Classes and Objects

CS 240 – Advanced Programming Concepts

Topics

- Classes
- Objects
- References
- Static Members
- Accessors and Mutators (getters and setters)
- Constructors
- Overriding
- Overloading
- Methods
  - toString
  - equals
  - hashCode
- Final Members
- The ‘this’ reference
- Enums
- OO Design Basics
- Class Design

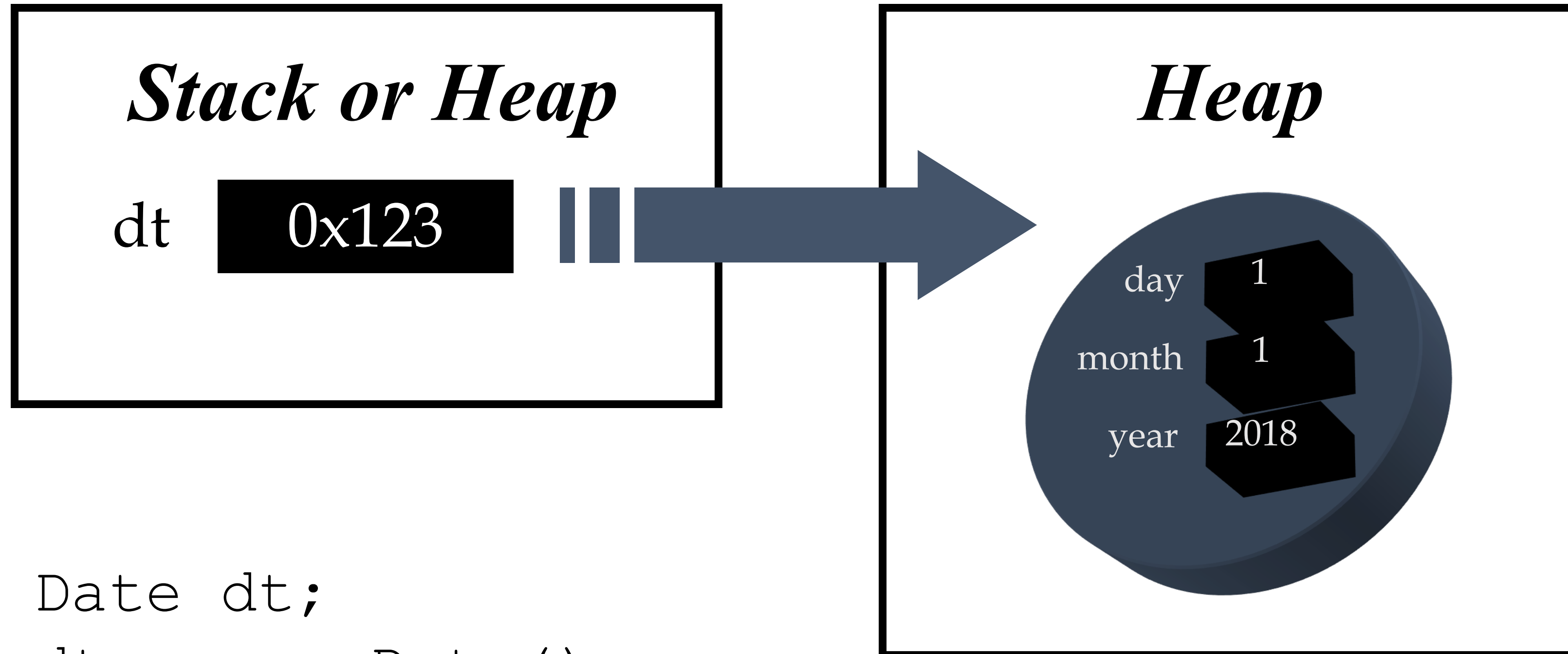
Class Example

```
package SimpleClassExample;

import java.util.Date;

public class Person {

    private String firstName;
    private String lastName;
    private Date birthDate;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Date getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
}
```

Classes and Objects

- What they are
- Difference between classes and objects
- All code (except package and import statements) in Java is written in a class (unlike C++)
- Create an object (instance of a class) with the new keyword, followed by a constructor invocation
    - Strings and arrays are objects but they have special construction syntax
- Questions?

Object References




```
Date dt;
dt = new Date();
```

Object References

- Refer to (allow access to) objects
- Do not allow pointer arithmetic (unlike C++)
- Creation of a reference does not create an object
- References and objects are closely related, but are not the same thing
- Multiple references can refer to the same object

Reference Equality vs. Object Equality

Reference Equality

```
Person p1 = new Person();
p1.setFirstName("Jerod");
p1.setLastName("Wilkerson");

Person p2 = p1;

if(p1 == p2){
   // True or False?
}

p2 = new Person();
p2.setFirstName("Jerod");
p2.setLastName("Wilkerson");

if(p1 == p2){
   // True or False?
}
```

Object Equality

```
Person p1 = new Person();
p1.setFirstName("Jerod");
p1.setLastName("Wilkerson");

Person p2 = p1;

if(p1.equals(p2)){
   // True or False?
}

p2 = new Person();
p2.setFirstName("Jerod");
p2.setLastName("Wilkerson");

if(p1.equals(p2)){
   // True or False?
}
```

Instance vs. Static Variables

Instance Variables

- Each object (instance) gets it’s own copy of all of the instance variables defined in it’s class
- Most variables should be instance variables
- **Example:** Allows two date objects to represent different dates

Static Variables

- Static variables are associated with the class not with instances
- Use in special cases where you won’t create instances of a class, or all instances should share the same values
- **Example:** If the variables of a Date class where static, all dates in a program would represent the same date

Instance vs. Static Methods

Instance Methods

- Methods are associated with a specific instance (object)
- Invoked from a reference that refers to an instance
- When invoked (on an object), the variables they access are that object's instance variables

Static Methods

- Methods are associated with a class (not an instance)
- Invoked by using the class name (not a reference)
- Can be invoked from a reference, but the method is still not associated with an instance
- Cannot access instance variables

Static Method Example (won’t work)

```
public class StaticExampleWrong1 {

    private int myInstanceVariable;

    public static void main(String [] args) {
        myInstanceVariable = 10;
        myInstanceMethod();
    }

    public void myInstanceMethod() {

    }
}
```

Static Example (will work, but still wrong)

```
public class StaticExampleWrong2 {

    private static int myInstanceVariable;

    public static void main(String [] args) {
        myInstanceVariable = 10;
        myInstanceMethod();
    }

    public static void myInstanceMethod() {

    }
}
```

Static Example (right way)

```
public class StaticExample3 {

    private int myInstanceVariable;

    public static void main(String [] args) {
        StaticExample3 instance = new StaticExample3();
        instance.myInstanceVariable = 10;
        instance.myInstanceMethod();
    }

    public void myInstanceMethod() {

    }
}
```

Getters / Setters (Accessors / Mutators)

- Methods for getting and setting instance variables
- Allow you to control access to instance variables
  - Make variables private and only allow access through getters and setters
- Not required to provide getters and setters for all variables
- Can use your IDE to generate them from variable declarations

- Code Example
  - Person.java

Constructors

- Code executed at object creation time
- Must match the class name
- Like a method without a return type
- All classes have at least one
  - Default constructors: Written by the compiler if you don't write any
- Classes can have multiple constructors (with different parameter types)
- Constructors invoke each other with 'this(…)'
- Constructors invoke parent constructor with 'super(…)'
- this(…) or super(…) is **always** the first statement
- Code Examples
  - Person.java
  - Employee.java

Inheritance

- Inherit members of a parent (super) class without explicitly writing them in the child (sub) class
- Use the ‘extends’ keyword
  - `public class Employee` **`extends`** `Person {…}`
- Use the “is-a” rule
- Every class extends Object (either directly or indirectly)

- What is inherited?
  - All instance variables (even private ones)
  - All non-private, non-static methods
- What is not inherited?
  - Constructors
  - Static methods
  - Private methods

Method Overriding

- A subclass replaces an inherited method by redefining it
  - Argument list must be the same
  - Return type must be the same (or a subclass)
  - Must not make access modifier more restrictive
  - Must not throw new or broader checked exceptions
- Can call the overridden version of the method by using super
  - Examples: Person.java, Employee.java (see toString() methods)
- Use @Override annotation to prevent typos
  - Examples:
    - In previous example, replace toString with tostring (lowercase ‘S’) and see what happens
    - Remove @Override and replace toString with tostring

Common Methods to Override

- `public String` **`toString`**`()`
- `public boolean` **`equals`**`(Object obj)`
- `public int` **`hashCode`**`()`

The hashCode() Method

The general contract of hashCode :

- Whenever it is invoked on the same object more than once during an execution of a Java application, **the hashCode method must consistently return the same integer**, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects **must produce the same integer result**.
- It is **not required** that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, **the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables**.

Implementing a HashCode() Method

- Hash each value in the object that you want included in the hash
  - For integral numbers, use the number as the hash for that value
  - For Strings (or other objects), call the object's hashCode() method
  - For Objects that may be null, can use the Objects.hashCode() method
  - For arrays, either hash each element in the array, or call Arrays.hashCode()
- While computing a hash, accumulate the hashed values, multiplying the accumulated value by an odd prime number (not 2) before adding the next hashed value
  - We usually multiply by 31
- Resources for learning more:
  - API documentation for hashCode() method in Object class
  - https://www.baeldung.com/java-hashcode

Method Overloading

- Reuse a method name with a different argument list
- Example: The PrintWriter class:

```
public class PrintWriter
extends Writer {
  public void print(boolean)
  public void print(char)
  public void print(char[])
  public void print(double)
  public void print(float)
  public void print(int)
  public void print(long)
  public void print(Object)
  public void print(String)

  public void println(boolean)
  public void println(char)
  …
}
```

Final

- Final Variables
  - Can’t be changed after a value is assigned
  - For instance variables, the last chance to assign is in a constructor
  - `public final int myVariable = 10;`
- Final Reference Variables
  - `public final ArrayList list = new ArrayList();`
  - What can’t change?   What can change?
- Final Methods
  - `public final void myMethod() {…}`
  - What can you not do to a final method?
  - **Hint:** In Java, all non-final instance methods are virtual

The ‘this’ Reference

- What is it?
- When do I have to use it and when can it be inferred by the compiler?

Enums

- Like a class
- Use where you would otherwise have an unrestricted String with only a few values being valid

```
public enum Gender {
    Male, Female;

    @Override
    public String toString() {
        return this == Male ? "m" : "f";
    }
}
```

Enum Example

Without Enum

```
public class Person {
    private String firstName;
    private String lastName;
    private String gender;

    …

    // Problem: Gender can be set to any
    // String, even if we only consider m and f
    //to be valid
    public void setGender(String gender) {
        this.gender = gender;
    }
}
```

With Enum

```
public class Person {
    private String firstName;
    private String lastName;
    private Gender gender;

    …

    //  Problem solved. Now gender can only
    // be set to Gender.Male or
    //Gender.Female
    public void setGender(Gender gender) {
        this.gender = gender;
    }
}
```

Object-Oriented Design

- Decompose a program into classes
- Use separate classes to represent each concept in the application domain
- Identify relationships between classes
  - Represent Is-A relationships with inheritance
  - Represent Has-A and Uses-A relationships with references

Class Design Guidelines

- Keep data private
- Not all fields need individual getters and setters (think about who needs access)
- Use a standard structure for class definitions
- Break up classes that have too many responsibilities
- Make the names of your classes and methods reflect their responsibilities
- Classes have noun names, methods usually have verb names
- Use static methods as an exception, not a general rule

Standard Class Structure

```
public class MyClass {
  // Static Variables

  // Instance Variables

  // Main Method (if it exists)

  // Constructors

  // Methods
  //   (grouped by functionality, not by accessibility, not by static vs
  //    instance, etc)
}
```