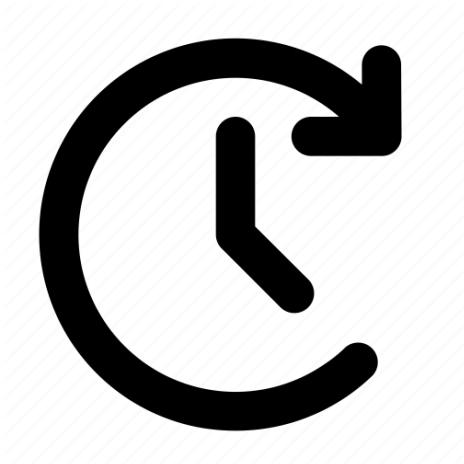
Closing

In closing, let's return to your paper from the beginning of the lecture:

- Ask yourself the following questions:
  1. For the item you felt you knew well:
     - Did you learning anything new?
     - Did you know it as well as you thought?
  2. For the items you wanted to know more about:
     - Did you learn anything new about those concepts?
     - Did you find that you knew more than you thought?

For Next Time

- Review Chapter 2.3 - 2.6
- Review the Lecture
- Come to Class
- Read the Build Automation Article
- Continue working on Homework 01



Are there any questions?