Inheritance (an intuitive description)

- Recall the Orange class
  - properties found in Orange are also shared with other Fruits (e.g. Apple, Banana, Pineapple)

- We associate behavior as well as state with more abstract notions (e.g. Fruit). Oranges are a specialization of that abstraction.

- In OO programming inheritance is a relationship between entities referred to as parents and children where
  - the behavior and data associated with the child classes are always an extension of the properties associated with parent classes.
Inheritance (an intuitive description)

- **A child class**
  - will be given all the properties of the parent class
  - may in addition define new properties of its own
  - may redefine some of the properties of the parent class to
    - constrain
    - override

- **Inheritance is transitive**
  - if we have Dog inherits from Mammal and Mammal inherits from Animal then Dog has behavior defined in both Animal and Mammal
Inheritance in Java (the Object class)

- The “mother” of all classes in Java is the Object.

\[
\text{public class Orange} \{ \\
\quad \ldots \\
\} \\
\]

\[
\text{public class Orange extends Object} \{ \\
\quad \ldots \\
\}
\]

- `extends` in Java defines an inheritance relationship between Object (parent) and Orange (child)
- Every Java class inherits from Object
- Java classes can have
  - at most one parent class
  - zero or more child classes
Inheritance diagrammatically

- Use an empty headed arrow, arrow points to parent class

![Diagram showing inheritance relationship between parent and child classes](image)
Inheritance and terminology

- **Superclass**
  - refers to the parent class from which code is inherited

- **Subclass**
  - refers to the child class that code was inherited to.
Inheritance and its forms

- **Specialization**
  - child class is a special case of the parent class; the child is a *subtype*.

- **Specification**
  - parent class defines behavior that is *implemented* in the child class

- **Construction**
  - child class makes use of the behavior found in the parent class *but* the child is not a subtype

- **Extension**
  - child class adds new functionality and does not change the inherited behavior
Inheritance and its forms (cont)

• Limitation
  - child class restricts the usage of some of the behavior found in the parent class

• Combination
  - child class inherits features from more than one parent
    • Java does not *directly* support this last form, although we can simulate it (more on this next time)

• Address each form separately with examples in Java.
Specialization

- child class is a special case of the parent class; the child is a subtype.

- same behavior as Orange
  - extra instance variable
  - extra constructor method

- All other instance methods are inherited from Orange
Specialization (cont)

• super is Java keyword and denotes the superclass (i.e. Orange) constructor method

• super can be used to call methods defined in the superclass
  - e.g. super.showInfo()

```java
public class ShelfOrange extends Orange{
    int lifetime;
    ShelfOrange(int newWeight, int newPrice,
                int mylifetime){
        super(newPrice, newWeight);
        this.lifetime = mylifetime;
    }
}
```
Type, subtype and supertype

• Subtype
  – *Type S is a subtype of type T if an instance of type S can be substituted for an instance of type T with no observable effect.*

• This means
  – an instance of S can understand the same messages as an instance of T
    – for any method in T, there is a corresponding method in S with the same name, same number of arguments and same types for each argument.
  – S can have more method definitions but not less.
  – T is the supertype of S.
Type, subtype and supertype (cont)

```java
public class Main {
    public static void main(String[] args) {
        Orange simpleOrange = new Orange(2, 3);
        ShelfOrange shelfOrange = new ShelfOrange(4, 5, 3);
        simpleOrange.showInfo();
        shelfOrange.showInfo();
        // casting forces shelfOrange to be manipulated as an Orange
        Orange pretender = (Orange) shelfOrange;
        shelfOrange.showInfo();
        // this still works, the message is understood
        // and the same info as line 7 is displayed.
    }
}
```
Type, subtype and supertype

• Widening and Narrowing
  – Conversion of a subtype to one of its supertypes is called **widening**
  – Conversion of a supertype to one of its subtypes is called **narrowing**

• Rule of assignment
  – The type of the expression at the right-hand side of an assignment must be a subtype of the type of the variable at the left-hand side of the assignment.

  – e.g. Orange pretender = new ShelfOrange(2,3,4)
Specification

- Parent class defines behavior that is implemented in the child class
- There are two ways that you can impose this on Java programs
  - abstract classes
  - interfaces
- Abstract classes
  - cannot be instantiated
  - contain instance variables, instance methods etc.
  - methods can be declared abstract
    - their implementation is deferred and has to be defined by subclasses
abstract class Fruit{
    int weight;
    int price;

    public void setWeight(int anInt) {
        weight = anInt;
    }

    public void setPrice(int anInt) {
        price = anInt;
    }

    public int getWeight() {
        return weight;
    }

    public int getPrice() {
        return price;
    }

    abstract public void prettyPrint();
}
Specification (cont)

- No constructor
- `prettyPrint()` is defined to be abstract and no implementation is provided in `Fruit`
Orange has to provide an implementation for `prettyPrint()`. The method signature must be identical to the one found in `Fruit`.

```java
public class Orange extends Fruit{
    Orange(int aweight, int aprice){
        this.price = aprice;
        this.weight = aweight;
    }

    public void prettyPrint(){
        System.out.println("This is an Orange of weight "+weight+" and Price "+ price);
    }
}
```
Construction

- child class makes use of the behavior found in the parent class but the child is not a subtype
- typically used to simplify implementation
- the two classes might be completely unrelated concepts

```
<table>
<thead>
<tr>
<th>RTriangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>- rSide: int</td>
</tr>
<tr>
<td>- lSide: int</td>
</tr>
<tr>
<td>- hypo: double</td>
</tr>
<tr>
<td>+ getArea(): double</td>
</tr>
<tr>
<td>+ getPerimeter(): double</td>
</tr>
</tbody>
</table>

Square

Square(int)
```
public class RTriangle{
    private int rSide;
    private int lSide;
    private double hypo;

    RTriangle(int sideA, int sideB, double sideC){
        this.rSide = sideA;
        this.lSide = sideB;
        this.hypo = sideC;
    }

    public double getArea(){
        return (rSide*lSide)/2.0;
    }

    public double getPerimeter(){
        return rSide+lSide+hypo;
    }
}

class Square extends RTriangle{
    Square(int sideA){
        super(sideA,sideA, Math.sqrt(2*(sideA*sideA)));
    }

    public double getArea(){
        return 2*super.getArea();
    }

    public double getPerimeter(){
        return (2*super.getPerimeter()) - (2*getHypo());
    }
}
Construction (cont)

• Instances of Square cannot be substituted freely with instances of RTriangle

• The usage of RTriangle is merely for making implementation easy since we can reuse code that is already there and tested.

• This usage of inheritance is sometimes frowned upon since it breaks substitutability.
Extension

- child class adds new functionality and does not change the inherited behavior

```java
public class ShelfOrange extends Orange{
    int lifetime;
    ShelfOrange(int newWeight, int newPrice,
                int mylifetime){
        super(newPrice, newWeight);
        this.lifetime = mylifetime;
    }

    public void showInfo(int noOfTimes){
        for (int i =0 ; i < noOfTimes;i++)
        {
            prettyPrint();
        }
    }

    public void setLifetime(int newLifetime){
        lifetime = newLifetime;
    }

    public int getLifetime(){
        return lifetime;
    }
}
```
Limitation

- child class restricts the usage of some of the behavior found in the parent class
  - e.g remove the ability to call setter methods in Orange
- An inherited method can be redefined or overridden in a subclass definition.

```java
public class FixedOrange extends Orange {

    //overrides setters
    public void setPrice(){
        System.out.println("FixedOrange does not allow setters");
    }
    public void setWeight(){
        System.out.println("FixedOrange does not allow setters");
    }
}
```
Overriding

• In order to override a method in a subclass
  – the method name must be the same
  – the number of arguments and their corresponding types must be the same
  – the method modifiers must be the same

```java
public class FixedOrange extends Orange{

    //overrides setters
    public void setPrice(){
        System.out.println("FixedOrange does not allow setters");
    }
    public void setWeight(){
        System.out.println("FixedOrange does not allow setters");
    }
}
```
Overloading

- Overloading uses the same method name but different arguments
  - e.g. different number of arguments, different types

```java
public class FixedOrange extends Orange{

    // overrides setters
    public void setPrice(){
        System.out.println("FixedOrange does not allow setters");
    }
    public void setWeight(){
        System.out.println("FixedOrange does not allow setters");
    }

    // overload prettyPrint
    public void prettyPrint(int noOfTimes){
        for (int i =0 ; i < noOfTimes;i++){
            prettyPrint();
        }
    }
}
```