Hoare Triples

Hoare triples are a way to reason about the correctness of code. They consist of a precondition, a code block, and a postcondition. The precondition is an assumption that holds before the code runs, the postcondition is a property that holds after the code finishes, and the code block is the code itself. The notation is:

\[ \{ P \} \text{Code} \{ Q \} \]

Where am I? Big ideas so far.

- **Java variables have types (L1)**
  - A type is a set of values and operations on them (int: +, -, *, /, %, etc.)

- **Classes define new types (L2)**
  - **Methods** are the operations on objects of that class.
  - **Fields** allow objects to contain data (L3)

Class House

```java
public class House {
    private int bdrs; // number of bedrooms, >= 0.
    private int baths; // number of bathrooms, in 1..5

    /** Constructor: number of bedrooms b1, number of bathrooms b2
    *  Prec: b1 >= 0, 0 <= b2 <= 5 */
    public House(int b1, int b2);

    /** Return number of bedrooms */
    public int getBeds() {
        return bdrs;
    }

    /** Return number of bathrooms */
    public int getBaths() {
        return baths;
    }

    contains other methods!
}
```

Class Object

```java
public class Object {
    // Other methods!
}
```
Class Object: the superest class of all

```java
public class House extends Object {
  private int bdrs; // number of bedrooms,
  private int baths; // number of bathrooms,
  /** Constructor: number of bedrooms b1, number of bathrooms b2 */
  House(int b1, int b2) {
    bdrs = b1 >= 0, 0 < b2 <= 5
  }
  /** Return number of bedrooms */
  public int getBeds() {
    return bdrs;
  }
  /** Return number of bathrooms */
  public int getBaths() {
    return baths;
  }
  House(...) getBeds() getBaths()
  private Apartment downstairs;
  setBeds(...) setBaths(...) 
}
```

Accessing superclass things

- Subclasses are different classes
  - Public fields and methods can be accessed
  - Private fields and methods cannot be accessed
  - Protected fields can be access by subclasses

Overrides methods

```java
public class House extends Object {
  private int bdrs; // number of bedrooms,
  private int baths; // number of bathrooms,
  /** Constructor: number of bedrooms b1, number of bathrooms b2 */
  House(int b1, int b2) {
    bdrs = b1 >= 0, 0 < b2 <= 5
  }
  /** Return number of bedrooms */
  public int getBeds() {
    return bhrs;
  }
  /** Return number of bathrooms */
  public int getBaths() {
    return baths;
  }
  //constructor
  public House(int floor, Apartment downstairs) {
    floor = floor;
    downstairs = downstairs;
  }
  Inside-out rule shows that field x is inaccessible!
 :x
}
```

Classes can extend other classes

```java
/** An instance is a subclass of JFrame */
public class C extends java.awt.JFrame {
  hide() show()
  JFrame JFrame:
  C.subclass of JFrame
  JFrame: superclass of C
  C inherits all methods that are in a JFrame
  Object has 2 partitions:
  one for JFrame methods, one for C methods
}
```

Overriding methods

```java
Object defines a method toString() that returns the name of the object
Apartment@af8
toString() equals(Object) hashCode()
```

Java Convention: Define toString() in any class to return a representation of an object, giving info about the values in its fields.
New definitions of toString() override the definition in Object.toString()
When should you make a subclass?

- The inheritance hierarchy should reflect **modeling semantics**, not implementation shortcuts.
- A should extend B if and only if A “is a” B
  - An elephant is an animal, so Elephant extends Animal
  - A car is a vehicle, so Car extends Vehicle
  - An instance of any class is an object, so AnyClass extends java.lang.Object
- Don’t use extends just to get access to protected fields!

When should you make a subclass?

- Which of the following seem like reasonable designs?
  - Triangle extends Shape { ... }
  - PHDTester extends PHD { ... }
  - BankAccount extends CheckingAccount { ... }

When should you make a subclass?

- Which of the following seem like reasonable designs?
  - Triangle extends Shape { ... }
  - Yes! A triangle is a kind of shape.
  - PHDTester extends PHD { ... }
  - No! A PHDTester “tests” a PHD, but itself is not a PHD.
  - BankAccount extends CheckingAccount { ... }
  - No! A checking account is a kind of bank account, we likely would prefer:
    CheckingAccount extends BankAccount { ... }

Static Methods

- Most methods are **instance methods**; every instance of the class has a copy of the method.
- There is only one copy of a **static method**. There is not a copy in each object.

An Example

```java
/** = “this object is below”.
 * Pre: a is not null. */
public boolean isBelow(Apartment a){
    return this == a.downstairs;
}

/** = “a is below b”.
 * Pre: b and c are not null. */
public static boolean isBelow(Apartment b, Apartment a){
    return b == a.downstairs;
}
```

Referencing a static method

```java
public static void main(String[] args){
    Apartment.isBelow(a, b);
}
```
Good example of static methods

- java.lang.Math
  http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

- Or find it by googling Java 8 Math

Static Fields

- There is only one copy of a static method. There is not a copy in each object.
- There is only one copy of a static field. There is not a copy in each object.

What are static fields good for?

Use of static variables: Maintain info about created objects

- public class Apartment extends House {
  public static int numAps; // number of Apartments created
  /** Constructor */
  public Apartment(…) {
    ...
    numAps = numAps + 1;
  }
}

To have numAps contain the number of objects of class Apartment that have been created, simply increment it in constructors.

Class java.awt.Color uses static variables

- An instance of class Color describes a color in the RGB (Red-Green-Blue) color space. The class contains about 20 static variables, each of which is (i.e. contains a pointer to) a non-changeable Color object for a given color:
  - public static final Color black = …;
  - public static final Color blue = …;
  - public static final Color cyan = new Color(0, 255, 255);
  - public static final Color darkGray = …;
  - public static final Color gray = …;
  - public static final Color green = …;
  ...

Uses of static variables: Implement the singleton pattern

- public class WhiteHouse extends House {
  private static final WhiteHouse instance = new WhiteHouse();
  private WhiteHouse() {} // ... constructor
  public static WhiteHouse getInstance() {
    return instance;
  }
  // ... methods
}

Only one WhiteHouse can ever exist.