IAT 265

OO programming – Inheritance & Polymorphism

Topics

- Recap: Classes & objects

- Inheritance
  - Inheritance and 'is a' relationship
  - *super* and *this* keywords
  - Case study: EatingBug extends Bug

- Polymorphism
  - Overriding polymorphism
  - Inclusion polymorphism

- String and String functions

Recap: Classes

- **Data Types**
  - Primitives (primitive data): int, float, char, boolean ...
  - Reference (Objects): array, string, *class* ...

Objects

- We can make our own objects, to keep related data together, with methods to control those data

Classes

- Classes are the blueprints for our new objects
- To declare a new Class (a new type of objects):

```
class MyToy {
   // fields (member variables)
   // Constructors (methods for instantiation)
   // methods (object functions)
}
```

Fields and Methods

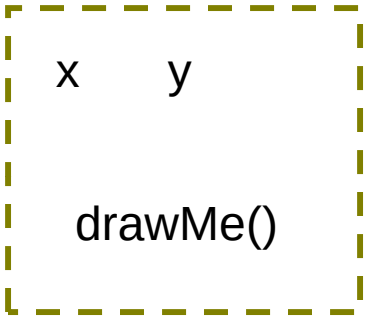
```
class MySquare {
  int xPos, yPos;

  MySquare(int x, int y) {
        xPos = x;
        yPos = y;
  }

  void drawMe() {
        rect(xPos, yPos, 50, 50);
  }
}
```

fields

constructor

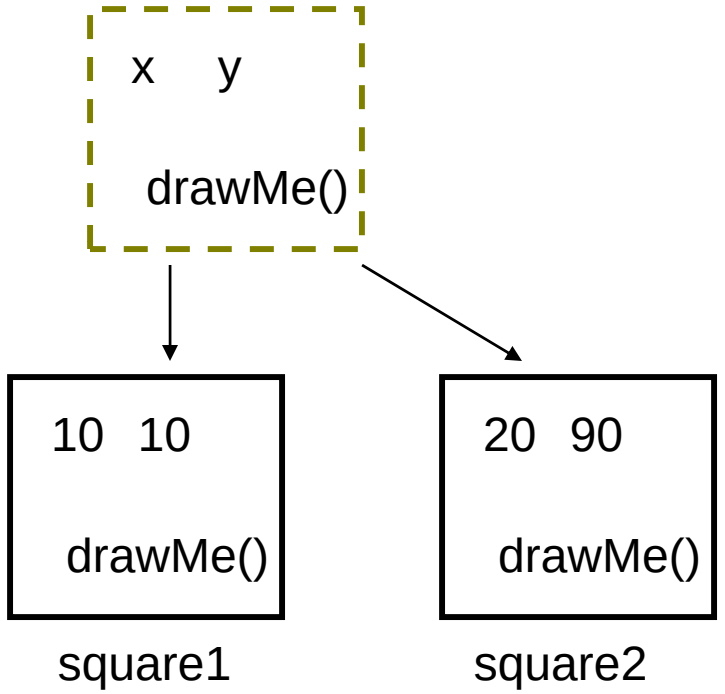
(one kind of method)

methods



Fields and Methods

```
class MySquare {
   int xPos, yPos;

   MySquare(int x, int y) {
         xPos = x;
         yPos = y;
   }

   void drawMe() {
         rect(xPos, yPos, 50, 50);
   }
}
```



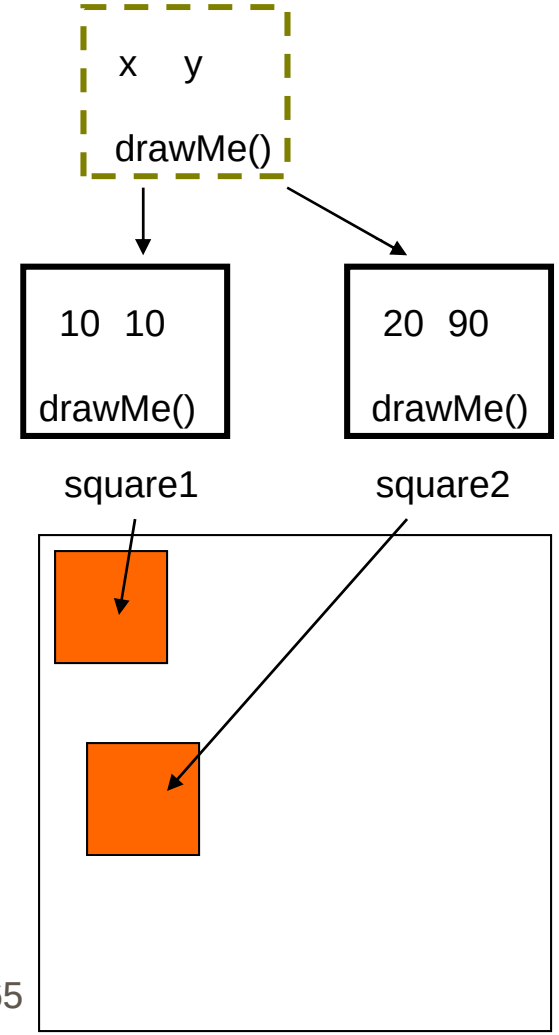

```
MySquare square1 = new MySquare(10, 10);
MySquare square2 = new MySquare(20, 90);
```

Fields and Methods

```
class MySquare {
   int xPos, yPos;

   MySquare(int x, int y) {
         xPos = x;
         yPos = y;
   }

   void drawMe() {
         rect(xPos, yPos, 50, 50);
   }
}
```

```
MySquare square1 = new MySquare(10, 10);
MySquare square2 = new MySquare(20, 90);
```

square1.drawMe();
square2.drawMe();



Arrays of Objects?

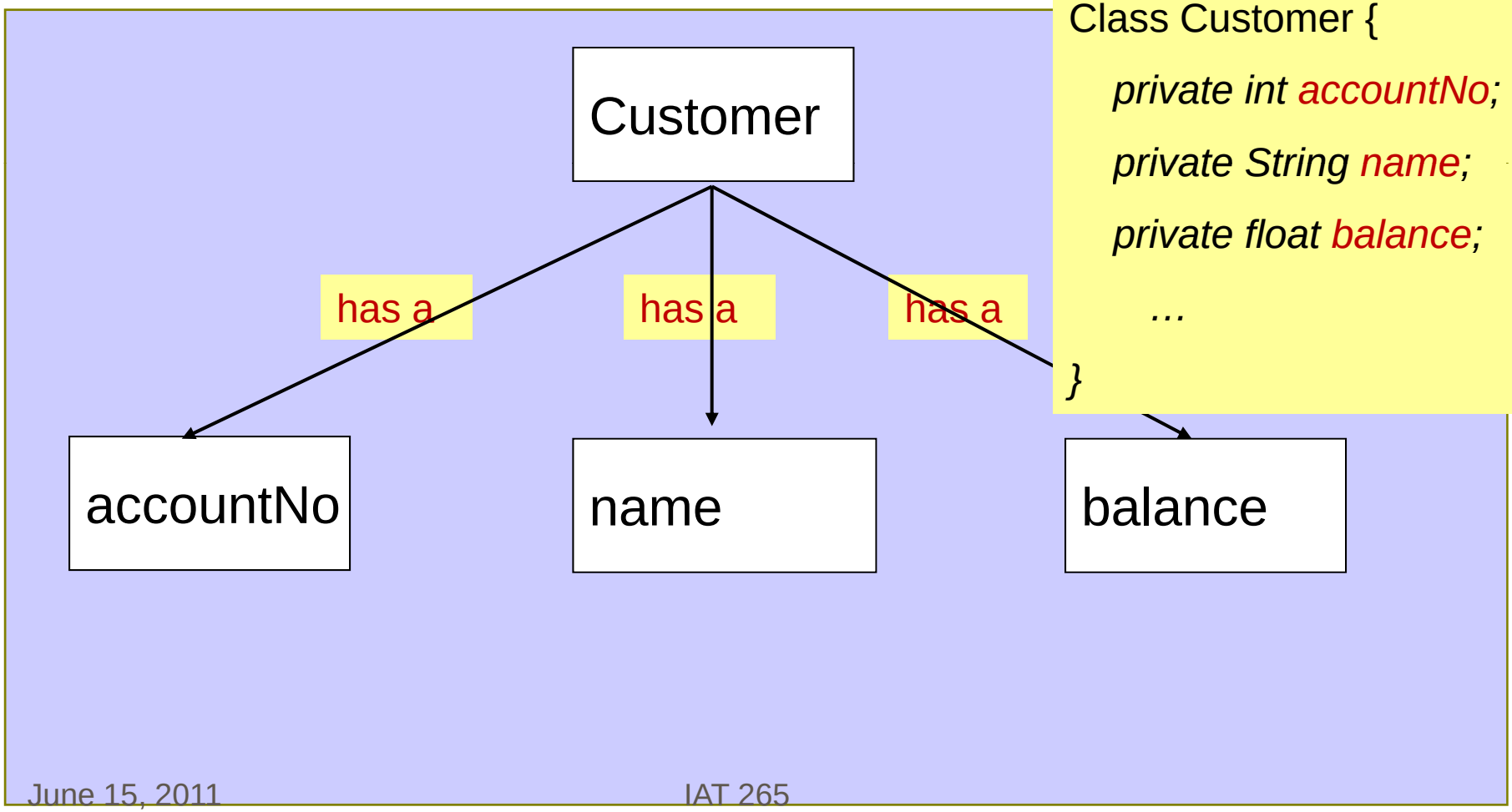
- Let's make a bunch of squares!

```
MySquare[] squares = new MySquare [10] ;

// initialize all of our squares.
for (int i = 0; i < 10; i ++) {
  squares[i] = new MySquare(i*10, i*10);
}

squares[4].drawMe();  // draw the 4th square.
```

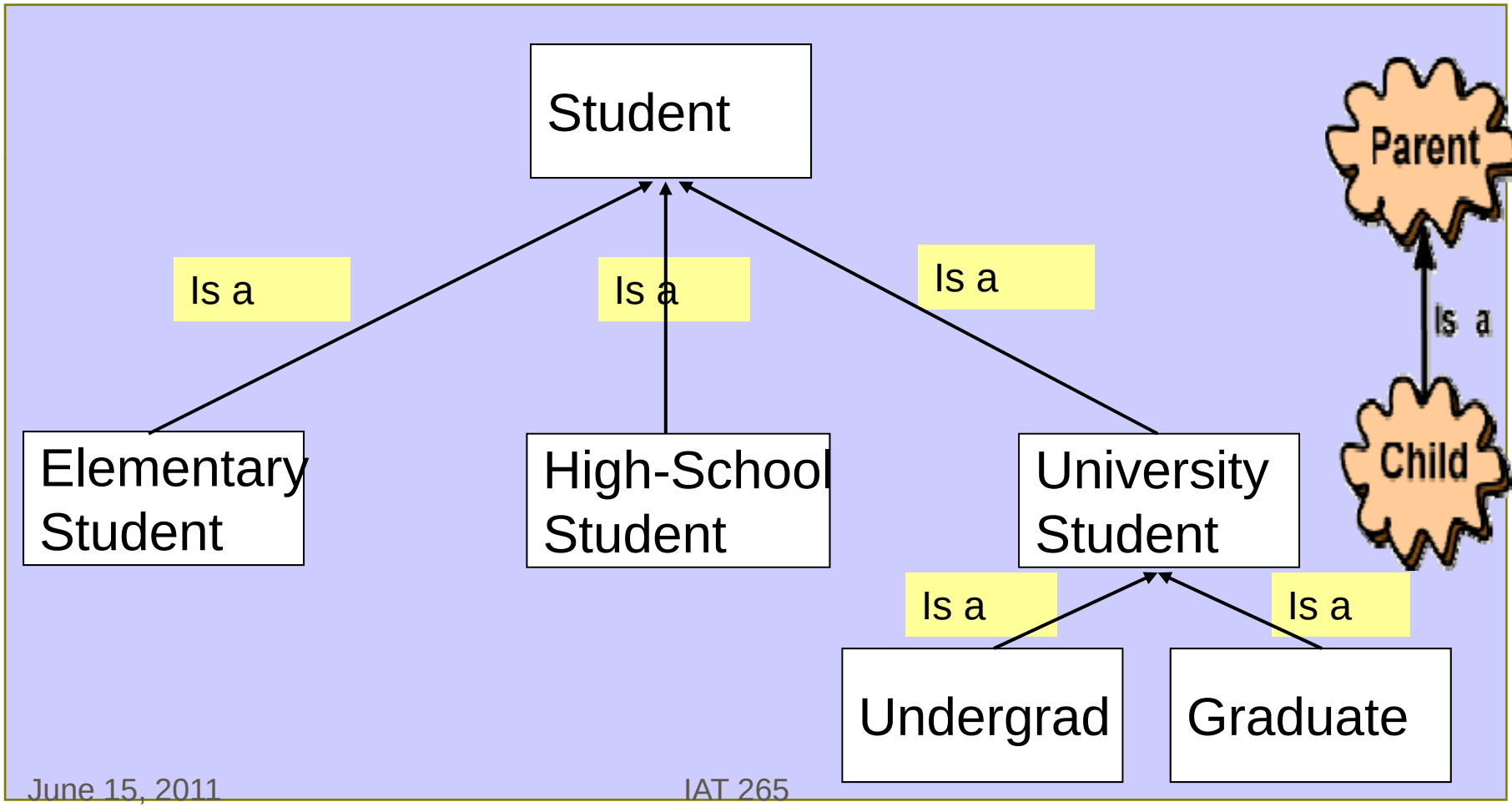
Three Principles of OOP

- Encapsulation
- **Inheritance**
- **Polymorphism**

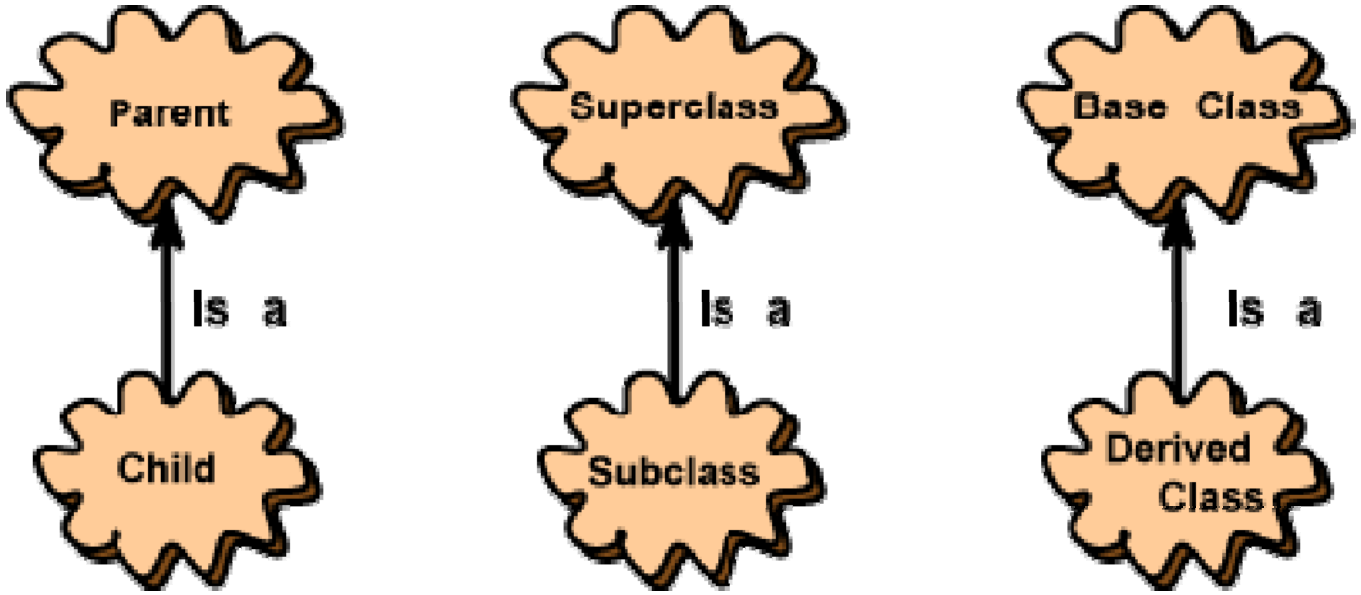
Encapsulation: good for 'has a' relationship



Inheritance: good for '*is a*' Relationship



Phrases for Inheritance



In Java, (unlike with humans) children inherit characteristics from *just one* parent. This is called **single inheritance**

Inheritance

- **Inheritance:** child class extends the functionality of a parent class while inheriting all of its attributes and behaviors

  - Subclass inherits all the fields and methods from its super class

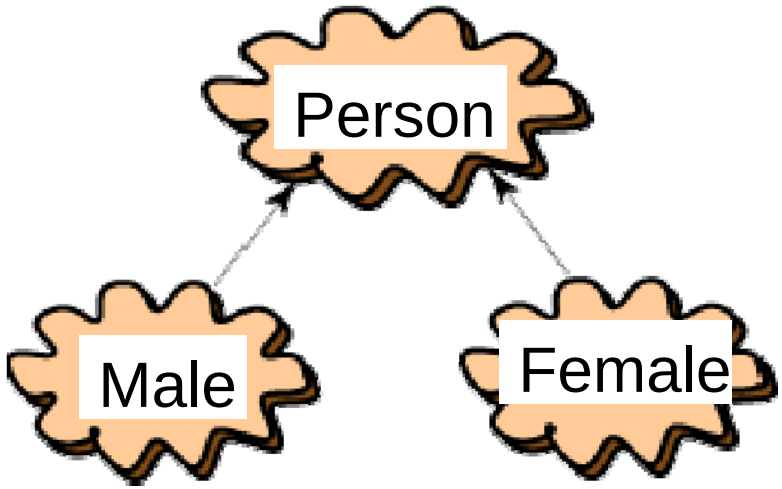
  - Subclass can define its own fields and methods

Why Inheritance?

- Mainly code reuse and extension:
  - allows classes to *inherit* commonly used attributes and behaviours from other classes, rather than reinvent the wheel
  - extends existing classes to add more functionality
    - Allows subclass code to focus exclusively on the features that is unique to itself

Example: Person vs. Male, Female

```
class Person {
    //Properties common to both Male/Female
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    // Behavior common to both Male / Female
    public void haveLunch(){
        println(this.name + " is having lunch");
    }
}
```



Example: Person vs. Male, Female

```
class Male extends Person {
        private String beardType;  //property unique to male
        public Male(String name, int age, String beardType) {
            super(name, age);
            this.beardType  = beardType;
        }
        public void showBeardType () {
            println(super.getName() + " is a male with beard of " + beardType);
        }
}
class Female extends Person {
        private String hairStyle;          //property unique to female
        public Female(String name, int age, String hairStyle) {
            super(name, age);
            this.hairStyle = hairStyle;
        }

        public void showHairStyle (){
            println(super.getName() + " is female with hair style of “ +
hairStyle);
        }
}
```

Super and *this*

- *this* is a keyword that always refers to the current object: Useful to refer to something of yourself within a class
  - this.field – refers to a field of yourself (useful to differentiate when you use the same names for fields and parameters)
  - this(parameters if any) – calls a constructor on yourself (useful for one version of a constructor to call another)

- *super* is a keyword that always refers to the superclass portion of an object
  - super.method() – calls the superclass' method (but normally you just directly call the method without super.)
  - super(parameters if any) – calls the superclass' constructor

Revisit our example

- So far we have Bugs that move around in a garden
- What if we want some bug to be able to eat other bugs smaller than them
  - But we don't want to get rid of our regular Bug, instead we want to reuse its fields and methods - whcih are common to all types of bugs
- Create a **subclass** that extends Bug!
  - EatingBug *is a* Bug → a perfect case for Inheritance

Inheritance: case study

- Subclasses inherit fields and methods from parent

```
class EatingBug extends Bug{
  …
}
```

Our subclass needs a constructor & something extra

- We want EatingBug objects to have a center dot with randomized color → need a field for that

```
color dotColor;
```

- We want the **EatingBug** constructor initialize **dotColor** as well as to do the same work as the Bug constructor

```
EatingBug(float x, float y, float chgX, float chgY,
float sz) {
  super(x, y, chgX, chgY, sz);
  dotColor = color(random(255), random(255),
    random(255));
}
```

super() here is to call the parent's constructor. Please note super() must be the 1st statement in children's constructors

Now we have EatingBug

- We can use EatingBug now in our example
- But, so far it's basically just a copy of Bug (except for a new field ***dotColor***)
- The only reason to define an EatingBug is to add new capabilities or to *override* old ones

Add an **eat()** method

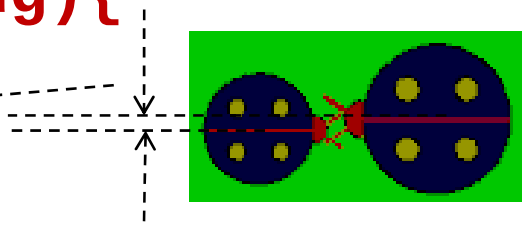
- We want an EatingBug object's **eat()** method to eat smaller bugs and grow itself

```
void eat(Bug otherBug) {
    if(bSize > otherBug.bSize) {
      //grow itself by 10%
      bSize *= 1.1;
      //kill the other bug
      otherBug.alive = false;
    }
}
```

Add a **checkHeadon()** method

- We want an EatingBug eats smaller bugs only when it hits them in head-on collisions → need a method to check on that:

```
boolean checkHeadOn(Bug otherBug){
  if(changeX*otherBug.changeX < 0
   && abs( bugY-otherBug.bugY) <
    max(bSize/8, otherBug.bSize/8)){
      return true;
 }
 return false;
}
```



- Now add code in loop to eat other bugs

Put EatingBug objects into action

```
EatingBug[] bugs = new EatingBug[count];

void setup() {
    ...      //anything the same as before

  for(int i=0; i<count; i++) {
    bugs[i] = new EatingBug (
     random(gardenW), random(gardenH),
     random(-1,1), random(-1,1),
     random(12,36));
  }
}
```

```
void draw() {
    …      //anything the same as before

   //if bug i and bug k collide and is head-on
    if(bugs[i].detectCollision(bugs[k]) &&
        bugs[i].checkHeadOn(bugs[k])) {

        bugs[i].eat(bugs[k]);
    }
   //if bug i and bug k collide but NOT head-on
    else if (bugs[i].detectCollision(bugs[k]) &&
        !bugs[i].checkHeadOn(bugs[k])) {

        bugs[i].bounce(bugs[k]);
    }

    …      //anything the same as before
}
```

Then what ...

- Can EatingBug has its own drawBug() method, so that it draws an EatingBug object differently (e.g. with a bigger dot at its center with randomized color)?
- If so, we know it inherits a drawBug() method from Bug, then which one gets called at runtime?

- Given Bug is the parent of EatingBug, can we use it as the type for declaring EatingBug objects?

These questions relate to OOP's third principle

- Encapsulation
- Inheritance
- **Polymorphism**
  - the ability to create a method or a reference variable in more than one form
  - Two types:
    - **Overriding polymorphism**
    - **Inclusion polymorphism**

Overriding polymorphism

- A subclass can redefine its parent's methods with the same signatures
- Can EatingBug has its own drawBug() method, so that it draws an EatingBug object differently?
  - Yes, and when drawBug() gets called at runtime, it is the one to be called (i.e. it *overrides* its parent version)

Override Bug's drawBug()

```
//Override parent's drawBug method
  void drawBug() {
    //call parent's drawBug() method to draw
    // a regular bug
    super.drawBug();

    //Draw a center bigger dot on top of
    //parent's version
    pushMatrix();
    translate(bugX, bugY);
    if(alive) {    //draw only if the bug is alive

      //make the bug rotate
      if( changeX <0 ) {
        rotateY(PI);
      }

      //draw the center dot with dotColor
      fill(dotColor);
      ellipse(0, 0, bSize/4, bSize/4);

      //redraw the body line
      stroke(160, 0, 0);
      line (-bSize/2, 0, -bSize/2, 0);
    }
    popMatrix();
}
```

Inclusion polymorphism

- You can use a superclass as the type to declare a name that reference to objects of all its subclasses
  - The name could be a *variable* or a *parameter*

Example of Inclusion polymorphism

- Remember that we defined two children classes (**Male** & **Female)** of class **Person**
- Remember that classes are types
  - So **Person** is a type, so are **Male** and **Female**
- So, here are some legal assignments
  - `Male p1 = new Male("Mark", 17, "moustache");`
  - `Person p2 = new Male("John", 20, "none");`
  - `Person p3 = new FeMale("Linda", 18, "longhair");`
- But this is illegal
  - `Male p4 = new Person("Ken", 22);`
- So it is perfectly legal to do this:
  - `Bug bug = new EatingBug (random(gardenW), random(gardenH), random(-1,1), random(-1,1), random(12,36));`

Same goes for parameters as well...

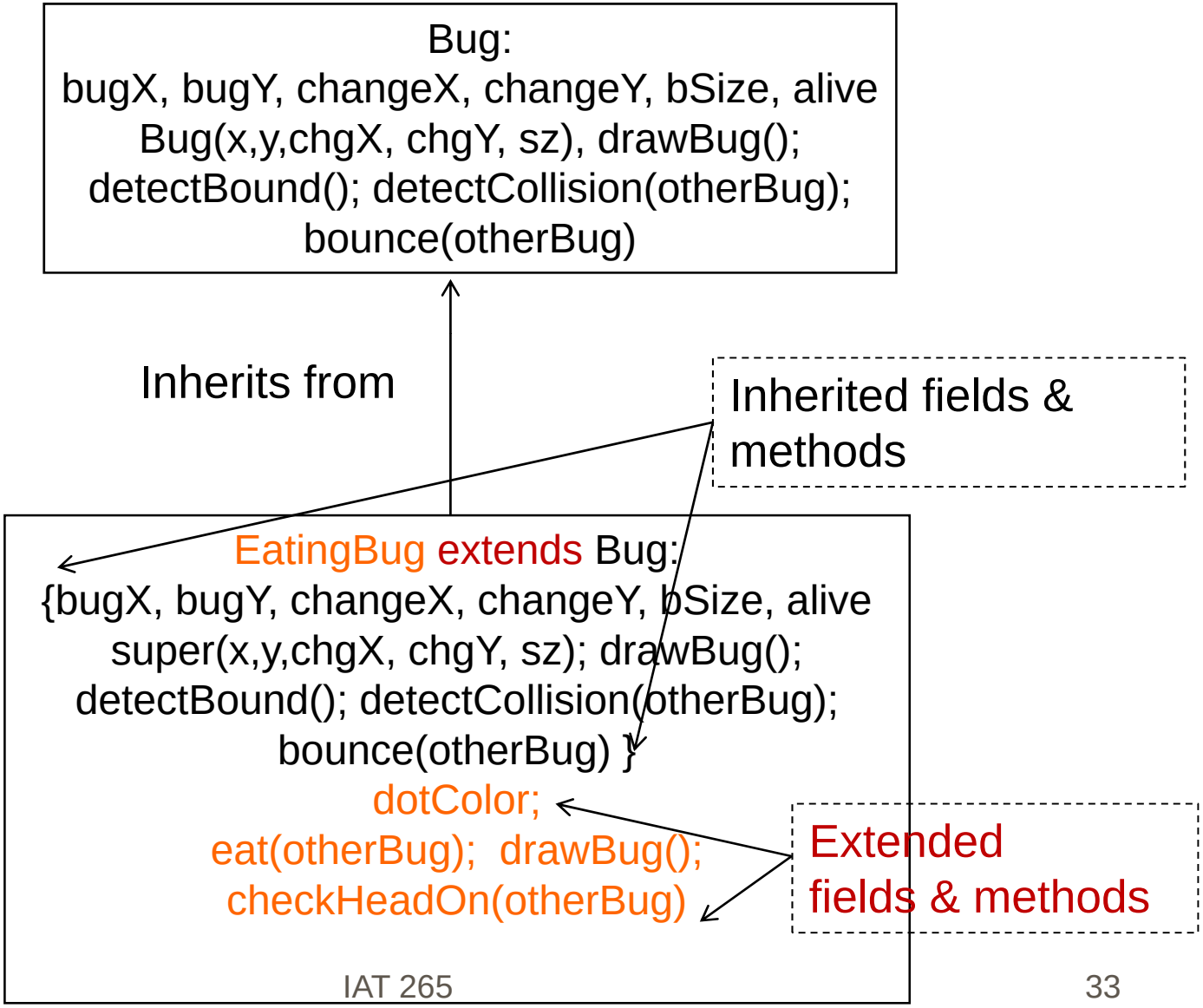
- A parameter of a superclass type can accept its subclass objects as arguments
  - This is useful when you have more than one subclass

```
void eat(Bug otherBug) {
  if(bSize > otherBug.bSize) {
    //grow itself by 10%
    bSize *= 1.1;
    //kill the other bug
    otherBug.alive = false;
  }
```

Here you can pass in objects of any Bug's subclasses as its argument,

e.g. an object of EatingBug

Bug Inheritance



String details

- A string is *almost* like an array of **char**s
  - char someletter = 'b';
  - String somewords = "Howdy-do, mr. jones?";
  - Note the use of double-quotes (vs. apostrophes)

- Like the objects we've created with classes, it has several methods, too...

String methods

- From http://processing.org/reference/String.html
  - *length()*
    - returns the size of the String (number of letters)
  - *charAt(number)*
    - returns the char at an index *number*
  - *toUpperCase()* and *toLowerCase()*
    - returns a copy of the String in UPPERCASE or lowercase respectively
  - *substring(beginIndex, endIndex)*
    - returns a portion of the String from *beginIndex* to *endIndex-1*

```
String howdy = "Hello!";        String expletive = howdy.substring(0,4);
```

String concatenation

- Concatenation means – appending a string to the end of another string
- With Strings, this is done using the **+** symbol
- So, if you have:

```
String s1 = "She is the";        String s2 = "programmer." ;

String sentence = s1 + " awesomest  " + s2;
```

- You'll get out:

```
println(sentence);  // sentence = "She is the awesomest programmer."

// outputs:  She is the awesomest programmer.
```

MORE String concatenation

- You can also add in numbers, too!

```
String anothersentence = s1 + "#"+ 2 + " " + s2;
// "She is the #2 programmer."
```

- There is also a function called nf() which can format your numbers (it stands for number format)

```
anothersentence = s1 +  nf(7,3) + " " + s2;
// nf( integer, number of digits )
// "She is the 007 programmer."

anothersentence = s1 + nf(3.14159,3,2) + " " + s2;
// nf( float, digits before decimal, digits after decimal )
// "She is the 003.14 programmer."
```

- It has siblings! nfs(); nfp(); nfc(); Consult the reference

Strings and Arrays

- Did you know that you can take an Array of Strings and join it into one String?

```
String[] a = { "One", "string", "to", "rule", "them", "all…" };

String tolkien = join(a, " ");

// join(stringArray, separator) ;
// tolkien == "One string to rule them all…"
```

- Did you also know that you can split a String into an Array?

```
String b = "Another string to bind them…" ;

String[] tolkien2= split(b, " ");
// tolkien2 == { "Another", "string", "to", "bind", "them…" }
```

Special characters

- Special characters
  - tab: "\t"
  - new line: "\n"

(\ - *escape character* , which tells the computer to look to the next character to figure out what to do)

String twolines = "I am on one line.**\n** I am **\t**on another."

I am on one line.
I am on another.

  - other *escape characters* include "\\" "\""

Summary

- Inheritance
  - Inheritance of 'is a' relationship
  - *super* and *this* keywords
  - Case study: EatingBug extends Bug
- Polymorphism
  - Overriding polymorphism
  - Inclusion polymorphism
- String and String functions