C8: Understanding Inheritance

Intuitive description

- Intuitive: FLORISTS are SHOPKEEPERS, inheriting various shopkeeper behaviors
- Tension in OOP languages:
  - Extension: child class inherits everything, and may add new properties
  - Contraction: child class is more specialised
- Inheritance is transitive: Dog extends Mammal extends Animal, so Dog inherit from both Mammal and Animal
- Further complication: overriding

Universal base class Object

- Class Object is the universal root of all classes, i.e.
  class X { ..} is equivalent to
  class X extends Object { .. }
- Object provides four useful default methods:
  - equals(Object o) determines whether this is equal to o
  - getClass() returns the class of this
  - hashCode() computes a hash value for this
  - toString() converts this into a String
- Can be overridden, if you change equals(o), you most likely want to change hashCode(), and v.v.

Subclass, subtype, substitutability

- Substitutability: type of a variable does not have to match the type of the actual value, subclasses are OK, too.
- A subclass is usually substitutable, because:
  - Subclass instances have all parent fields
  - Subclasses implement all parent methods
  - Thus, subclasses support parent protocol
- But: not always true (see later)
- Interfaces can also be used for subtyping
- Subtype <> subclass, "stronger notion"

Inheritance for Specialization

- Child class satisfies ALL properties we expect from the parent class, and, in addition, overrides one or more methods
- E.g.: PinBallGame extends Frame, setSize(), show(), etc. inherited from Frame, but paint() overridden
- Good design should strive for this kind of inheritance

Inheritance for Specification

- Special case of I.f.Specialization: not refinements, but actually realizations of incomplete, abstract specs
- Two ways of implementation:
  - Interface
  - Abstract class
    - At least one abstract method => class abstract
    - Abstract class => no direct instances/objects

Abstract class Number

```
public abstract class Number {
    public abstract int intValue();
    public abstract long longValue();
    public abstract float floatValue();
    public abstract double doubleValue();
    public byte byteValue() {return (byte) intValue(); }
    public short shortValue() {return (short) intValue();}
}
```

- Non-abstract subclasses must provide implementations for all abstract methods
- Not all methods need to be abstract

Inheritance for Construction

- Sometimes a class can inherit almost all desired functionality without any conceptual/subtype relationship [e.g. class Hole extends Ball …]
- Pragmatic
- Often directly breaks substitutability

Another example: stack

```
class Stack extends Vector {
    public Object push(Object item)
        { addElement(item); return item; }
    public boolean empty()
        { return isEmpty(); }
    public synchronized Object pop() {
        Object obj = peek();
        removeElementAt(size()-1);
        return obj;
        }
    public synchronized Object peek()
        { return elementAt(size()-1); }
}
```

Inheritance for Extension

- Only adds new methods/fields, no overriding whatsoever
- E.g. Properties are a subclass of HashTable, allowing to read/write key/value pairs from/to files
- Such subclasses are <u>always</u> subtypes

Inheritance for Limitation

- Make subclass more limited (usually when parent can/should not be changed)
- Take a parent method and make it illegal
- Such subclasses are <u>NOT</u> subtypes
- Thus, <u>should be avoided</u>

Example: Vector as set

```
class Set extends Vector {
    public int indexOf(Object obj) {
        System.out.println("Don't use Set.indexOf");
        return 0;
        }
    public Object elementAt(int Index)
        { return null; }
}
```

- Ideally, would use exceptions, but illegal (would change type-signature)
- Actually, not possible as indexOf and elementAt are final in class Vector

Inheritance for Combination

- A new abstraction combining two or more abstractions: multiple inheritance (teaching assistant is both a teacher and a student), forbidden in Java
- But: can implement multiple interfaces
- E.g. RandomAccessFile implements both DataInput and DataOutput protocols
- Usually subtype of all parts

Inheritance forms summary

- Forms:
  - Specialization
  - Specification
  - Construction
  - Extension
  - Limitation
  - Combination
- JAVA: "all subclasses are subtypes" assumed
- If not: possible bug source

Modifiers and inheritance

  - public: can be accessed everywhere
  - protected: inside same package and all subclasses
  - private: only within the class definition
  - static: shared by all instances of a class
  - abstract: no direct instances
  - final: method can't be overridden, class can't be subclassed
- Don't use final for efficiency reasons, JITs should deal with that, use final if you really what to prevent subclassing/overriding

Benefits of inheritance

- Reusability
- Reliability
- Code Sharing
- Interface consistency
- Software components
- Rapid prototyping
- Polymorphism and Frameworks
- Information hiding

Costs of inheritance

- Total speed
- (Binary) Size
- Message-passing Overhead
- Program complexity:
  - Yo-yo problem
- But: [Wulf 72/79] "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity."