eBuilder TechTalks #1

Practical OOP

Speaker: Wimal Perera
Date: 27/3 (Tuesday) from 9.30AM - 11.00AM
Venue: 5th floor, main board room

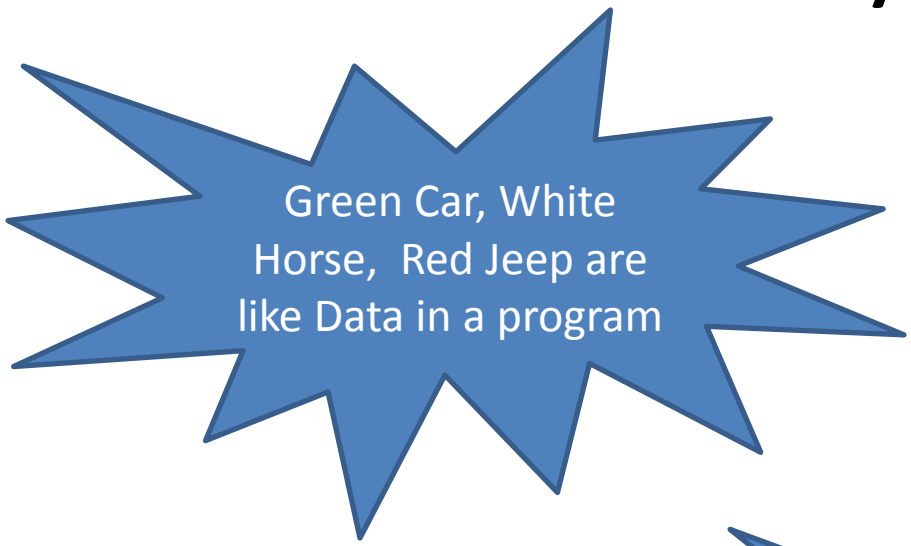
Why OO?

It’s all a mess …
Can you lend me your green car out of this messy toys yard?

Why OO?

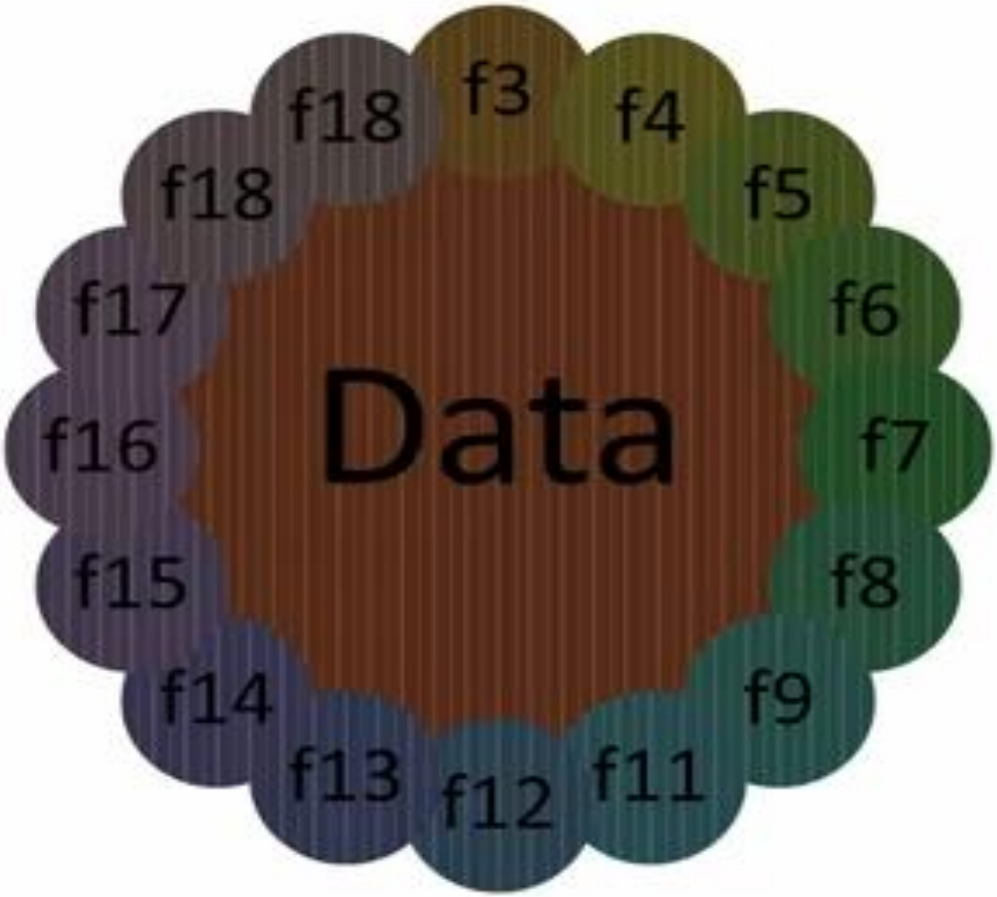
Is it easier now??

Why OO?


Green Car, White
Horse, Red Jeep are
like Data in a program


Find Vehicle, Pick Animal
are like Functions
performed with Data

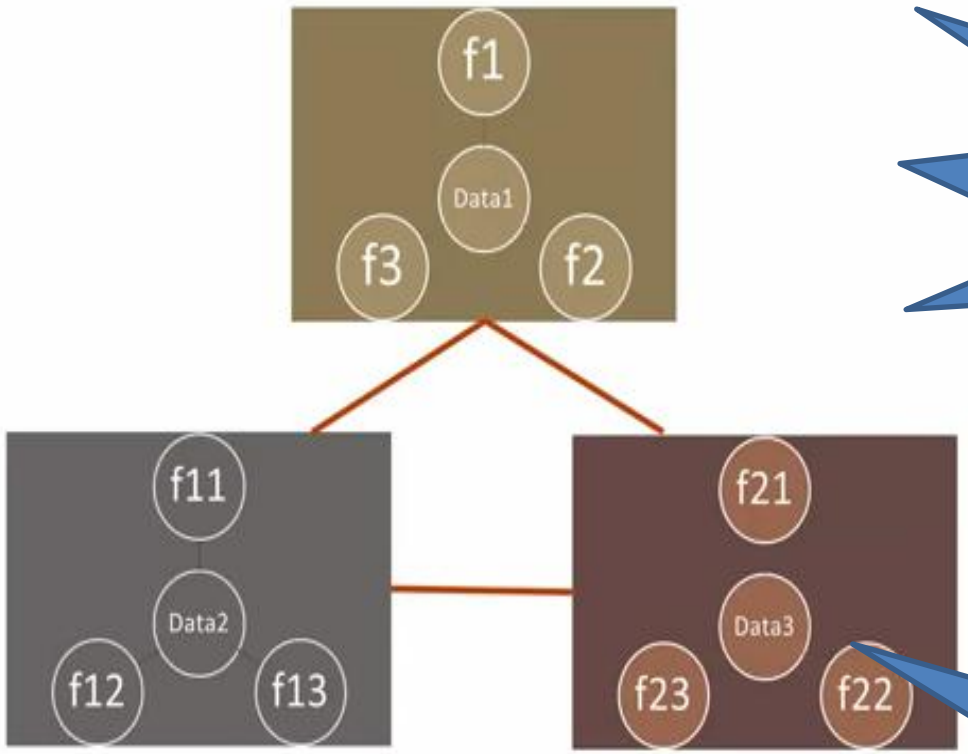
Why OO?



Before OO, a program was like your messy toys bundle

You would have misplaced your white horse during finding a green car …

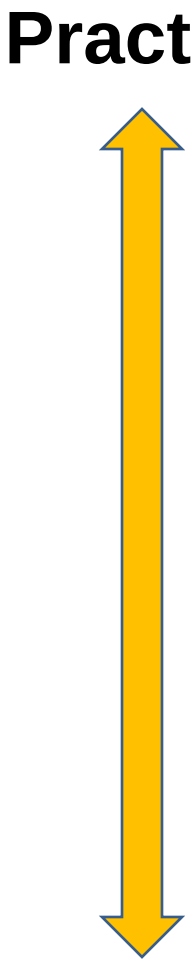
Why OO?



With OO, you can organize vehicles in 1 box, animals in another box etc.

You can perform findVehicle() only within the box related with Vehicles

OO Overview



OO Design Patterns

Recurring solutions to common software design problems found in real-world application development

OO Design Principles

Guidelines to help avoiding a bad OO design (SOLID)

OO Concepts

Foundation of OO; Abstraction, Encapsulation, Inheritance and Polymorphism

OO Concepts

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

OO Design Principles

- **SOLID**
  1. **S**ingle Responsibility Principle
  2. **O**pen-close Principle
  3. **L**iskov's Substitution Principle
  4. **I**nterface Segregation Principle
  5. **D**ependency Inversion Principle

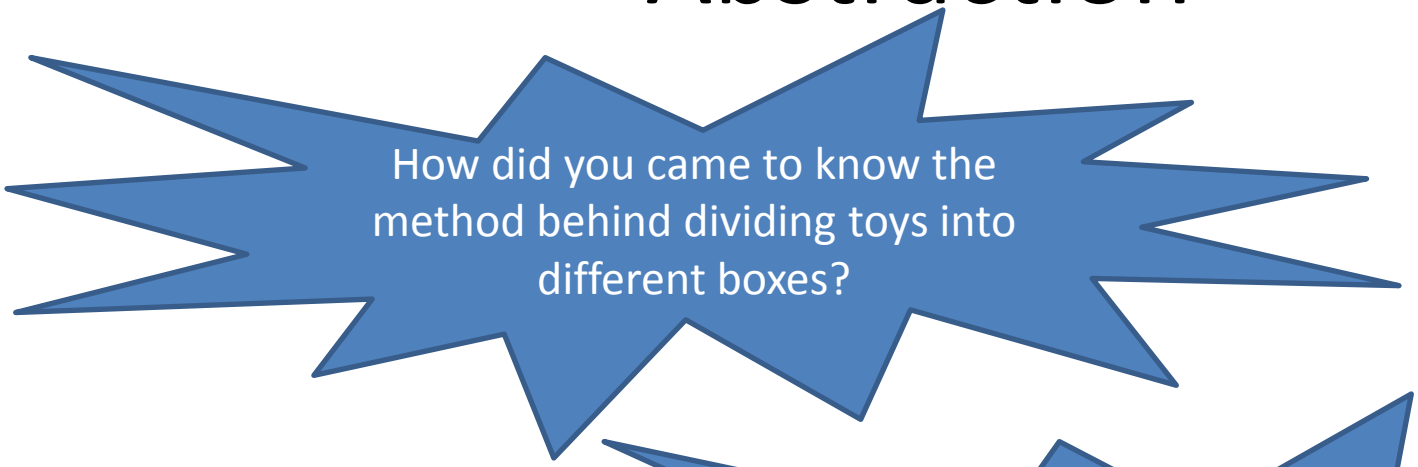
OO Design Patterns (**GoF**)

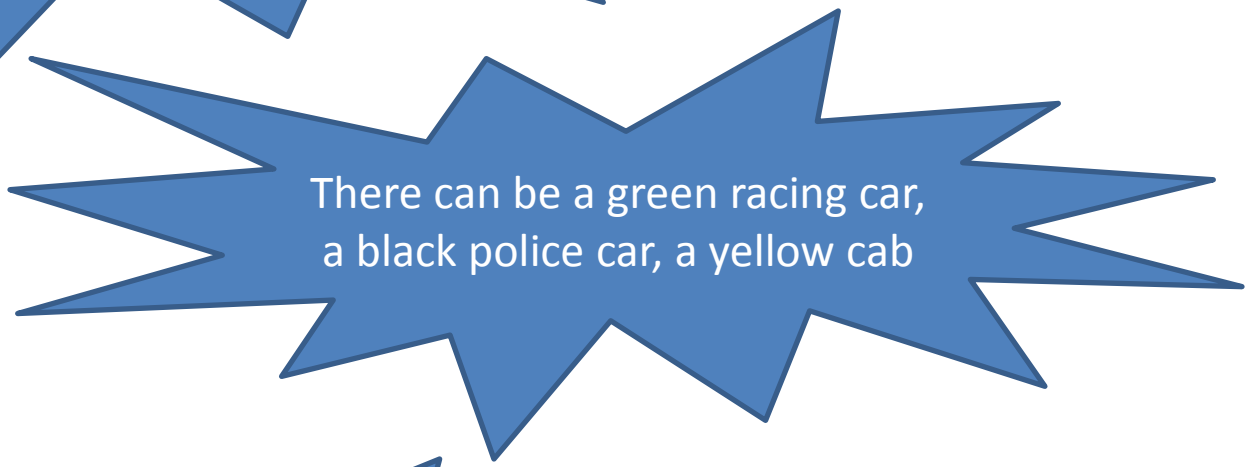
- Creational
  - **Factory, Abstract Factory, Builder, Prototype, Singleton**
- Behavioural
  - **Template Method, Strategy, Memento, Visitor, Command, Interpreter, Iterator, Mediator, State, Chain-of-Responsibility, Strategy**
- Structural
  - **Decorator, Bridge, Adaptor, Proxy, Façade, Flyweight, Composite**

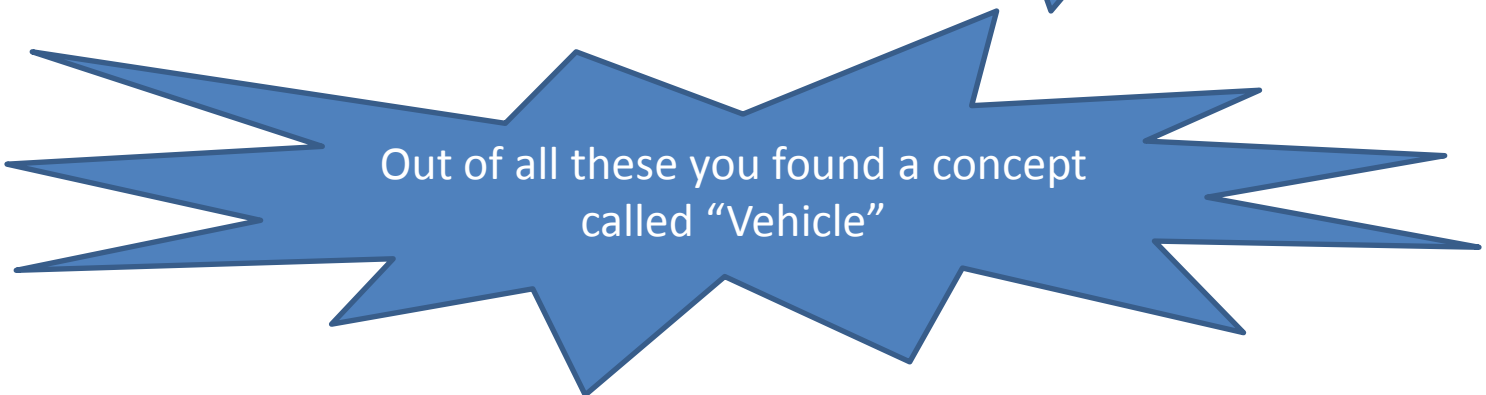
OO Session 1

- We focus on "**OO Concepts**"
  1. Abstraction
  2. Encapsulation
  3. Inheritance
  4. Polymorphism

Abstraction



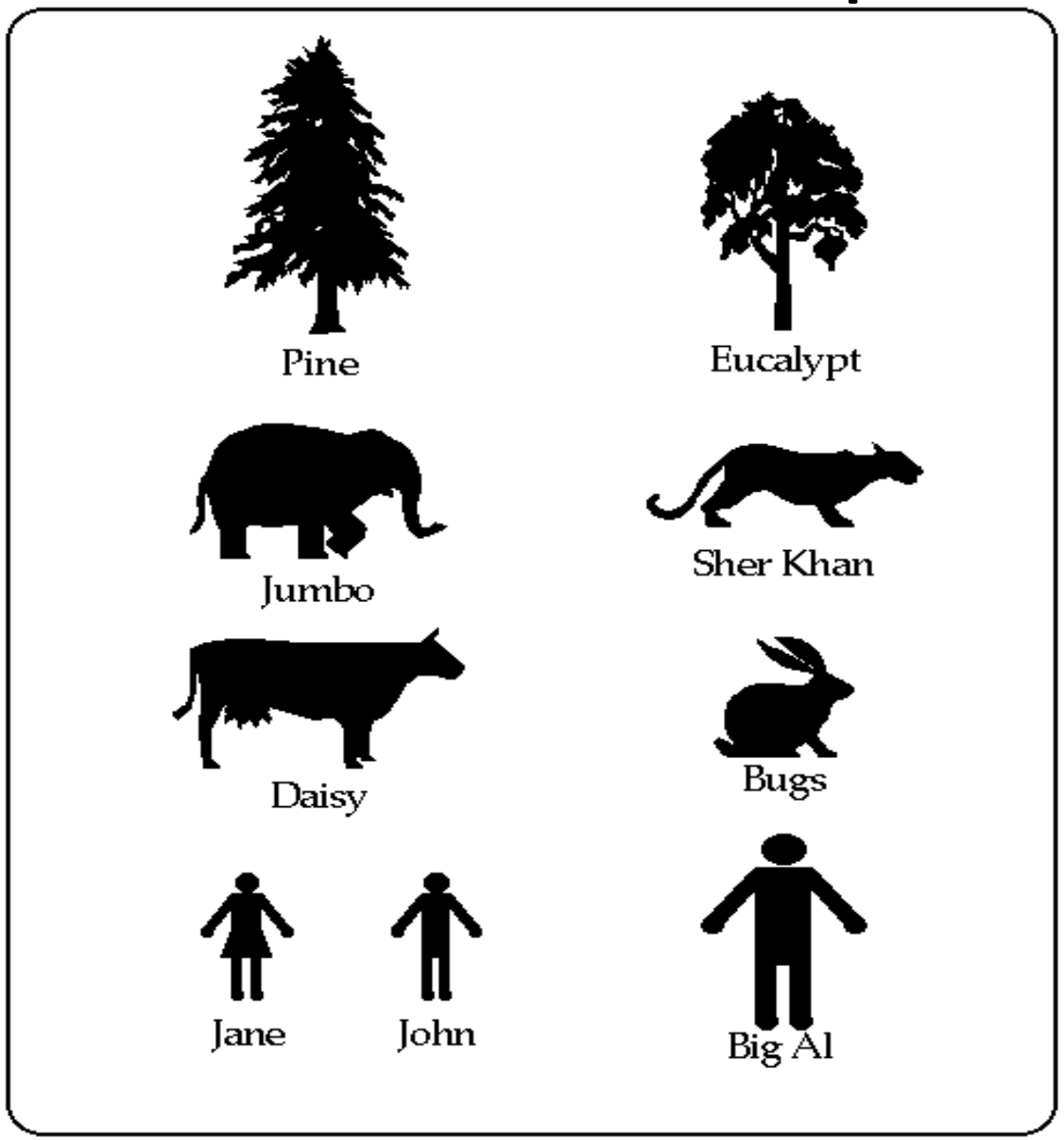




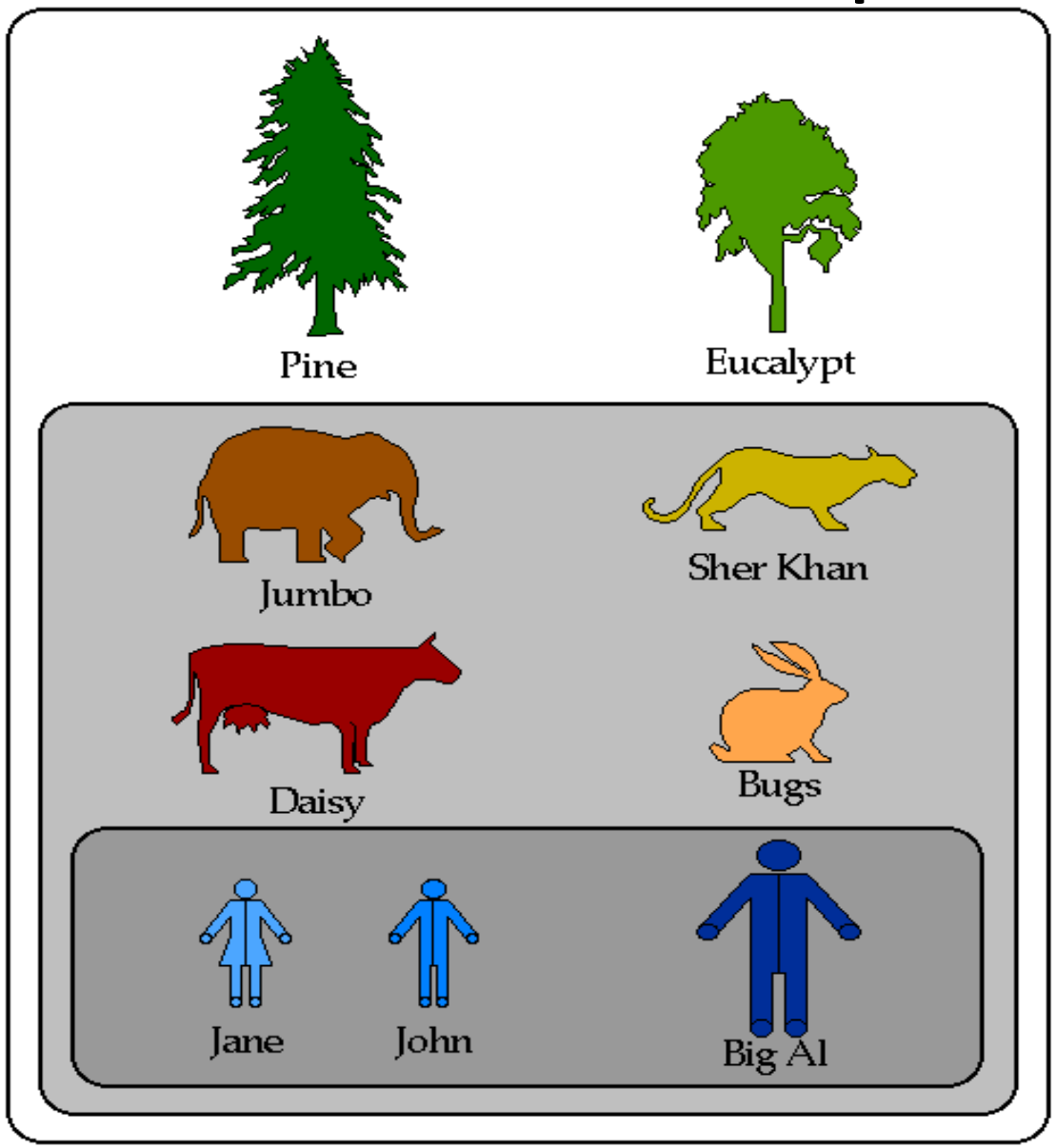
Abstraction

Concrete Thinking	Abstract Thinking
About Tommy, who is my dog	Dogs in general
My dog sitting on a chair	Animal, Spatial Relation, Object
A big ball	Size of an Object
Count up to three cookies	Number of Objects
John likes Betty	Person with Emotion

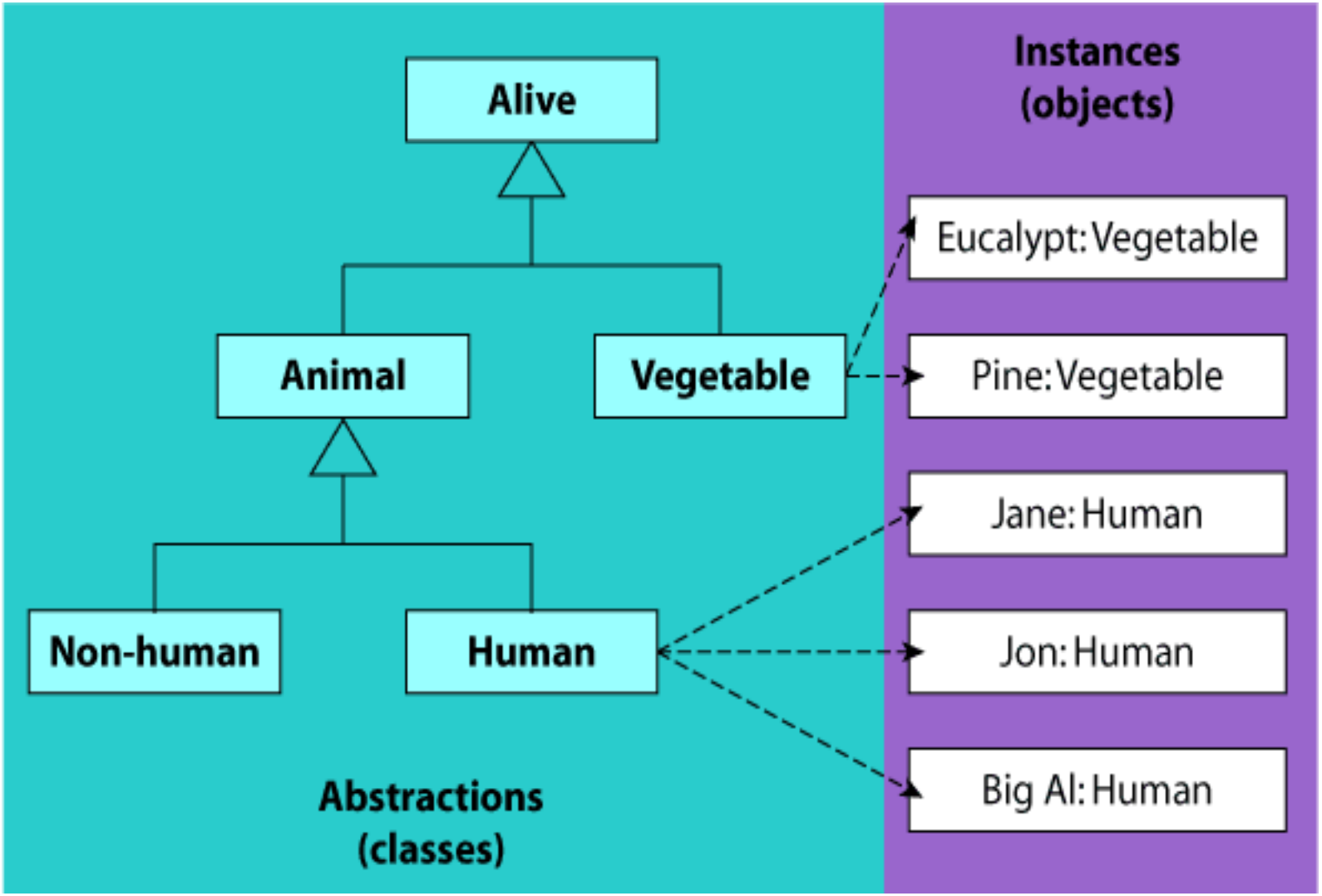
Abstraction – Example 1



Abstraction – Example 1



Abstraction – Example 1

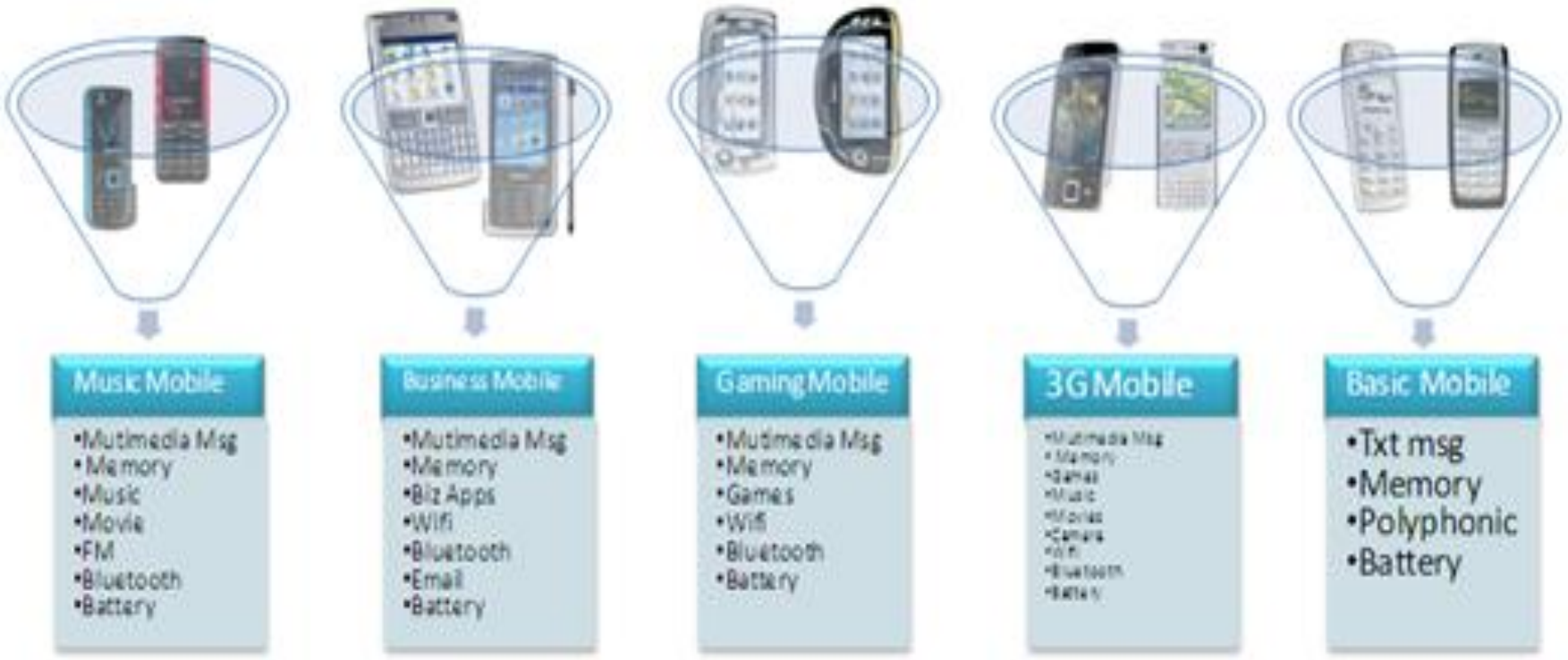


Abstraction – Example 2

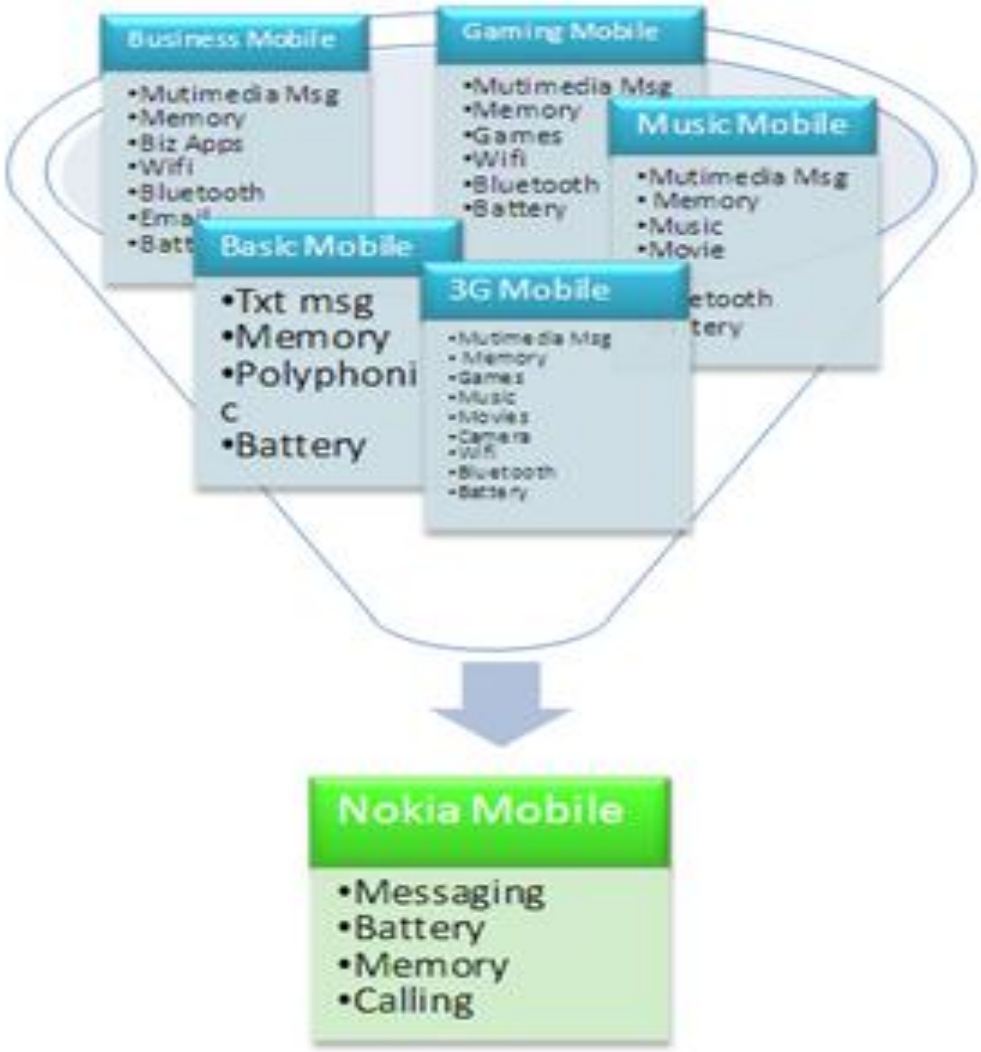
- Its all about Nokia Mobiles
- You could see Slide, Flip and Bar models
- You could see Music, Business & 3G Mobiles
- The Color, Size, Weight, Look etc…

Abstraction – Example 2

Abstraction – Example 2



Abstraction – Example 2



Methods of Abstraction

- Abstraction by Generalization
- Abstraction by Classification
- Abstraction by Aggregation

- In Examples 1 and 2;
  - Example 1 starts with generalization and ends up in classification
  - Example 2 starts with classification and ends up with generalization

Abstraction by Aggregation

- Consider E-builder;
  - We start by thinking about "**Employee**"
  - A collection of Employees become a "**Team**"
  - A collection of Teams become a "**Project**"
  - A collection of Projects become a "**Department**"
  - A collection of Departments become an "**Organization**"

Abstraction by Aggregation

A collection of Toys is a Bin, A collection of Bins is a Rack

Encapsulation

No Encapsulation

```
 2
 3 public class Time {
 4
 5     // ...
 6     public int hours;
 7     public int minutes;
 8     // ...
 9
10 }
11
12
```

Public attributes; visible and can be MODIFIED OUTSIDE "**Time**" class

Encapsulation – Step 1

```
public class Time {
    private int hours;
    private int minutes;

    public Time(int hours, int minutes) {
        this.hours   = hours;
        this.minutes = minutes;
    }

    public final void setHours(int hours) {
        this.hours = hours;
    }
    public final void setMinutes(int minutes) {
        this.minutes = minutes;
    }
    public int getHours() {
        return this.hours;
    }
    public int getMinutes() {
        return this.minutes;
    }
}
```

Private Attributes

Setters

Getters

Encapsulation – Step 2

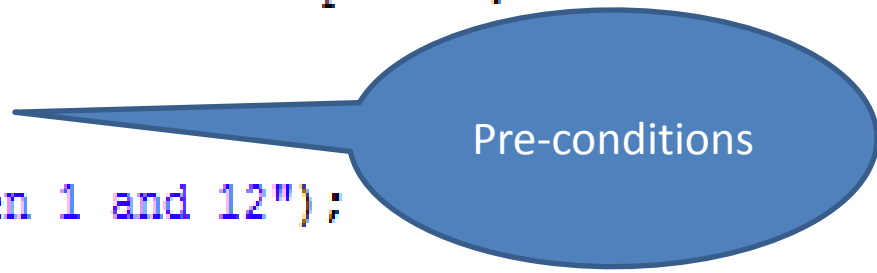
```
public class Time {
    private int hours; // 1-12
    private int minutes; // 0-59

    public Time(int hours, int minutes) throws Exception {
        if (hours < 1 || hours > 12) {
            throw new Exception(
                    "Hours must be between 1 and 12");
        }
        if (minutes < 0 || minutes > 59) {
            throw new Exception(
                    "Minutes must be between 0 and 59");
        }
        this.hours = hours;
        this.minutes = minutes;
    }
```

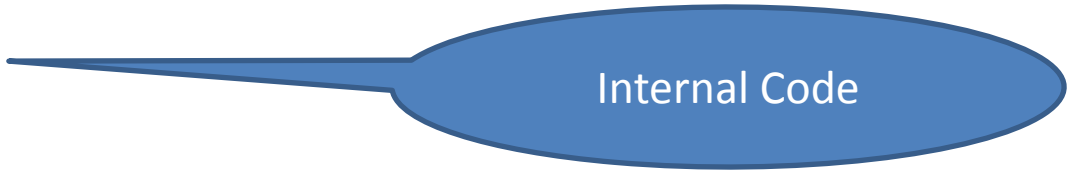
Pre-conditions

Internal Code

Encapsulation – Step 3

```
public final void setHours(int hours) throws Exception {
    if (hours < 1 || hours > 12) {
        throw new Exception(
                "Hours must be between 1 and 12");
    }
    this.hours = hours;
}

public final void setMinutes(int minutes) throws Exception {
    if (minutes < 0 || minutes > 59) {
        throw new Exception(
                "Minutes must be between 0 and 59");
    }
    this.minutes = minutes;
}
```





Encapsulation – Step 4

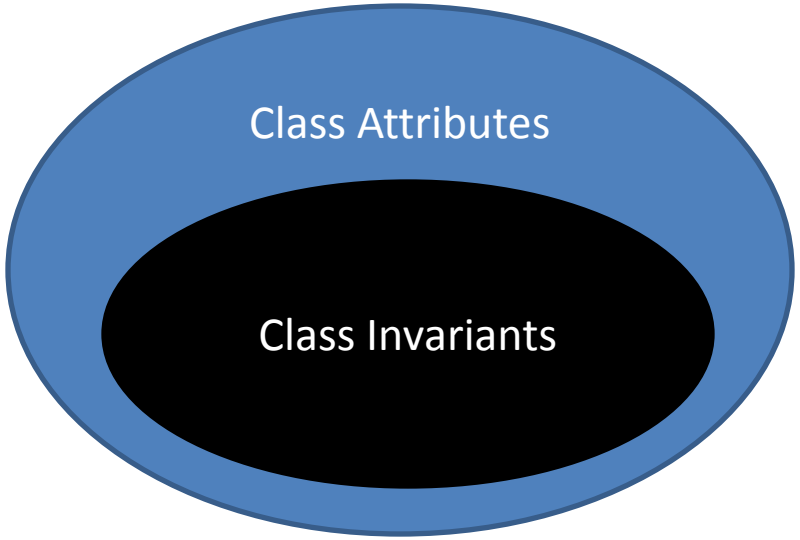
```
public int getHours() {
    int latestHoursValue = this.hours;
    if (latestHoursValue < 1 || latestHoursValue > 12) {
        throw new RuntimeException(
                "Hours must be between 1 and 12");
    }
    return latestHoursValue;
}
public int getMinutes() {
    int latestMinutesValue = this.minutes;
    if (latestMinutesValue < 1 || latestMinutesValue > 59) {
        throw new RuntimeException(
                "Minutes must be between 0 and 59");
    }
    return latestMinutesValue;
}
}
```

Internal Code

Post-conditions

Encapsulation

- Design by Contract
  - Pre-conditions
  - Post-conditions
  - Class Invariants



Design by Contract

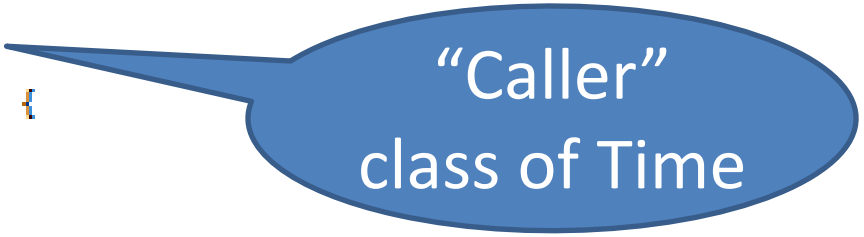
- A **contract** between the **callee** (Time) and the **caller** (ClockSimulator)
- Terms of this Contract are;
  1. If the **values of parameters** for a method (i.e. setHours) **confirms with its pre-conditions** then the method **MUST** return its **intended result**
  2. Otherwise if the **values of parameters** for a method **DON'T** confirm to its **pre-conditions** then the method **MUST** return **ONLY one out of the specified errors**

Why Design by Contract?

- If **ALL** your modules **confirm** with "**Design by Contract**";
  - when there is a bug, can **locate** which module created the bug

Encapsulation – Step 5

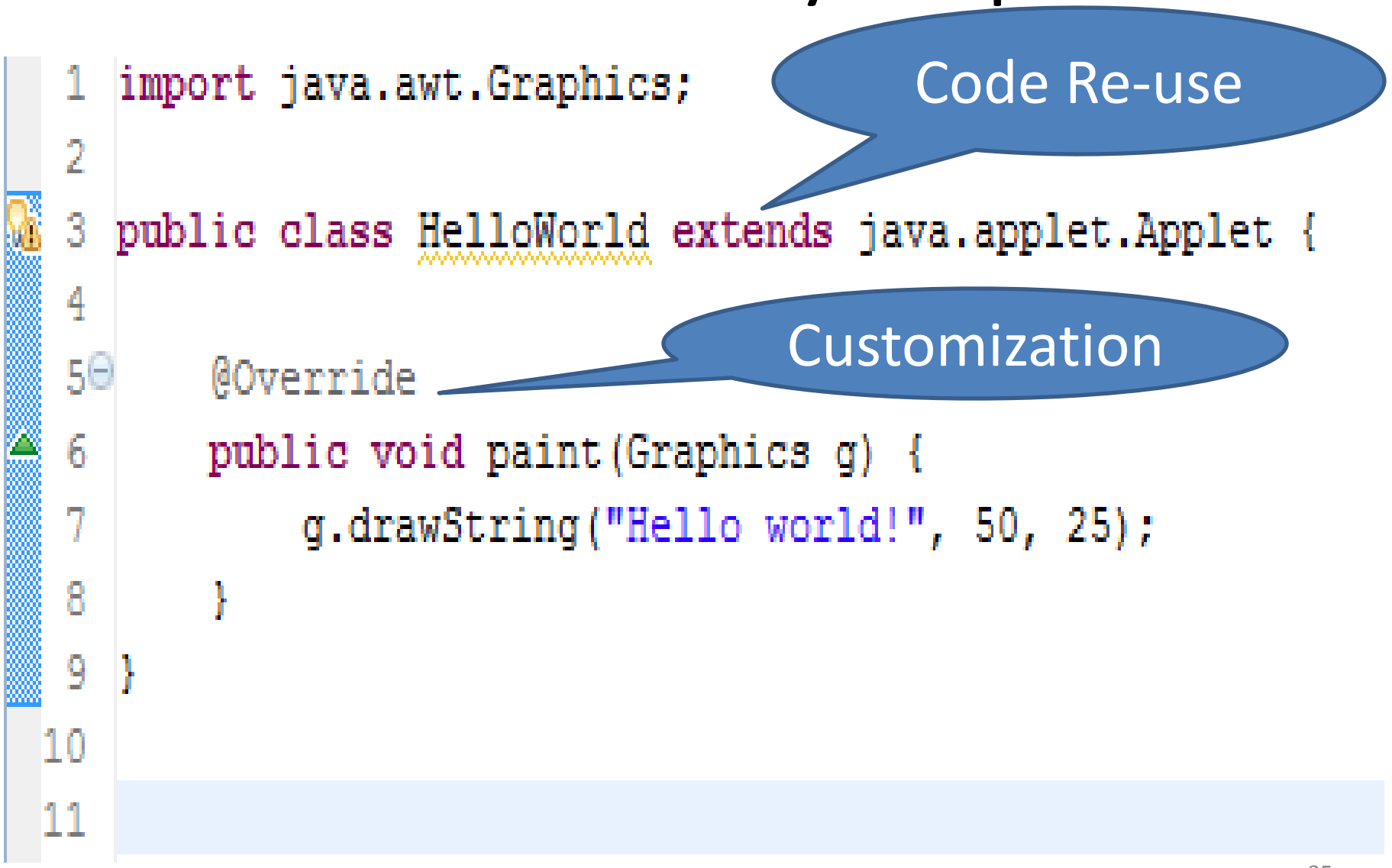
```
public class ClockSimulator {
    public void someMethod1() {
        Time currTime = null;
        try {
            currTime = new Time(5, 30);
            // ...
            // some code goes here
            // ...
            currTime.setHours(90); // can you see my bug
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
    public void someMethod2(Time currTime) {
        // if an exception pops at this point
        // bug is in the time module
        currTime.getHours();
    }

}
```



Encapsulation

- Where is the bug?
  - If a pre-condition fails, the bug lies within the caller class (i.e. ClockSimulator)
  - If a post-condition fails, the bug lies within the callee class (i.e. Time)
- What about exception types?
  - For pre-conditions use compile time exceptions
  - For post-conditions use runtime exceptions

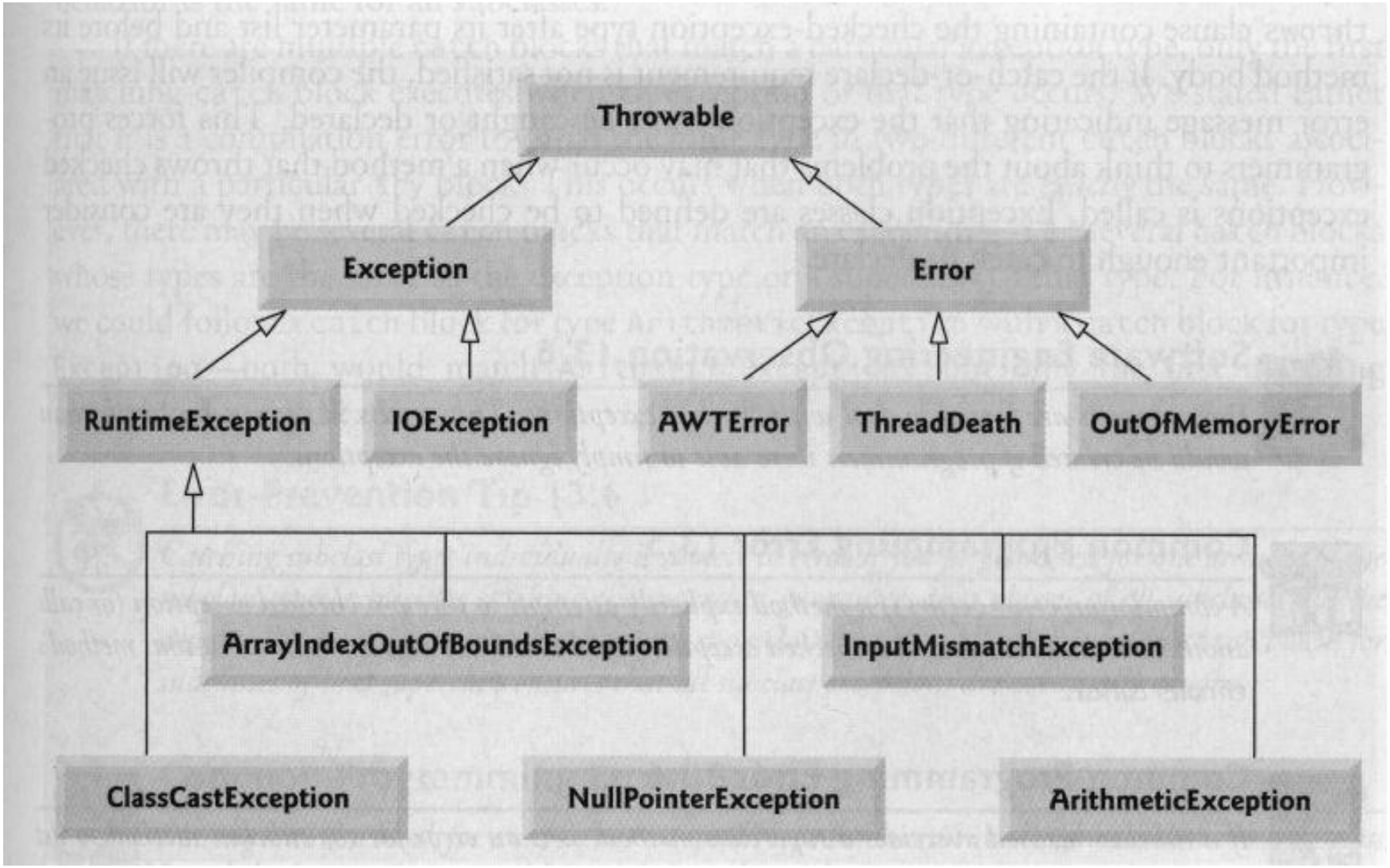
Inheritance & Polymorphism



Inheritance & Polymorphism

- "**java.applet.Applet**" class contains all code for running an applet in a web browser
- Why not "**reuse"** the same code with "**Inheritance**" when writing "**our own applet**"?
- How do we "**customize**" the applet to suit our needs?
- "**Polymorphism**" enables "**customization**" while "**re-using**" code with "**Inheritance**"

Inheritance with Code Re-use



Inheritance with Code re-use

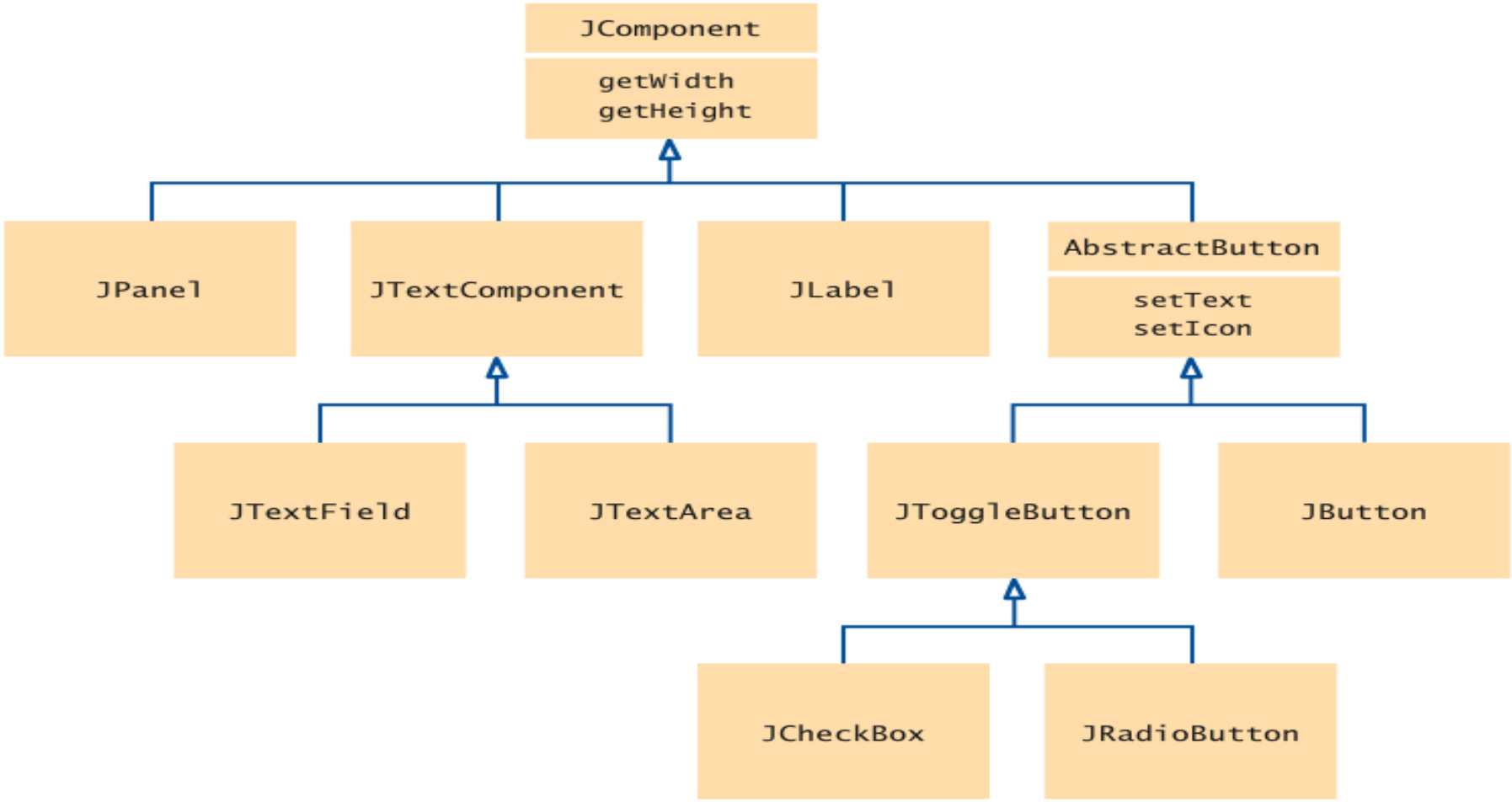


Figure 2 A Part of the Hierarchy of Swing User Interface Components

Inheritance

- Inheritance is for "**code re-use**"
- But **USE WITH CARE**

Misusing Inheritance

- Case 1:
  - Confusing between **IS-A**, **HAS-A**
    - E.g. Implementing my own **Queue** by **reusing java.util.ArrayList**
    - Hope you know that Queue is a data structure with **FIFO** (First-in-First-Out functionality)

Writing a Queue – "**IS-A**" Approach

```
1  import java.util.ArrayList;
2
3  public class Queue extends ArrayList {
4
5      /**
6       * Adds the given object to the end of the queue
7       * @param object Object to add to the end of the Queue
8       */
9      public void enQueue(Object object) {
10         super.add(object);
11     }
12
13     /**
14      * Removes and returns the first object of the queue
15      * @return The first object of the queue
16      */
17     public Object deQueue() {
18         return super.remove(0);
19     }
20 }
```

Writing a Queue – "**HAS-A**" Approach

```
1  import java.util.ArrayList;
2
3  public class Queue {
4      private ArrayList list = new ArrayList();
5
6      /**
7       * Adds the given object to the end of the queue
8       * @param object Object to add to the end of the Queue
9       */
10     public void enQueue(Object object) {
11         list.add(object);
12     }
13
14     /**
15      * Removes and returns the first object of the queue
16      * @return The first object of the queue
17      */
18     public Object deQueue() {
19         return list.remove(0);
20     }
21 }
22
```

Writing a Queue

- What is the correct approach?
  - "**IS-A**" or "**HAS-A**"

- The real problem here is that;
  - Whether a Queue **is a** kind of a java.util.ArrayList?
  - Whether a Queue internally **has a** java.util.ArrayList?

Why "**IS-A**" Approach **DOESN'T** work?

```
1  package nooo;
2
3  public class SomeBusinessLogicClass {
4      public void someMethod() {
5          Queue someQueue = new Queue();
6          SomeDomainClass obj1 = new SomeDomainClass();
7          someQueue.enQueue(obj1);
8          SomeDomainClass obj2 = new SomeDomainClass();
9          someQueue.enQueue(obj2);
10
11         // After some code
12         // ...
13
14         SomeDomainClass obj3 = new SomeDomainClass();
15         // How can you add an object to the middle of the queue?
16         // But it is allowed with Approach 1
17         // add(index, object) is a method inherited from java.util.ArrayList
18         someQueue.add(2, obj3);
19     }
20 }
21
22 class SomeDomainClass { }
23
```

Misusing Inheritance

- About "**Liskov Substitution Principle**";
  - "Methods that use references to the base classes must be able to use the objects of the derived classes without knowing it"
  - In other words;
    - the subtypes must be replaceable for the super type references without affecting the program execution.

Misusing Inheritance(violating LSP)

```
01  class Bird {
02    public void fly(){}
03    public void eat(){}
04  }
05  class Crow extends Bird {}
06  class Ostrich extends Bird{
07    fly(){
08      throw new UnsupportedOperationException();
09    }
10  }
11
12  public BirdTest{
13    public static void main(String[] args){
14      List<Bird> birdList = new ArrayList<Bird>();
15      birdList.add(new Bird());
16      birdList.add(new Crow());
17      birdList.add(new Ostrich());
18      letTheBirdsFly ( birdList );
19    }
20    static void letTheBirdsFly ( List<Bird> birdList ){
21      for ( Bird b : birdList ) {
22        b.fly();
23      }
24    }
25  }
```

Misusing Inheritance

- What is the root cause for this problem?
  - Work done during Abstraction seems incomplete
  - Abstraction has not been properly captured during;
    - Abstraction by Generalization
    - Abstraction by Classification

One Possible Fix !!!

- Redesign the class hierarchy

```
23
24 class Bird{
25       public void eat(){}
26 }
27 class FlightBird extends Bird{
28       public void fly(){}
29 }
30 class NonFlightBird extends Bird{}
31 class Crow extends FlightBird {}
32 class Ostrich extends NonFlightBird {}
```

Polymorphism

- Same name, multiple behaviours
  1. Overloading
  2. Overriding
  3. Program to Interface

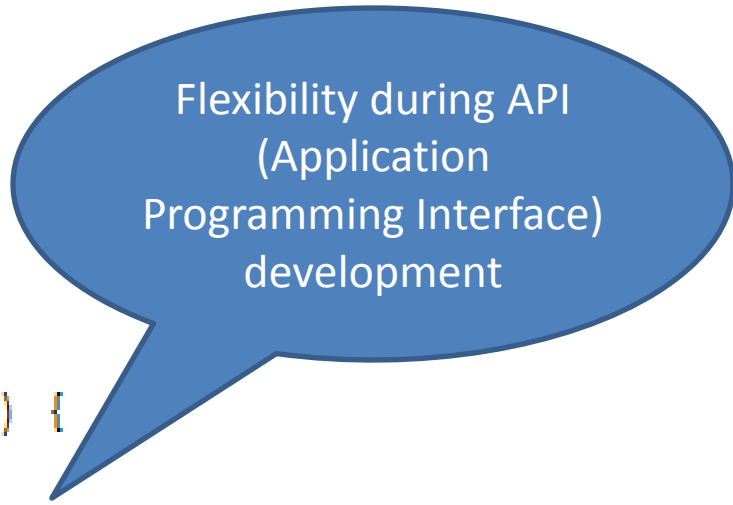
Polymorphism I - Overloading

```
 3 enum Unit { cm, m, km }
 4 public class Length {
 5
 6     private float value;
 7     private Unit unit;
 8
 9     // ...
10     // Constructor Overloading
11     public Length(float value, Unit unit) {
12         this.value = value;
13         this.unit = unit;
14     }
15     public Length(float value) {
16         this(value, Unit.m);
17     }
```

Flexibility during Object Creation

Polymorphism I - Overloading

```
19      // ...
20      // Method Overloading
21      public void add(float value) {
22          this.add(value, Unit.m);
23      }
24      public void add(float value, Unit unit) {
25          // ...
26      }
27      public void add(Length length) {
28          this.add(length.getValue(), length.getUnit());
29      }
30
```



Polymorphism II - Overriding

```
3  class ReportGenerator {
4
5      protected void generateHeader() { /* Header code for any general report */ }
6      protected void generateFooter() { /* Footer for any general report */ }
7      protected void generateMargins() { /* Margins for any general report */ }
8      protected void tabulate() { /* Tabulating statistics for any general report */ }
9      protected void generateReportBody() { /* Report body for any general report */ }
10
11     public void generateReport() {
12         // ...
13         this.generateHeader();
14         // ...
15         this.tabulate();
16         // ...
17         this.generateReportBody();
18         // ...
19         this.generateMargins();
20         // ...
21         this.generateFooter();
22     }
23 }
```

Polymorphism II - Overriding

```
25 class TraciReportGenerator extends ReportGenerator {
26     @Override
27     protected void generateHeader() { /* Custom Header for the Traci report */ }
28 }
29
30 class CodeReviewReportGenerator extends ReportGenerator {
31     @Override
32     protected void tabulate() { /* Custom Tabulation for code review report */ }
33     @Override
34     protected void generateFooter() { /* Custom Footer for the code review report */ }
35 }
```

CUSTOMIZED behaviours for Code Review Report

Polymorphism II & Yo-yo Problem

```
37 class ReportGenerationService {
38
39     private ReportGenerator[] reportGenerators = {
40         new TraciReportGenerator(),
41         new CodeReviewReportGenerator()
42     };
43
44     public void triggerReporting() {
45         // ...
46         try {
47             // ...
48             for(int i = 0; i < reportGenerators.length; i++) {
49                 reportGenerators[i].generateReport();
50             }
51             // ...
52         } catch(Exception ex) {
53             // Suppose the exception pops up at line 49;
54             // then the bug is in any of these report generators
55             // Got to debug harder to find the real cause
56         }
57         // ...
58     }
59 }
60
```

Yo-yo Problem - Worst Case

```
36
37 class PaginatedEmployeeReportGenerator extends ReportGenerator {
38     private int pageSize = 10;
39     public PaginatedEmployeeReportGenerator(int pageSize) { this.pageSize = pageSize; }
40
41     @Override
42     protected void generateReportBody() { /* Report body customized for Pagination */ }
43     @Override
44     protected void tabulate() { /* Tabulation customized for Pagination */ }
45
46     @Override
47     public void generateReport() {
48         /* Say we have 1027 employees */
49         int pages = 1027 / pageSize;
50         for(int i = 0; i < pages; i++) {
51             super.generateReport();
52         }
53     }
54 }
```

Polymorphism III – Program to Interface

Interface is used here, because it doesn't make sense to provide a default way for instrument to play or tune - too much variety

```
13
14 interface Instrument
15 {
16     public void play(double volume);
17     public void tune();
18 }
19
```

Note these methods have no body, because it doesn't make sense to give generic behavior

Polymorphism III – Program to Interface

We're NOT overriding play() because there is NO default definition, we are IMPLEMENTING it

```
19
20 class Guitar implements Instrument
21 {
22    public void play(double volume) { /* play guitar music here */ }
23    public void tune() { /* do guitar tuning here */ }
24 }
25
26 class Piano implements Instrument
27 {
28    public void play(double volume) { /* play piano music here */ }
29    public void tune() { /* do piano tuning here */ }
30 }
31
```

Polymorphism III – Program to Interface

```
32 class InstrumentSimulator {
33     public static void main(String[] args) {
34         Instrument[] instruments = { new Guitar(), new Piano() };
35         for(int i = 0; i < instruments.length; i++) {
36             instruments[i].tune();
37             instruments[i].play(1.0);
38         }
39     }
40 }
```

Polymorphism III – Program to Interface

```
22
23  interface Weapon
24  {
25      public void attack();
26  }
27
```

Different weapons can attack in many different ways, NO DEFAULT attack description

Polymorphism III – Program to Interface

Guitar can now have seemingly unrelated behavior of Instrument and Weapon - can't do this with inheritance

```
29 class Guitar implements Instrument, Weapon
30 {
31     public void play(double volume) { /* play guitar music here */ }
32     public void tune() { /* do guitar tuning here */ }
33
34     public void attack() { /* swing guitar at opponent or victim */ }
35 }
```

Polymorphism III – Program to Interface

```
37 class Piano implements Instrument, Weapon
38 {
39     public void play(double volume) { /* play piano music here */ }
40     public void tune() { /* do piano tuning here */ }
41
42     public void attack() { /* drop piano from window onto victim's head */ }
43 }
44
```

Polymorphism III – Program to Interface

```
45 class Simulator {
46     public static void main(String[] args) {
47 
48         Guitar guitar = new Guitar();
49         Piano piano = new Piano();
50 
51         Instrument[] instruments = { guitar, piano };
52         for(int i = 0; i < instruments.length; i++) {
53             instruments[i].tune();
54             instruments[i].play(1.0);
55         }
56 
57         Weapon[] weapons = { guitar, piano };
58         for(int j = 0; j < weapons.length; j++) {
59             weapons[j].attack();
60         }
61     }
62 }
63 
```

Q & A