Advanced Java Programming

After mastering the basics of Java you will now learn more complex but important programming concepts as implemented in Java.

James Tam

Attributes Vs. Locals

- **Attributes**
  - Declared inside a class definition but outside the body of a method

```
public class Person {
    private String [] childrenName = new String[10];
    private int age;
}
```

- **Locals**
  - Declared inside the body of a method

```
public class Person {
    public nameFamily() {
        int i;
        Scanner in = new Scanner(System.in);
}
```

James Tam

Scope Of Attributes Vs. Locals

- **New term**: Scope is the location where an identifier (attribute, local, method) may be accessed
  - Scope of attributes (and methods): anywhere inside the class definition
  - Scope of locals: after the local has been declared until the end of closing brace (e.g., end of method body)
- Example:

```
public class Person {
    private String [] childrenName = new String[10];
    private int age;

    public nameFamily() {
        int i;
        for (i = 0; i < 10; i++) {
            childrenName[i] = in.nextLine();
        }
    }
}
```

Local (method scope)

Attribute (class scope)

James Tam

When To Use: Attributes

- Typically there is a separate attribute for each instance of a class and it lasts for the life of the object.

```
class Person
{
    private String [] childrenName = new String[10];
    private int age;
    /*
      For each person it's logical to track the age and
      the names any offspring.

    */
}
```

Q: Life of an object?

James Tam

When To Use: Locals

- Local variables: temporary information that will only be used inside a method

```
public nameFamily()
{
    int i;
    Scanner in = new Scanner(System.in);
    for (i = 0; i < 10; i++)
    {
        childrenName[i] = in.nextLine();
    }
}
```

Scope of 'i' (int)

Scope of 'in' (Scanner)

- Q: Does it make sense for every '`Person`' to have an '`i`' and '`in`' attribute?

James Tam

A Common Language-Based Convention

- Variables that are used as loop controls are sometimes declared as local only to the loop.
- Example:

```
for (int j = 1; j <= 4; j++)
{
   System.out.print(j + " ");  // In scope
}
// Error: Not in scope
// j = 0;
```

James Tam

Scoping Rules

- Rules of access
  1. Look for a local (variable or constant)
  2. Look for an attribute
- General example

```
public class Person
{

    public void method()
    {

        x = 12;
    }
}
```

Second: look for the definition of an attribute e.g., "private int x;"

First: look for the definition of a local identifier e.g., "int x;"

Reference to an identifier

James Tam

Scoping Rules: Example

```
public class C
{
    private int x;
    public void m()
    {
        int y;

        x = 1;
        y = 2;
    }
}
```

James Tam

Shadowing

- The name of a local matches the name of an attribute.
- Because of scoping rules the local identifier will 'hide' (shadow) access to the attribute.
- This is a common logic error!

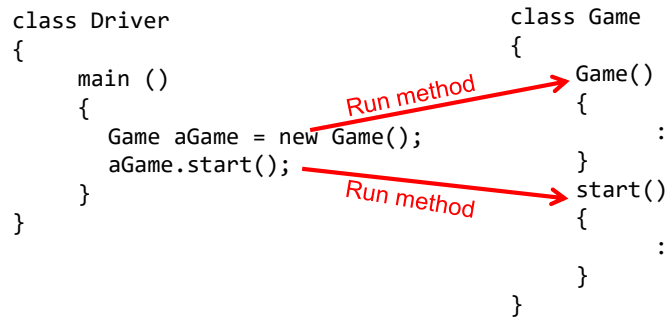
```
public class Person {
    private int age = -1;
    public Person(int newAge) {
        int age;  // Shadows/hides attribute
        age = newAge;
    }
    public void setAge(int age) { // Shadow/hide attribute
        age = age;
    }
}

Person aPerson = new Person(0);  // age is still -1
aPerson.setAge(18);              // age is still -1
```

James Tam

Messaging Passing

- Invoking the methods of another class.



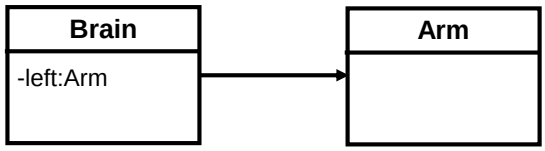
James Tam

Relationships Between Classes

- Association relation ("*has-a*") exists between classes if an instance of one class is an attribute of another class.
- Unidirectional association relation:
- **Example**:

```
class Brain                    class Arm
{                              {
     private Arm left;                ...
     ...
}                              }
```

- **UML**:



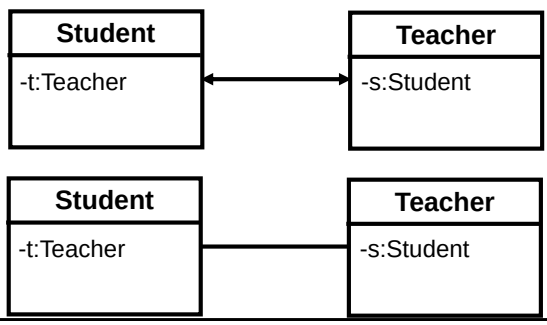
James Tam

Relationships Between Classes (2)

- Bidirectional association relation:
- **Example**:

```
class Student                  class Teacher
{                              {
     private Teacher t;             private Student s;
}                              }
```

- **UML**:



James Tam

Associations And Message Passing

- Having an association between classes allows messages to be sent from one object to another (objects of one class can call the methods of another class).

```
public class Car
{
    private Engine anEngine;
    private [] Lights carLights;
    ...
    public start()
    {
       anEngine.ignite();
       carLights[0].turnOn();
       ...
    }
}
```

```
public class Engine
{
    public boolean ignite () {
    .. }
}
public class Lights
{
    private boolean isOn;
    public void turnOn() {
        isOn = true;}
}
```

- Unidirectional: messages can be sent from car to engine or car to lights but not vice versa.

James Tam

Extra Exercise (Advanced)

- How do we ensure that:
  - A particular instance of one class refers to a particular instance of a second class?

  And

  - That instance of the second class refers to the previously referred to instance of the first class?
- Name of the example program:
  - /home/219/examples/advanced/1relationships
- What is wrong with the code?
- How can it be fixed?

James Tam

The Driver Class

```
public class Driver
{
    public static void main(String [] args)
    {
        Student s = new Student();
        System.out.println("<< DEBUG: This message will never
                            appear >>");
    }
}
```

James Tam

Class Student & Teacher

```
public class Student {
    private Teacher t;
    public Student() {
        t = new Teacher();
    }
}

public class Teacher {
    private Student s;
    public Teacher() {
        s = new Student();
    }
}
```

- JT's hint: similar to the "chicken and the egg" problem except in reverse!

James Tam

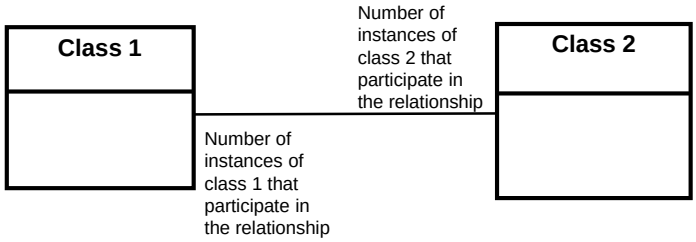
Multiplicity

- It indicates the number of instances that participate in a relationship

Multiplicity	Description
1	Exactly one instance
n	Exactly “n” instances {n: a positive integer}
n..m	Any number of instances in the inclusive range from “n” to “m” {n, m: positive integers}
*	Any number of instances possible

James Tam

Multiplicity In UML Class Diagrams



James Tam

Why Represent A Program In Diagrammatic Form (UML)?

- Images are better than text for showing structural relations.

Text

Jane is Jim's boss.

Jim is Joe's boss.

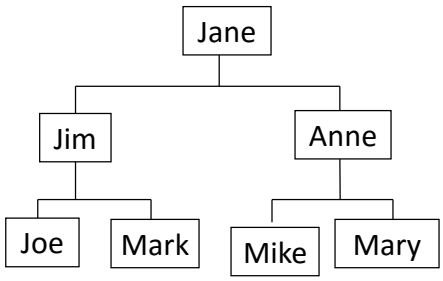
Anne works for Jane.

Mark works for Jim

Anne is Mary's boss.

Anne is Mike's boss.

Structure diagram



- UML can show relationships between classes at a glance

James Tam

Relationships Between Classes

- Design rule of thumb.
- It can be convenient to create a relationship between classes (allow methods to be invoked/messages to be passed).
- But unless it is necessary for a relationship to exist between classes do not create one.
- That's because each time a method can be invoked there is the potential that the object whose method is called can be put into an invalid state (similar to avoiding the use of global variables to reduce logic errors).

James Tam

Review: Previous Class

- What you have learned in your prerequisite class: some variables directly contain data:

```
num1 = 12
num2 = 3.5
ch = 'a'
```

- What you may have learned your prerequisite class: some variables 'refer' to other variables.

```
list = []
list = [1,2,3]
```

James Tam

Review: This Class

- In Java when you use objects and arrays there are two things involved:
  - Reference
  - Object (or array)
- Example with an object

```
Person charlie;   // Creates reference to object
charlie = new Person("Sheen");  // Creates object
```

- Example with an array

```
double [] salaries; // Creates reference to array
salaries = new double[100];  // Creates array
```

James Tam

Addresses And References

- Real life metaphor: to determine the location that you need to reach the 'address' must be stored (electronic, paper, human memory)

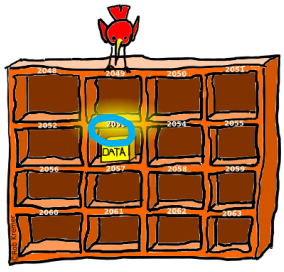
- Think of the delivery address as something that is a 'reference' to the location that you wish to reach.
  - Lose the reference (electronic, paper, memory) and you can't 'access' (go to) the desired location.

James Tam

Recap: Variables

- Variables are a 'slot' in memory that contains 'one piece' of information.

```
num = 123
```



- Normally a location is accessed via the name of the variable.
  - Note however that each location is also numbered!
  - This is the address of a memory location.

Image: Curtesy of Rob Kremer

James Tam

Addresses And References

- A reference to an array does not directly contain the contents of the array
  - Instead the reference contains the address ("refers to") of the array

James Tam

References And Objects

- Full example under:
  /home/219/examples/advanced/2referenceExamples

```
public class Person
{
    private String name;
    public Person() {  name = "none";  }

    public Person(String newName) {  setName(newName);
    }

    public String getName() {  return(name);  }

    public void setName(String newName) {
        name = newName;
    }
}
```

James Tam

References And Objects (2)

- In `main()`:

```
Person bart;
Person lisa;

bart = new Person("bart");
System.out.println("Bart object name: " + bart.getName());

lisa = bart;
bart = new Person("lisa");
System.out.println("Bart object name: " + bart.getName());
System.out.println("Lisa object name: " + lisa.getName());
```

```
Bart object name: bart
```

```
Bart object name: lisa
```

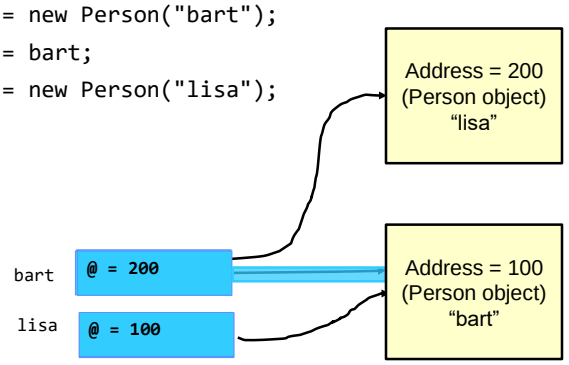
```
Lisa object name: bart
```

James Tam

References And Objects (3)

- What happened?

```
Person bart;
Person lisa;
bart = new Person("bart");
lisa = bart;
bart = new Person("lisa");
```



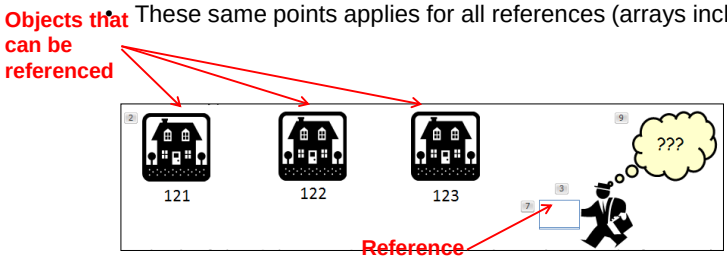
James Tam

References And Objects (4)

```
Person bart;
Person lisa;
bart = new Person("bart");
lisa = bart;
bart = new Person("lisa");
```

Note:

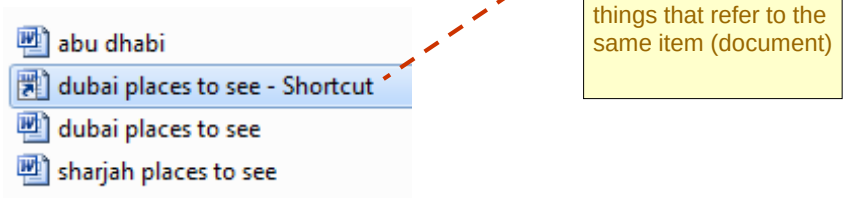
- The object and the reference to the object are separate e.g., 'bart' originally referenced the 'bart object' later it referenced the 'lisa object'
- The only way to access the object is through the reference.
- These same points applies for all references (arrays included)



James Tam

Shallow Copy Vs. Deep Copies

- Shallow copy (new term, concept should be review)



- **Copy the address from one reference into another reference**
- Both references point to the same location in memory

James Tam

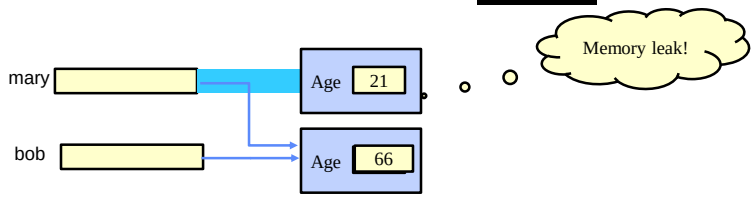
Shallow Copy Vs. Deep Copies (2)

- Shallow copy, full example under:
  /home/219/examples/advanced/3shallowDeep

```
Person mary = new Person(21);
Person bob = new Person(12);
System.out.println(mary.age + " " +
   bob.age);
mary = bob;   // Shallow;
bob.age = 66;
System.out.println(mary.age + " " +
   bob.age);
```

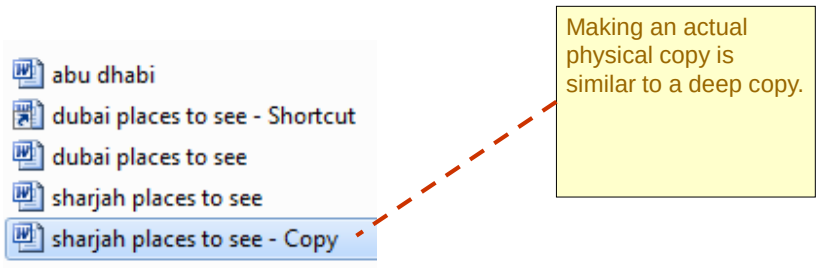
```
21 12
```

```
66 66
```



Shallow Copy Vs. Deep Copies (3)

- Deep copy (new term, concept should be review)



  - It's not the addresses stored in the references that's copied
  - **Instead the data referred to by the references are copied**
  - After the copy each reference still refers to a different address (the address refers to a data variable)

New Terms And Copying References

- **Dynamically allocated memory**:
  - Memory that is created/reserved/used only when explicitly requested.
  - In Java it's typically in conjunction with the 'new' keyword.
  - Examples:

```
int [] grades;
grades = new int[3];  // Dynamically allocated memory
```

- **Memory leak:**
  - Memory that has been dynamically allocated but it hasn't been deallocated or freed up when it's no longer needed
  - Shallow copies may often result in memory leaks

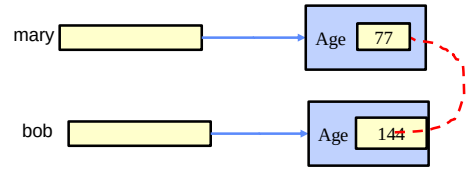
James Tam

Shallow Copy Vs. Deep Copies (4)

- Deep copy, full example under:
  /home/219/examples/advanced/3shallowDeep

```
// Mary still 66
bob = new Person(77);
mary.age = bob.age;     // Deep
bob.age = 144;
System.out.println(mary.age + " " +
   bob.age);
```

```
77 144
```

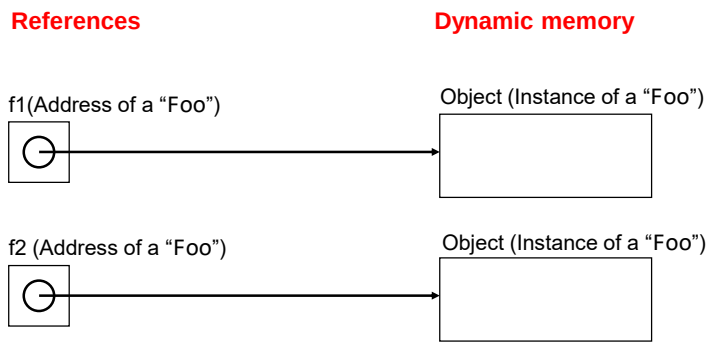


James Tam

Automatic Garbage Collection Of Java References

- Dynamically allocated memory is automatically freed up when it is no longer referenced (Foo = a class) e.g.,

```
Foo f1 = new Foo();
Foo f2 = new Foo();
```

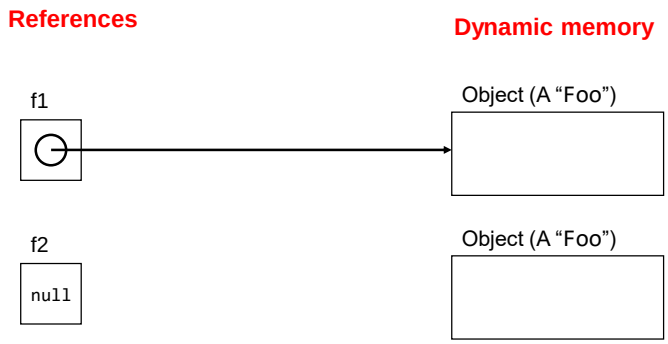


James Tam

Automatic Garbage Collection Of Java References (2)

- Dynamically allocated memory is automatically freed up when it is no longer referenced e.g.,

```
f2 = null;
```



James Tam

Automatic Garbage Collection Of Java References (3)

- Dynamically allocated memory is automatically freed up when it is no longer referenced e.g.,

  `f2 = null;`

  - Recall that a `null` reference means that the reference refers to nothing, it doesn't contain an address).

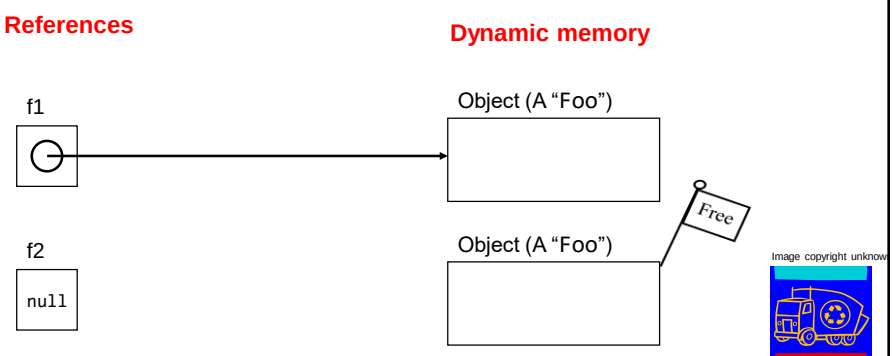


Image copyright unknown

James Tam

Caution: Not All Languages Provide Automatic Garbage Collection!

- Some languages do not provide automatic garbage collection (e.g., C, C++, Pascal).
- In this case dynamically allocated memory must be manually freed up by the programmer.
- Memory leak: memory that has been dynamically allocated (such as via the Java 'new' keyword') but has not been freed up after it's no longer needed.
  - Memory leaks are a sign of poor programming style and can result in significant slowdowns.

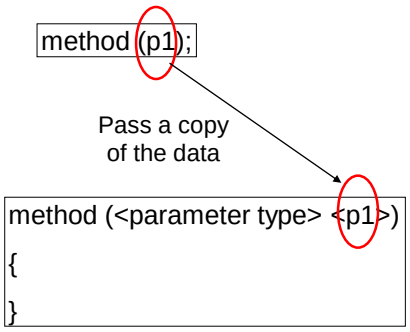
James Tam

Methods Of Parameter Passing

- Pass by value
  - The data stored (the "*value*" stored) in the parameter is copied
- Pass by reference
  - Pass the address of the parameter
  - This allows references to the parameter inside the method (the method has a "*reference*" to the original parameter).

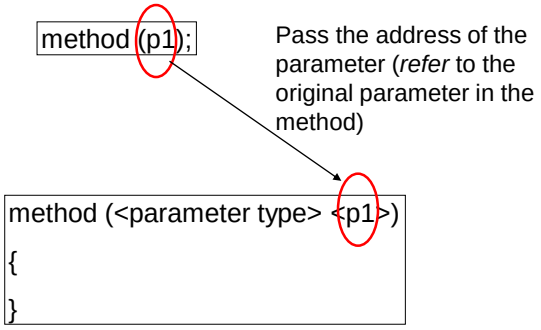
James Tam

Passing Parameters As Value Parameters



James Tam

Passing Parameters As Reference Parameters



James Tam

Which Parameter Passing Mechanism Is Used?

Passed by value

- All 'simple' built in types:
  - Integers (`byte`, `short`, `int`, `long`)
  - Floating point (`float`, `double`)
  - Character (`char`)
  - Boolean (`boolean`)

Pass by reference

- Objects
- Arrays
- (That is anything that consists of a reference and the item referenced).

James Tam

Parameter Passing Example

- Full example under:
  `/home/219/examples/advanced/4parameters`

James Tam

Class Person

```
public class Person {
    private int age;
    private String name;

    public Person() {
        age = -1;
        name = "none";
    }

    public int getAge() {
        return(age);
    }

    public String getName() {
        return(name);
    }
```

James Tam

Class Person (2)

```
    public void setAge(int anAge) {
        age = anAge;
    }

    public void setName(String aName) {
        name = aName;
    }
}
```

James Tam

Class ParameterExample

```
public class ParameterExample
{
    public void modify(Person aPerson, int aNum)
    {
        aPerson.setName("Eric Cartman");
        aPerson.setAge(10);
        aNum = 888;
        System.out.println("Person inside modify()");
        System.out.println(aPerson.getName() + " " +
                           aPerson.getAge());
        System.out.println("Number inside modify()");
        System.out.println(aNum);

    }
}
```

Modifies parameters here

James Tam

The Driver Class

```
public class Driver
{
    public static void main(String [] args)
    {
        int num = 13;
        Person aPerson = new Person();
        ParameterExample pe = new ParameterExample();

        System.out.println("Person in main() before edit");
        System.out.println(aPerson.getName() + " " +
                           aPerson.getAge());
        System.out.println("Number inside main() before edit");
        System.out.println(num);
        System.out.println("---
```

```
Person in main() before edit
none -1
Number inside main() before edit
13
```

The Driver Class (2)

```
        pe.modify(aPerson,num);
        System.out.println("----------");
```

```
public void modify(Person aPerson, int aNum)
{
  aPerson.setName("Eric Cartman");
  aPerson.setAge(10);
  aNum = 888;
```

```
Person inside modify()
Eric Cartman 10
Number inside modify()
888
```

```
        System.out.println("Person in main() after edit");
        System.out.println(aPerson.getName() + " " +
                           aPerson.getAge());
        System.out.println("Number inside main() after edit");
        System.out.println(num);

    }
}
```

```
Person in main() after edit
Eric Cartman 10
Number inside main() after edit
13
```

Previous Example: Analysis

- Why did the parameter that was passed by reference change and the simple type (passed by value) did not?

James Tam

Benefits Of Employing References

- References require a bit more complexity but provide several benefits over directly working with objects and arrays.
- Benefit 1: As you have just seen a reference contains the address of 'something' (object, array).
  - As long as the address of the object or array is retained changes made inside the method will persist after the method ends.
  - Recall that functions or methods can only return zero or one things (passing out of a function after it ends).
  - Passing by reference (passing into the function just as it starts executing) allows more than one change to persist after the function has ended:

```
fun(reference1,reference2,reference3…etc.)
```

James Tam

Benefits Of Employing References (2)

- Benefit 2: If an array or object is large then it's more memory efficient to pass a reference instead.
- Example:
  - References are typically 32 or 64 bits in size.
  - An array or object will almost always be larger.

```
char [] array1 = new char[1000000]; // 4 MB

class SocialNetworkUser
{
    // attribute for images
    // attribute for videos
}
```

James Tam

Modifying Simple Types (Parameters)

- What to do when only one thing needs to be changed: return the updated value after the method ends
- What to do when more than one thing needs to be changed:
  - Pass an array (e.g., three integers must be modified in a method, then pass an array of integers with 3 elements).
  - Enlist the aid of a wrapper (class).

Image copyright unknown

James Tam

Wrapper Classes

- A class definition built around a simple type

```
public class Coordinate {
    private int xCoordinate;
    private int yCoordinate;
    ...
}
```

- Benefits illustrated by this example:
  - Related pieces of information can be passed into methods together rather than separately.

```
Coordinate aLocation = new Coordinate();
Method(aLocation);   // vs method(x,y);
```

  - The values of two atomic types x & y can be changed inside a method call (because an object 'wraps' them and the object is passed by reference).

James Tam

Wrapper Classes (2)

- Also Wrapper classes are also used to provide class-like capabilities (i.e., methods) to simple types (e.g., `int`) e.g., class `Integer`
  - http://docs.oracle.com/javase/6/docs/api/java/lang/Integer.html
  - Example useful method `parseInt(String)`: converting strings to integers

```
int num = Integer.parseInt("123");  // More on this later
```

James Tam

Arrays: Parameters And Return Values

- Full example under:
  `/home/219/examples/advanced/5arrayParameters`

- **Format, method call**:
  -When the method is called, passing an array as a parameter and storing a return value appears no different as passing other types.
  -Example (`list1` and `list2` are arrays)

```
list2 = ape.oneDimensional(list1);
```

James Tam

Arrays: Parameters And Return Values (2)

- **Format, method definition**:
  - Use 'square brackets' to indicate that the return value or parameter is an array.
  - Each dimension requires an additional square bracket.
  - One dimensional:

```
public int [] oneDimensional(int [] array1) { ... }
```

  - Two dimensional:

```
public char [][] twoDimensional(char [][] array1) {
    ...
}
```

James Tam

Array Of 'Objects'

- Although referred to as an array of objects they are actually arrays of references to objects.
- Recall for arrays: 2 steps are involved to create the array

```
int [] array;          // Reference to array
array = new int[3];    // Creates array of integers
```

- Recall for objects: 2 steps are required to create the object

```
Person jim;          // Reference to Person object
jim = new Person();  // Creates object
```

James Tam

Array Of 'Objects' (2)

- An array of objects is actually an array of references to objects.
- So 3 steps are usually required
  - Two steps are still needed to create the array

```
// Step 1: create reference to array
Person [] somePeople;

// Step 2: create array
somePeople = new Person[3];
```

  - In Java after these two steps each array element will be null.

```
somePeople[0].setAge(10);  // Null pointer exception
```

James Tam

Array Of 'Objects' (3)

- The third step requires traversal through array elements (as needed): create a new object and have the array element refer to that object.

```
for (i = 0; i < 3; i++)
{
    // Create object, array element refers to that object
    somePeople[i] = new Person();

    // Now that array element refers to an object, a method
    // can be called.
    somePeople[i].setAge(i);
}
```

James Tam

Array Of Objects: Example

- Location of the full example:
  - /home/219/examples/advanced/6arrayReferences/simple

James Tam

Class Person

```
public class Person {
    private int age;

    public Person() {
        age = 0;
    }

    public int getAge() {
        return(age);
    }

    public void setAge(int anAge) {
        age = anAge;
    }
}
```

James Tam

Driver Class

```
public class Driver
{
    public static void main(String [] args) {
        Person [] somePeople; // Reference to array
        int i;
        somePeople = new Person[3];  // Create array
        for (i = 0; i < 3; i++) {
            // Create object, each element refers to a newly
            // created object
            somePeople[i] = new Person();
            somePeople[i].setAge(i);
            System.out.println("Age: " +
                               somePeople[i].getAge());
        }
    }
}
```

```
Age: 0
Age: 1
Age: 2
```

James Tam

Design Example

- Suppose we wanted to simulate a 2D universe in the form of a numbered grid ('World')

```
class World
{
    private [][] Tardis grid;
}
```

- Each cell in the grid was either an empty void or contained the object that traveled the grid ('Tardis')[1]

```
class Tardis
{

}
```

1 Tardis and "Doctor Who" © BBC

James Tam

General Description Of Program

- The 'world/universe' is largely empty.
- Only one cell contains the Tardis.
- The Tardis can randomly move from cell to cell in the grid.
- Each movement of Tardis uses up one unit of energy.

James Tam

Designing The World

Class World

- Attributes?
- Methods?

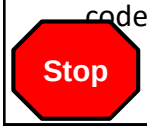
Class Tardis

- Attributes?
- Methods?

James Tam

CAUTION: STOP READING AHEAD

- JT's note: Normally you are supposed to read ahead so you are prepared for class.
- In this case you will get more out of the design exercise if you don't read ahead and see the answer beforehand.
- That will force you to actually think about the problem yourself (and hopefully get a better feel for some design issues).
- So for now skip reading the slides that follow this one up to the one that has a corresponding 'go' symbol all over it.
- After we have completed the design exercise in class you should go back and look through those slides (and the source code).



James Tam

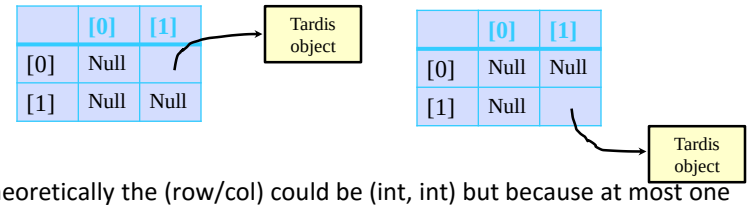
Tardis

- Attributes
  - Current energy level
- Methods:
  - Randomly generating movement:
    - Some method must reduce the energy level as the Tardis moves
    - The actual 'movement' from square to square in the grid will be a responsibility of class `World` because the grid is an attribute of the world.

James Tam

World

- Attributes
  - A 2D array that stores information about the 'universe'
  - Most array elements will be empty (`null`)
  - One element will refer to the Tardis object
  - The maximum number of rows and columns
  - The current location (row/column) of the Tardis
    - Needed to 'move' the Tardis from source cell to destination cell



  - Theoretically the (row/col) could be (int, int) but because at most one item can be returned from a method the location will be tracked as 1D integer array (details in code):
    - `World.move()->Tardis.calculateCoordinates()`

James Tam

World (2)

- Methods
  - Constructor(s) to create the world
  - Methods that modify the world (e.g., making sure each array element is truly null: `wipe()`
  - Displaying the world: `display()`
  - Changing the contents of the objects in the world (e.g., editing the world or moving objects): `move()`

James Tam

Manager

- It is responsible for things like determining how long the simulation runs.
- For very simple programs it may be a part of the `World` class (in this case it's part of the `Driver`).
- But more complex programs (e.g., need to track many pieces of information like multiple players, current scores etc. and simulation rules) may require a separate `Manager` class.
  - The `Driver` will then likely be responsible for instantiating a `Manager` object and calling some method of the manager to start the simulation.

James Tam

GO! GO!

END SECTION: Proceed Reading

- You can continue reading ahead to the slides that follow this one.
  - JT: Thank you for your understanding and co-operation.

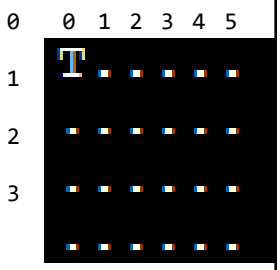
GO! GO!

Source Code: Design Exercise

- Location of the full source code:
  `/home/219/examples/advanced/6arrayReferences/doctor`

James Tam

Class Tardis




```
public class Tardis
{
    private int energy;
    public Tardis(int startEnergy) {
        energy = startEnergy;
    }
    // max row and column define the size of the world
    public int[] calculateCoordinates(int maxRow, int maxColumn) {
        Random aGenerator = new Random();
        int [] newCoordinates = new int[2];
        newCoordinates[0] = aGenerator.nextInt(maxRow);
        newCoordinates[1] = aGenerator.nextInt(maxColumn);
        energy--;
        return(newCoordinates);
    }
}
```

e.g., = 4 0, 1, 2, 3

e.g., = 7 0, 1, 2, 3, 4, 5, 6

James Tam

Class World: Attributes

```
public class World
{
    private Tardis [][] grid;  // Simulated world
    private int maxRow;     // Row capacity
    private int maxColumn;  // Column capacity
    private int [] currentLocation;  // (row/col) of Tardis
```

James Tam

Class `World`: Constructor

```
public World() {
    // Element 0: current row the tardis is located
    // Element 1: current column the tardis is located
    currentLocation = new int[2];

    Scanner in = new Scanner(System.in);
    System.out.print("Max rows: ");
    maxRow = in.nextInt();
    System.out.print("Max columns: ");
    maxColumn = in.nextInt();
    grid = new Tardis[maxRow][maxColumn];
    wipe();   // Empties the world, sets everything to null
    grid[0][0] = new Tardis(10); // Tardis starts top left
    currentLocation[0] = 0;  // Tardis row = 0
    currentLocation[1] = 0;  // Tardis col = 0
    display();
}
```

James Tam

Class `World`: Initialization

```
public void wipe()
{
    int r;
    int c;
    for (r = 0; r < maxRow; r++)
    {
        for (c = 0; c < maxColumn; c++)
        {
            grid[r][c] = null;
        }
    }
}
```

e.g., max = 2

e.g., max = 3

r = 0, c = {0,1,2}

r = 1, c = {0,1,2}

	[0]	[1]	[2]
[0]	null	null	null
[1]	null	null	null

James Tam

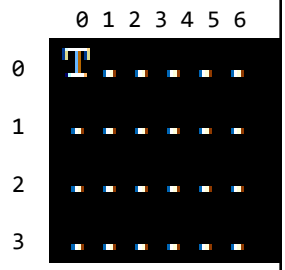
Class World: Display

```
public void display()
{
    int r;
    int c;
    for (r = 0; r < maxRow; r++)
    {
        for (c = 0; c < maxColumn; c++)
        {
            if (grid[r][c] == null)
                System.out.print(".");
            else
                System.out.print("T");
        }
        System.out.println();
    }
}
```

e.g., = 4

e.g., = 7

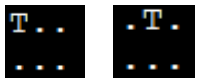
Move cursor to display new row on next line



James Tam

Movement

- To make it look like the Tardis has 'moved'.



- Set the destination (row/column) to refer to the Tardis object.
- Set the source (row/column) to null

Before move

	[0]	[1]
[0]	Null	
[1]	Null	Null

Tardis object

After move

	[0]	[1]
[0]	Null	Null
[1]	Null	

Tardis object

James Tam

Class World: Move

```
public void move()
{
     // currentLocation 1D array stores Tardis location
    int currentRow = currentLocation[0];
    int currentColumn = currentLocation[1];

    // Keep track of where the Tardis is currently located
    int oldRow = currentRow;
    int oldColumn = currentColumn;

    // Store new (row/col) in 1D array (currentLocation)
    currentLocation =
       grid[currentRow][currentColumn].calculateCoordinates
            (maxRow,maxColumn);
```

Recall:
`Tardis.currentCoordinates()`
randomly generates a new (row/column) location

James Tam

Class World: Move (2)

```
// Update temporary values with current location
currentRow = currentLocation[0];
currentColumn = currentLocation[1];

// Copy tardis from the old location to the new one.
grid[currentRow][currentColumn] = grid[oldRow][oldColumn];

// Check if tardis trying to move onto same square, don't
// 'wipe' if this is the case or tardis will be lost
// (Tardis object becomes a memory leak).
if ((currentRow == oldRow) &&
    (currentColumn == oldColumn)) {
      System.out.println("Same location");
}
else {
      // 'wipe' tardis off old location
      grid[oldRow][oldColumn] = null;
}
```

James Tam

Class World: Move (3)

```
        System.out.println("Tardis re-materializing");
        display();
    }
```

James Tam

The Driver Class (Also The "Manager")

```
public class Driver
{
    public static void main(String [] args) {
        Scanner in = new Scanner(System.in);
        World aWorld = new World();
        int i;
        for (i = 0; i < 10; i++) {
            aWorld.move();
            System.out.println("Hit enter to continue");
            in.nextLine();
        }
        System.out.println("\n<<<Tardis is out of energy,
           end simulation>>> \n");
    }
}
```

James Tam

Introducing A New Concept With..Class Sheep!

```
public class Sheep
{
    private String name;

    public Sheep()
    {
        name = "No name";
    }
    public Sheep(String aName)
    {
        setName(aName);
    }
    public String getName() { return name;}

    public void setName(String newName) { name = newName; }
}
```

James Tam

We Create Several Sheep

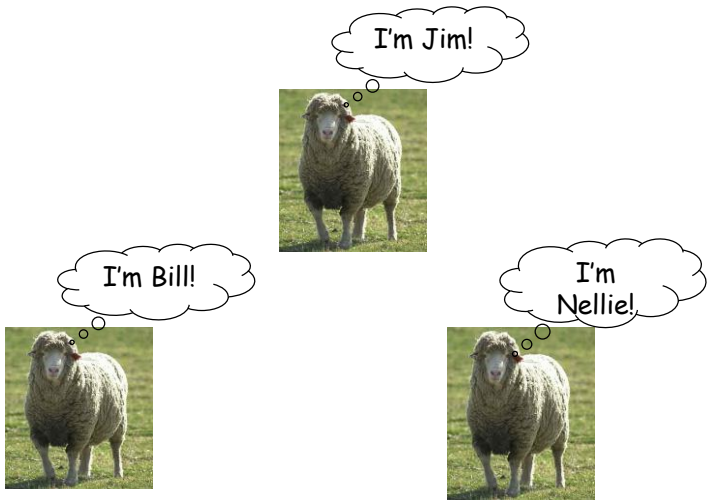


Image copyright unknown

James Tam

Question: Who Tracks The Size Of The Flock?

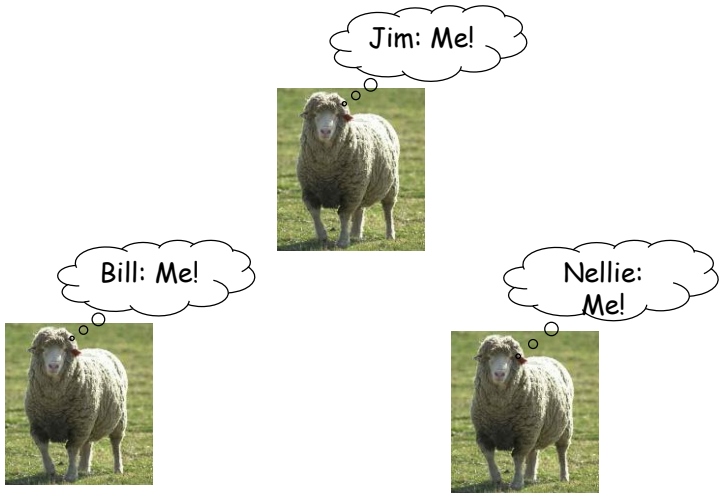


Image copyright unknown

James Tam

Answer: None Of The Above!

- Information about all instances of a class should not be tracked by an individual object.
- So far we have used instance fields.
- Each *instance* of an object contains *it's own set of instance fields* which can contain information unique to the instance.

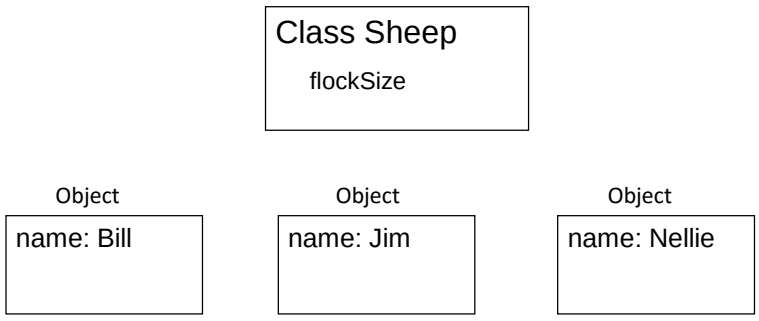
```
public class Sheep
{
     private String name;
       ...
}
```

Object	Object	Object
name: Bill	name: Jim	name: Nellie

James Tam

The Need For Static (Class Attributes)

- Static fields: One instance of the attribute exists *for the class* (not one attribute for each instance of the class)
- JT's note: in Java static DOES NOT specify unchanging (constant)
  - Reminder: the keyword 'final' signifies constant (unchanging)



James Tam

`Static` (Class) Methods

- Are associated with the class as a whole and not individual instances of the class.
  - Can be called without having an instances (because it's called through the class name not a reference/instance name).
  - Instance method:

```
Scanner in = new Scanner(System.in);
in.nextInt();   // referenceName.method()
```

  - Class Method:

```
double squareRoot = Math.sqrt(9);  // ClassName.method()
```

- Typically implemented for classes that are never instantiated e.g., class `Math`.

James Tam

Accessing Static Methods/Attributes

- Inside the class definition

Format:

```
<Access permission> static <attribute or method name>
```

Example:

```
class Sheep
{
    private static int flockSize = 0;

    public Sheep()
    {
        flockSize++;
    }
}
```

James Tam

Accessing Static Methods/Attributes (2)

- Outside the class definition

Format:

```
<Class name>.<attribute or method name>
```

Example:

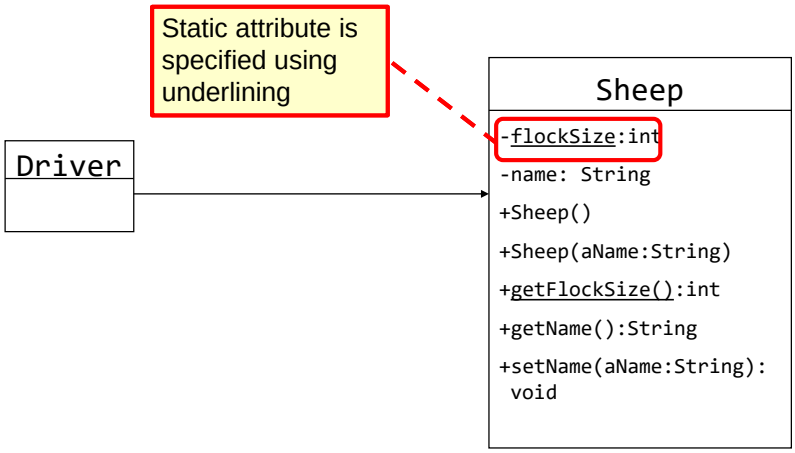
```
Sheep.getFlockSize();
```

James Tam

Static Data And Methods: UML Diagram

- Location of the online example:

  -/home/219/examples/advanced/6classAttributes



James Tam

Static Data And Methods: The Driver Class

```
public class Driver
{
    public static void main(String [] args) {
        System.out.println();
        System.out.println("You start out with " +
                           Sheep.getFlockSize() +
                          " sheep");
        System.out.println("Creating flock...");
        Sheep nellie = new Sheep("Nellie");
        Sheep bill = new Sheep("Bill");
        Sheep jim = new Sheep();
        System.out.println("Current count " +
                           Sheep.getFlockSize());
    }
}
```

James Tam

Static Data And Methods: The Sheep Class

```
public class Sheep
{
    private static int flockSize = 0;
    private String name;

    public Sheep() {
        flockSize++;
        name = "No name";
    }
    public Sheep(String aName) {
        flockSize++;
        setName(aName);
    }

    public static int getFlockSize () { return flockSize; }
    public String getName() { return name;}
    public void setName(String newName) { name = newName; }
}
```

James Tam

Rules Of Thumb: Instance Vs. Class Fields

- Reminder:
  - Instance field:
    - Static keyword is not used
    - There is one instance for each object created
    - E.g., `class Person { private int age; }`
  - Class field:
    - Requires the static keyword
    - There is one instance for the entire class
    - E.g., `class Person { private static int numberPeople; }`
- Rules of thumb:
  - Make it an instance field if the data can vary between instances e.g., age, height, weight
  - Make it a class field if the data relates to all instances e.g., number of objects created.
    - Possibly it may apply if no instances will be created e.g., a debug flag to specify the mode that the program is operating under

James Tam

Rule Of Thumb: Instance Vs. Class Methods

- Reminder:
  - Instance method e.g.,

```
class Person { private int age = 0;
    public void haveBirthDay() { age++; }
}
```

  - Class method e.g.,

```
class Math {
    public static double square(double num) {return(num*num);
} }
```

James Tam

Rule Of Thumb: Instance Vs. Class Methods (2)

- Rule of thumb
  - Static methods
    - If a method can be invoked regardless of the number of instances that exist (e.g.., the method can be run when there are no instances) then it probably should be a static method.
    - If it never makes sense to instantiate an instance of a class then the method should probably be a static method.

      E.g., the class doesn't have any variable attributes only static constants such as class `Math` no objects are instantiated (more coverage later)
  - Non static methods
    - If the above rules don't apply then the method should likely be an instance method e.g., the method operates on an instance field.

James Tam

Universally Accessible Constants

- What you currently know
  - –How to declare constants that are local to a method

```
class Driver {
    main() {
        final int A_CONST = 10;
    }
}
```

- If you need constants that are accessible throughout your program then declare them as **class constants**.

James Tam

Declaring Class Constants

- **Format:**

```
public class <class name>
{
    public final static <type> <NAME> = <value>;
}
```

- **Example:**

```
public class Person
{
    public final static int MAX_AGE = 144;
}
```

- Notes:
  - The keyword “final” signifies something that cannot change (a constant)
  - Because MAX_AGE is a constant the access level can be public.

James Tam

Accessing Class Constants

- **Format** (outside of the class definition)[1]:
  <Class name>.<constant name>;

- **Example** (outside of the class definition):

```
main()
{
    System.out.println("Max life span: " + Person.MAX_AGE);
}
```

- Accessing a class constant inside the class where it's been defined does not require the name of the class

```
public class Person {
    public final static int MAX_AGE = 144;
    public void sayMax() { System.out.println(MAX_AGE); }
}
```

James Tam

Static Vs. Final

- **Static**: Means there's one instance of the attribute for the class (not individual instances for each instance (object) of the class)
- **Final**: Means that the attribute cannot change (it is a constant)

```
public class Foo
{
    public static final int num1= 1;
    private static int num2;  /*  Rare */
    public final int num3 = 1;  /* Why bother (waste) */
    private int num4;
     :           :
}
```

James Tam

An Example Class With A Static Implementation

```
public class Math
{
    // Public constants
    public static final double E = 2.71…
    public static final double PI = 3.14…

    // Public methods
    public static int abs(int a);
    public static long abs(long a);
            :        :
}
```

- For more information about this class go to:
  - http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

James Tam

Should A Class Be Entirely Static?

- Usually purely static classes (cannot be instantiated) have only methods and no data (maybe some constants).
  - Rare: mostly cases there's variable data that is different from object-to-object so few classes are purely static
- Example (purely for illustration):

```
Math math1 = new Math();
Math math2 = new Math();
// What's the difference? Why bother?
math1.abs() vs. math2.abs();
```

- When in doubt *DO NOT* make attributes and methods static.

James Tam

Self Reference: The 'This' Reference

- From every (non-static) method of an object there exists a reference to the object (called the "this" reference) [1]

```
main(String args []) {
  int x;
  Person fred = new Person();
  Person barney = new Person();
  fred.setAge(35);
}

public class Person {
    private int age;
    public void setAge(int anAge) {
         age = anAge;
   }
            ...
 }
```

This is one reason why methods must be invoked via a reference name (the contents of the reference 'fred' will be copied into the 'this' reference (so both point to the 'Fred' object).

The 'this' reference is implicitly passed as a parameter to all non-static methods. One use of 'this' is to distinguish which object's method is being invoked (in this case Fred vs. Barney)

1 Similar to the 'self' keyword of Python except that 'this' is a syntactically enforced name.

James Tam

The 'This' Reference Is Automatically Referenced Inside (Non-Static) Methods

```
public class Person {
   private int age;
    public void setAge(int anAge) {
        // These two statements are equivalent
        age = anAge;
        this.age = anAge;
   }
}
```

James Tam

Parameter Types: Explicit Vs. Implicit

- Explicit parameter(s): explicitly passed (you can see them when the method is called and defined).

```
fred.setAge(10);    // 10 explicit
barney.setAge(num);   // num explicit

public void setAge(int age) { ... }    // age explicit
```

- Implicit parameter: implicitly passed into a method (automatically passed and cannot be explicitly passed): the 'this' reference.

```
public void setAge(int age) { ... }    // 'this' is implicit
```

James Tam

Benefits Of 'This': Attributes

- Another side benefit is the this reference can make it very clear which attributes are being accessed/modified.

```
public class Person
{
    private int age;

    public void setAge(int age) {
        this.age = age;
    }
}
```

Parameter (local variable) 'age'

Attribute 'age'

James Tam

Benefits Of 'This': Parameters

- Another side benefit is the `this` reference can make it clear which object is being accessed e.g., when a class method takes as a explicit parameter an instance of that class[1]

```
main (String [] args) {
    Person fred = new Person("Fred");
    Person barney = new Person("Barney");
    barney.nameBestBuddy(fred);    // JT: Explicit? Implicit?
}
// JT: What will be the output?
public void nameBestBuddy(Person aPerson) {
    println(this.name + " best friend is " + aPerson.name);
}
```

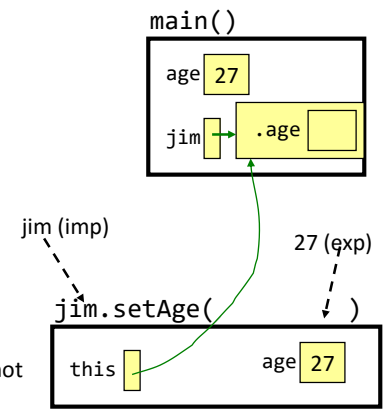
1 JT: more on this one later – see the 'equals()' method

James Tam

Benefits Of 'This': Scope

- Recall: according to scoping rules, local variables are not accessible outside of that function or method (unless returned back to the caller or passed into another method).

```
main (String [] args) {
    int age = 27;
    Person jim = new Person();
    jim.setAge(age);
}
class Person {
    public void setAge(int age) {
        this.age = age;
}
```



Normally the object referred to by the 'jim' reference not accessible outside of main() but the 'this' reference contains it's address (implicit pass by reference)

James Tam

Static Methods: No 'This' Reference

- Recall: **static methods** do not require an object to be instantiated because they are invoked via the **class name** not a reference name.

```
int result = Math.abs(-12);
```

- That means static methods do not have the implicit 'this' parameter passed in.
- Also recall I said for now avoid [for the 'Driver' class]:
  - Defining attributes for the Driver
  - Defining methods for the Driver (other than the main method)

James Tam

Driver.main(): Problem, No This

```
public class Driver {
    private int num;
    public static void main(String [] args) {
        num = 12; // Problem!
        // Recall
        num = 12;

        // Equivalent to
        this.num = 12;


        // But there is no this reference for static methods
        Driver d = new Driver();
        d.main("");  // Main was no called via a reference
    }
}
```

James Tam

This()

- Can be used when constructors have been overloaded.
- Calls one version of the constructor from another constructor.
- Example program:
  /home/219/examples/advanced/8thisMethod

James Tam

The Driver Class

```
public class Driver
{
    public static void main(String [] args)
    {
        Person aPerson = new Person();
        aPerson.show();

        aPerson = new Person(99);
        aPerson.show();

        aPerson = new Person("Bob");
        aPerson.show();
    }
}
```

James Tam

Class Person

```
public class Person {
    private int age;
    private String name;

    public Person() {
        age = -1;
        name = "none";
    }

    public Person(int anAge) {
        this();
        age = anAge;
    }
```


Class Person (2)

```
    public Person(String aName) {
        this();
        name = aName;
    }

    public void show()
    {
        System.out.println(age + " " + name);
    }
}
```

Displaying The Current State Of Objects

- The `toString()` method desplays the state of a particular object (contents of important attributes).
  - Returns a string representation of the state.
- It will automatically be called whenever a reference to an object is passed as a parameter to "`print()/println()`".

James Tam

`toString()` Example

- Location of the full example:
  - `/home/219/examples/advanced/9toString`

James Tam

Class Person

```
public class Person
{
    private int height;
    private int weight;
    private String name;

    public Person(String name, int height, int weight)
    {
        this.name = name;
        this.height = height;
        this.weight = weight;
    }
```

James Tam

Class Person (2)

```
    public String getName()
    {
        return(name);
    }

    public int getHeight()
    {
        return(height);
    }

    public int getWeight()
    {
        return(weight);
    }
```

James Tam

Class Person (3)

```
    public String toString()
    {
        String s;
        s = "Name: " + name + "\t";
        s = s + "Height: " + height + "\t";
        s = s + "Weight: " + weight + "\t";
        return(s);
    }
}
```

James Tam

The Driver Class

```
public class Driver
{
    public static void main(String [] args)
    {
        Person jim = new Person("Jim",69,160);
        System.out.println("Attributes via accessors()");
        System.out.println("\t" + jim.getName() + " " +
                           jim.getHeight() +
                          " " + jim.getWeight());
```

```
Attributes via accessors()
        Jim 69 160
```

```
        System.out.println("Attributes via toString()");
        System.out.println(jim);
    }
}
```

```
Attributes via toString()
Name: Jim       Height: 69      Weight: 160
```

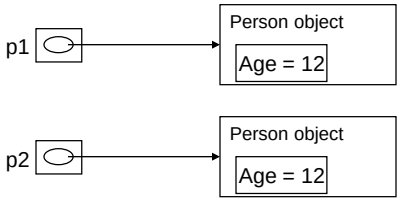
James Tam

Comparing Objects

- Recall from the discussion of parameter passing (pass by reference) that a reference contains the address of an object or array.
- Using the comparison operator on the references '==' will only determine if the address (and not data) is the same.

```
Person p1 = new Person(12);
Person p2 = new Person(12);


if (p1 == p2)
```



James Tam

Comparing Objects (2)

- Either each attribute of each object must be manually compared or else some form of `equals()` method must be implemented.
- Class String has two methods:
  - `compareTo()` `# ABC not same as Abc`
  - `compareToIgnoreCase()` `# ABC same as abc`

James Tam

Implementing Equals()

- Location of the full example:
  - /home/219/examples/advanced/10equals

Class Person

```
public class Person {
    private int height;
    private int weight;

    public Person(int height, int weight) {
        this.height = height;
        this.weight = weight;
    }

    public int getHeight() {
        return(height);
    }

    public int getWeight() {
        return(weight);
    }
```

Class Person (2)

```
    public void setHeight(int height) {
        this.height = height;
    }


    public void setWeight(int weight) {
        this.weight = weight;
    }
```

Implicit: **Jim** Explicit: **Bob**

```
    public boolean equals(Person compareTo) {
        boolean flag = true;
        // Access to compareTo privates allowed here!
        if (this.height != compareTo.height ||
            this.weight != compareTo.weight)
            flag = false;
        return(flag);
    }
}
```

James Tam

The Driver Class

```
public class Driver
{
    public static void main(String [] args)
    {
        Person jim = new Person(69,160);
        Person bob = new Person(72,175);
```

James Tam

The Driver Class (2)

```
new
Person(69,160);
new
Person(72,175);
```

```
System.out.println("Different data, addresses");
System.out.println("Compare data via accessors()");
if (jim.getHeight() == bob.getHeight() &&
    jim.getWeight() == bob.getWeight())
    System.out.println("\tObjects same data");
else
    System.out.println("\tNot equal");

System.out.println("Compare data via equals()");
if (jim.equals(bob) == true)
    System.out.println("\tObjects same data");
else
    System.out.println("\tNot equal");

System.out.println("Compare addresses");
if (jim == bob)
    System.out.println("\tSame address");
else
    System.out.println("\tDifferent addresses");
```

```
Compare data via accessors()
        Not equal
Compare data via equals()
        Not equal
Compare addresses
        Different addresses
```

James Tam

The Driver Class (3)

```
Person(72,175); # via set()
Person(72,175);
```

```
System.out.println();
System.out.println("Same data, different addresses");
jim.setHeight(72);
jim.setWeight(175);
if (jim.equals(bob) == true)
    System.out.println("\tObjects same data");
else
    System.out.println("\tNot equal");


System.out.println("Compare addresses");
if (jim == bob)
    System.out.println("\tSame address");
else
    System.out.println("\tDifferent addresses");
```

```
Same data, different addresses
        Objects same data
Compare addresses
        Different addresses
```

James Tam

Person(72,175); # via set()

Person(72,175);

The Driver Class (4)

```
System.out.println();
System.out.println("Same data, different addresses");
jim.setHeight(72);
jim.setWeight(175);
if (jim.equals(bob) == true)
    System.out.println("\tObjects same data");
else
    System.out.println("\tNot equal");


System.out.println("Compare addresses");
if (jim == bob)
    System.out.println("\tSame address");
else
    System.out.println("\tDifferent addresses");
```

```
Same data, different addresses
        Objects same data
```

```
Compare addresses
        Different addresses
```

James Tam

jim = bob;

The Driver Class (5)

```
System.out.println();
System.out.println("Same addresses");
jim = bob;
if (jim == bob)
    System.out.println("\tSame address");
else
    System.out.println("\tDifferent addresses");
```

```
Same  addresses
        Same address
```

James Tam

Mutable Vs. Immutable Types

- Mutable types
  - Original memory can be modified

```
int num = 666;
num = 777;
```

- Immutable types
  - The original memory location cannot be modified
  - Assigning new values will create a new memory location and leave the original untouched.

```
String s1 = “abc”;
String s2 = s1;
s1 = “xyz”;
System.out.println(s1 + “ “ + s2);
```

James Tam

Mutable Vs. Immutable

- Advantage of mutable types: speed
- Advantage of immutable types: ‘security’

James Tam

Mutable Advantage: Speed

- Location of full examples:
  - /home/219/examples/advanced/11mutableImmutable/speed

```
public class StringExample {
   public static void main
     (String [] args) {
        String s = "0";
        int i;
        for (i = 1; i < 100000; i++)
            s = s + i;
    }
}
```

```
public class StringBufferExample {
  public static void main
    (String [] args) {
        StringBuffer s;
        int i;
        s = new StringBuffer("0");
        for (i = 1; i < 100000; i++)
            s = s.append(i);
     }
}
```

James Tam

Immutable Advantage: Security

- Location of the full example:
  - /home/219/examples/advanced/11mutableImmutable/security

James Tam

Class SecurityExample

```
public class SecurityExample
{
    private String s;
    private StringBuffer sb;

    public SecurityExample() {
        s = new String("Original s");
        sb = new StringBuffer("Original sb");
    }

    public String getS() {
        return s;
    }

    public StringBuffer getSB() {
        return sb;
    }
}
```

James Tam

The Driver Class

```
public class Driver
{
    public static void main(String [] args)
    {
        SecurityExample se = new SecurityExample();
        String s;
        StringBuffer sb;

        System.out.println("Originals");
        System.out.println("\t" + se.getS());
        System.out.println("\t" + se.getSB());

        s = se.getS();
        sb = se.getSB();
```

```
Originals
        Original s
        Original sb
```

James Tam

The Driver Class (2)

```
        sb.delete(0,sb.length());
        sb.append("lolz! mucked ur data :P");
        s = "lolz! mucked ur data :P";
        System.out.println();

        System.out.println("Afte
        System.out.println("Values of locals");
        System.out.println("\t\tString=" + s);
        System.out.println("\t\tStringBuffer=" + sb);


        System.out.println("\tValues of attributes");
        System.out.println("\t\tString=" + se.getS());
        System.out.println("\t\tStringBuffer=" + se.getSB());
    }
}
```

```
Values of locals
        String=lolz! mucked ur data :P
        StringBuffer=lolz! mucked ur data :P
```

```
Values of attributes
        String=Original s
        StringBuffer=lolz! mucked ur data :P
```

James Tam

New Terminology/Definitions

- Scope
- Shadowing
- Message passing
- Association relation (bidirectional, unidirectional)
- Shallow and deep copy
- Dynamically allocated memory
- Memory leak
- Automatic garbage collection
- Memory leak
- Parameter passing: Pass by value, pass by reference
- Static attributes and methods

James Tam

New Terminology/Definitions (2)

- Final attributes
- Mutable
- Immutable

James Tam

After This Section You Should Now Know

- What is meant by scope
- Scoping rules for attributes, methods and locals
  - Design issues
    - When should something be declared as local vs. an attribute
- The hierarchy of scoping rules
  - How locals can shadow attributes
- What is meant by message passing
- What is an association, how do directed and non-directed associations differ, how to represent associations and multiplicity in UML
- What is multiplicity and what are kinds of multiplicity relationships exist
- Design and technical issues related to association relations

James Tam

After This Section You Should Now Know (2)

- References
  - How references and objects are related
  - The difference between a deep vs. shallow copy
  - What is the difference between comparing references vs. objects
  - What is automatic garbage collection and how it's related to the use of references
- How the two methods of parameter passing work, what types are passed using each mechanism
- What are the benefits of employing references
- What is a wrapper class and the value provided

James Tam

After This Section You Should Now Know (3)

- How to pass arrays as parameters and return them from methods
- Arrays of 'objects'
  - Why they are really arrays of references
  - How to declare such an array, create and access elements
- How to create a simple simulation using an array of references
- Static attributes and methods
  - How to create statics
  - How to access statics
  - When something should be static vs. non-static (instance)
  - How to represent static in UML
- How to declare class constants
  - The difference between static and final

James Tam

After This Section You Should Now Know (4)

- What is the '`this`' reference
  - When it is and is not an implicit parameter
  - What's the difference between implicit and explicit parameters
  - What are the benefits of having a `this` parameter
- How to use `this()` to call overloaded constructors within another constructor
- How to display the state of an object by implementing a `toString()` method
- How to check for equality by implementing an `equals()` method

James Tam

Copyright Notification

- "Unless otherwise indicated, all images in this presentation are used with permission from Microsoft."

James Tam