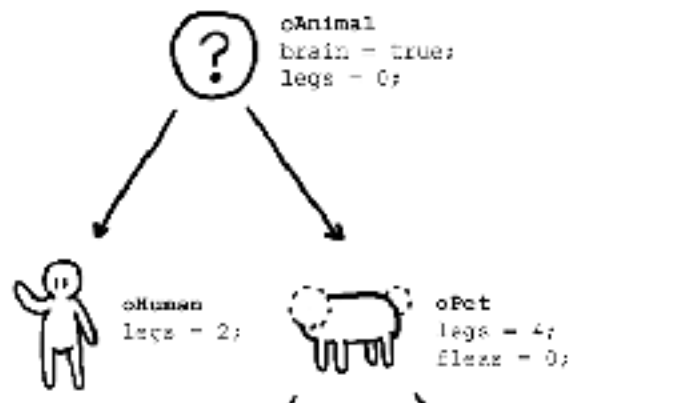
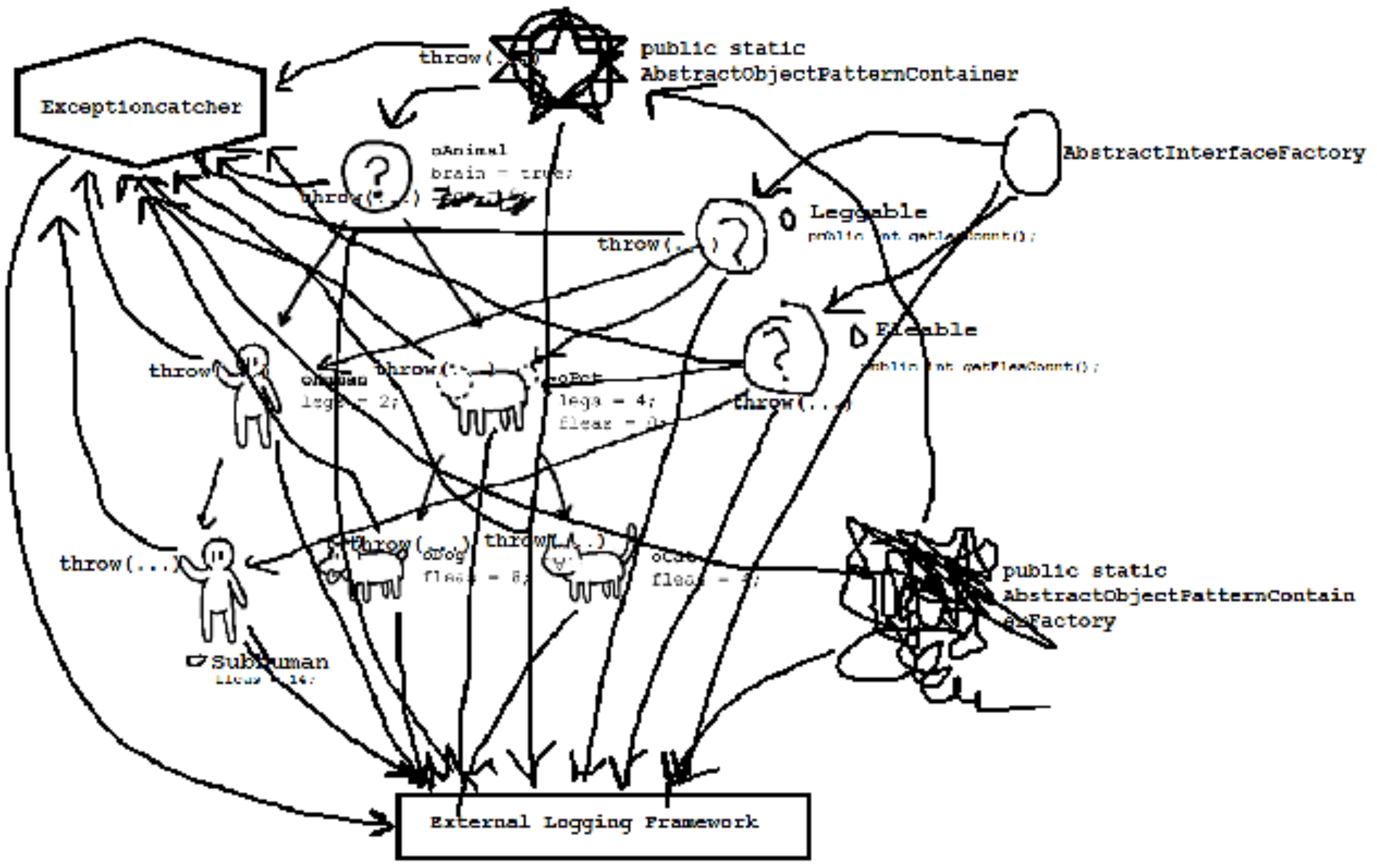
What OOP users claim



What actually happens



COMP 250: Java Object Oriented Programming

January 22-23, 2018

Carlos G. Oliver

Slides adapted from M. Blanchette

Objects behave according to their kind (Type)

Let us turn to Genesis 1:25.

"And God made the beast of the earth after his kind, and cattle after their kind, and every thing that creepeth upon the earth after his kind: and God saw that it was good." -- Genesis 1:25 (King James Bible)

"God made all sorts of wild animals, livestock, and small animals, each able to **produce offspring of the same kind**. And God saw that it was good." -- Genesis 1:25 (New Living Translation)

```
public class SportTeam {
    ... (from previous slides)
}

public class League {
    int nbTeams;
    public SportTeam teams[];   // an array of SportTeam

    League(int n) {       // constructor
     nbTeams = n;
     for (int i = 0 ; i < n ; i++ ) teams[i] = new SportTeam();
    }

    public static void main(String args[]) {
        League NHL = new league(30);
        NHL.teams[0].hometown = “Montreal”;
        NHL.teams[0].addWin();
   }
}
```

This

- Sometimes, it can be useful for an object to refer to itself:
  - the **this** keyword refers to the current object
  - “Within an instance method or a constructor, `this` is a reference to the *current object* — the object whose method or constructor is being called.”
- We could rewrite the constructor as:

```
public SportTeam() {
    this.victories = this.losses = this.points = 0;
    this.homeTown = new String("Unknown");
  }
```

- If there was a league object that needed to be updated:
  - league.addTeam(this);

This example

```
public boolean teamCompare(HockeyTeam h){

    //accessing the getWins attribute of the current ‘this’ object
    and the argument h object

    if (this.getWins() > h.getWins()) return true;

    return false;

}
```

More on when to use **this**: https://stackoverflow.com/questions/2411270/when-should-i-use-this-in-a-class

getters and setters

Remember we made some of the member variables private?

What if we want to view or set them while making sure nothing incorrect is done? —> getters and setters

```
public int getWins(){
    return this.wins;
}
public void setWins(int w){
    if (w > 0) this.wins = w;
    else System.out.println("Cannot have negative number of wins");
```

Static members

- Normally, each object has its own copy of all the members of the class, but...
- Sometimes we want to have members that shared by all objects of a class
- The **static** qualifier in front of a member (or method) means that all objects of that class share the same member
- Static members are **not** accessed through an instance object (no ‘this’)

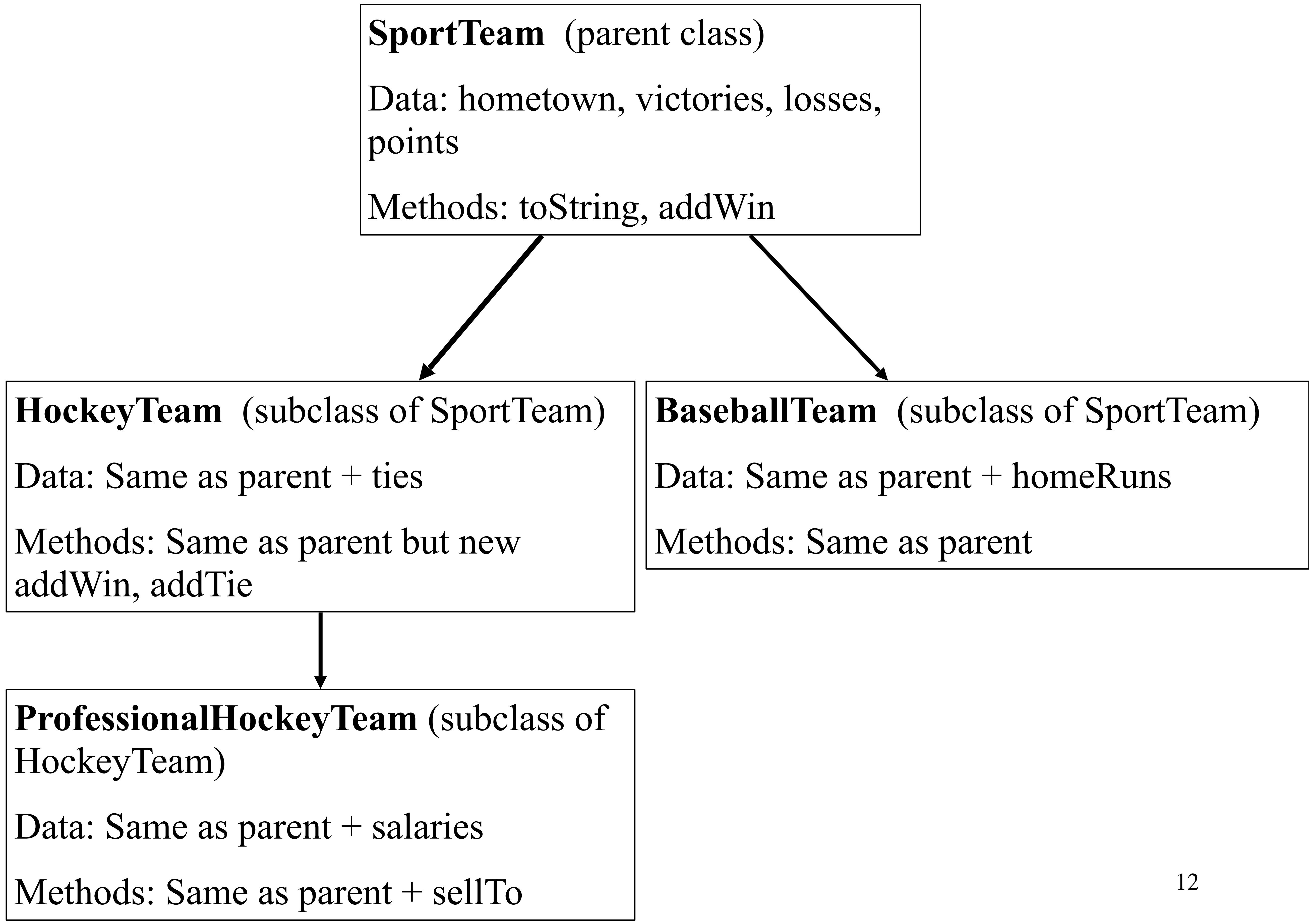
```
public class SportTeam {
   public String homeTown;
   private int victories, losses, points;
   static public double exchangeRate;    /* all objects of type SportTeam share
                                          the same exchangeRate */
   public SportTeam() { /* see previous page */}
   public SportTeam(String town) { /* see previous page */}
   public String toString() { /* see previous page */}
   public addWin() { /* see previous page */}
    public static void main(String[] args) {
       // now we can declare variables of type SportTeam
       SportTeam expos, alouettes;
       SportTeam.exchangeRate = 1.57;    /* static members can be used without
                                             an actual object */
       expos = new SportTeam();
       alouettes = new SportTeam("Montreal");
       expos.exchangeRate = 1.58;                // or from one particular object
       System.out.println("Rate from expos: " + expos.exchangeRate);
       System.out.println("Rate from alouettes: " + alouettes.exchangeRate);
     }
}
```

Inheritance

- Suppose you need to write a class X whose role would be very similar to an existing class Y. You could
  - Rewrite the whole code anew
    - Time consuming, introduces new bugs, makes maintenance a headache
  - Copy the code of Y into X, then make your changes
    - Maintenance problem: you need to maintain both X and Y
  - Inherit the code from Y, but override certain methods
    - Code common to X and Y is kept in Y. New methods are added in X

Inheritance - Example

- You want to extend SportTeam to make it specific to certain sports
  - HockeyTeam
    - Has all the members defined in sportTeam, but also number of ties.
    - Number of points = 3 * victories + 1 * ties
  - BaseballTeam
    - Has all the members defined in SportTeam, but also number of homeruns



SportTeam (parent class)

Data: hometown, victories, losses, points

Methods: toString, addWin

HockeyTeam (subclass of SportTeam)

Data: Same as parent + ties

Methods: Same as parent but new addWin, addTie

BaseballTeam (subclass of SportTeam)

Data: Same as parent + homeRuns

Methods: Same as parent

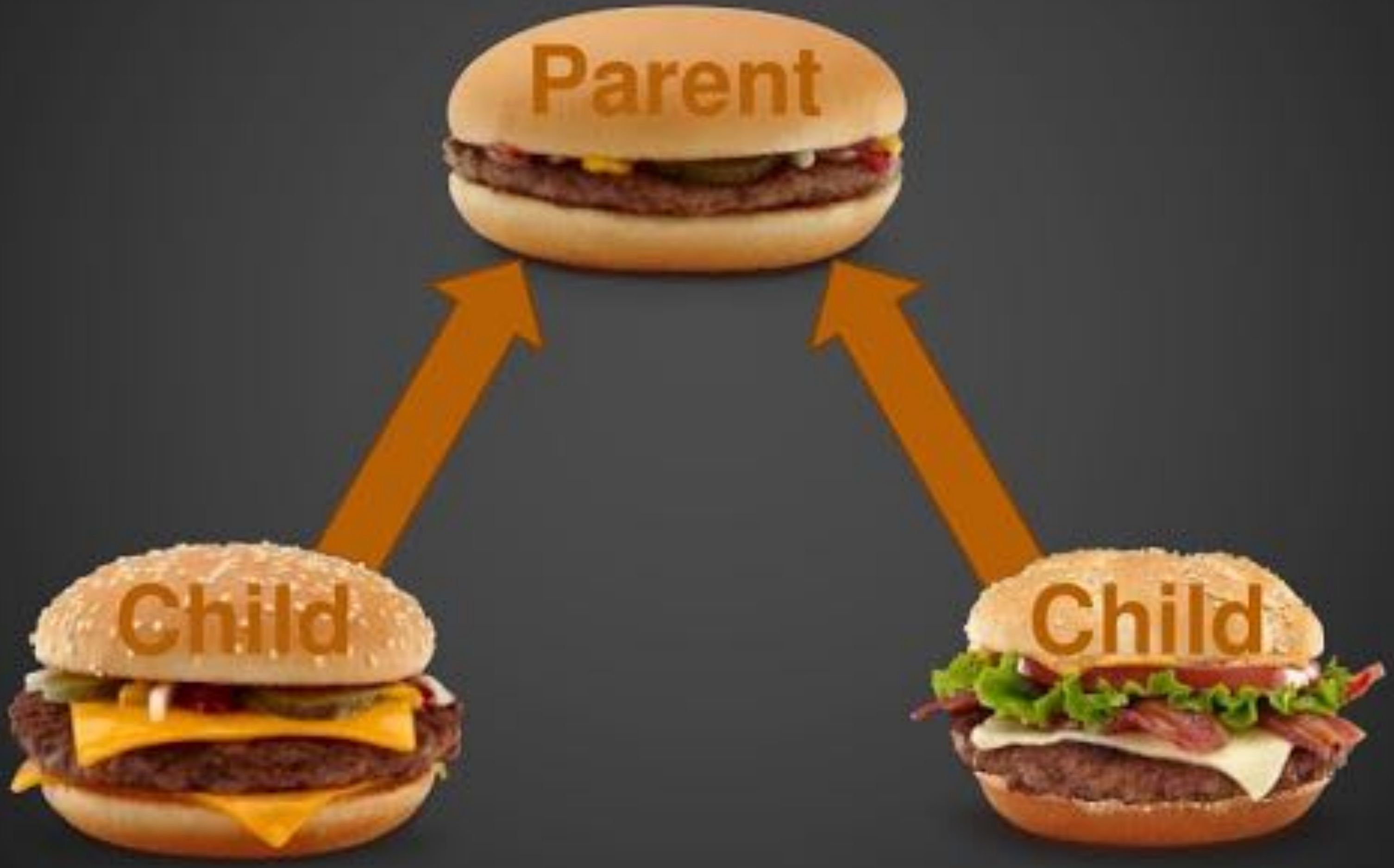
ProfessionalHockeyTeam (subclass of HockeyTeam)

Data: Same as parent + salaries

Methods: Same as parent + sellTo

4 Major Principles for OOP

Inheritance



```
public class HockeyTeam extends SportTeam {
   private int ties;
   public HockeyTeam() {    // constructor for HockeyTeam
      super();                          // super() calls the constructor of the superclass
      ties=0;
    }

    public void addWin() {
       super.addWin();          /* This calls the addWin method provided by the
                    parent class */
        points++;                     /* Since points is private, this wouldn't compile.
                                            We need to declare points as "protected"
                               instead of private to allow access to subclasses */
     }

    public void addTie() {
       ties++;
        points++;
     }
}
```

Types and dispatch

Dispatch is the way the Java links method **calls** to method **definitions.**

```
public static void main(String args[]) {
      HockeyTeam habs;
      habs = new HockeyTeam();
      habs.hometown = "Montreal";
      habs.addWin();                                        /* The addWin method called is the one
                                        from HockeyTeam. habs.points is 3*/
      habs.addTie();                                                      // ties is now 1, points is 4
      System.out.println(habs.toString());    /* HockeyTeam doesn't provide a
                          toString() method but SportTeam
      does, so that's the one called */
     SportTeam bruins = new HockeyTeam();  /* this is legal because HockeyTeam
                                              is a subtype of SportTeam */
      bruins.addWin();    // bruins.points is now 3
      HockeyTeam leafs = new SportTeam();   /* this is NOT legal because
                SportTeam is not a subtype of
      HockeyTeam */
}
```

Overloading vs Overriding

Overloading: multiple methods in the **same class** with the same name, distinguished by arguments, modifiers, return type.

```
public class Dog {

  public void bark(){

    System.out.println("Woof");

  }

  public void bark(int times){

    for int(i = 0; i < times; i++) System.out.println("woof");

  }

}
```

Overloading vs Overriding

Overriding: methods with identical signatures but one in parent and other in child class

```
public class Dog{

public void bark(){

      System.out.println(“woof”);

}

public class Husky extends Dog{

public void bark(){

      System.out.println(“awooooo”);

      }

}
```

Good reference: **https://www.programcreek.com/2009/02/overriding-and-overloading-in-java-with-examples/**

Exceptions - When things go wrong

- Some things are outside programmer's control:
  - User types "Go expos" when asked to enter number of victories
  - Try to open a file that doesn't exist
  - Try to compute sqrt(-1)
  - ...
- Exception mechanism allows to deal with these situations gracefully
  - When problem is detected, the code throws an exception
  - The execution of the program stops. JVM looks for somebody to catch the exception
  - The code that catches the exception handles the problem, and execution continues from there
  - If no code catches exception, the program stops with error message
- An exception is an object that contains information about what went wrong.

Throwing exceptions

```
Syntax:
try {
   <block of code>
}
catch (exceptiontype1 e1) {
   <block of code>
}
catch (exceptiontype2 e2) {
    <block of code>
}
...
finally {
   <block of code>
}
```

```
static double mySqrt(double x) {
   try {
      if (x<=0) throw new
                  ArithmeticException(”Sqrt is defined
                     only for positive numbers");
   /* Code for computing sqrt goes here */
   }
   catch (ArithmeticException e) {
      System.out.println("The mySqrt operation   failed
with error: ” +  e );
      return 0;
   }
}
```

Methods throwing exceptions

- Sometimes, it is not appropriate for a method to handle the exception it threw
- Methods can throw exceptions back to the caller:

```
static double mySqrt(double x)
          throws ArithmeticException {
  if (x<0) {
    throw new ArithmeticException(”Sqrt of "
          + x + " is not defined");
    }
    /* Code for computing sqrt goes here */
}
```

```
public static void main(String args[]) {
  double x = 0, y = 0, z = 0 ;
  try {
    x = mySqrt(10);
    y = mySqrt(-2);
    z = mySqrt(100);
   }
   catch ( ArithmeticException e ) {
     System.out.println(e.toString());
    }
    // what is the value of x, y, z now?
    // x is 1, y and z are zero
```

Java resources

- Java Application Programming Interface (API)

  http://docs.oracle.com/javase/7/docs/api//

- Java books: 1594 different books on Amazon
  - The Java Programming Language -- by Ken Arnold (Author), et al;
    By the authors of Java itself. The ultimate reference. Not easy to read for beginners.
  - Java in a Nutshell, Fourth Edition, by David Flanagan
    A text version of the Java API