Classes and Methods Part III

class relationships
this
static
overloading constructors
aliases and pass by value
enum

CLASS DESIGN: A REVIEW

Class Design

- A class is a *concept* of an object
- Once we define a class, we can instantiate as many objects of that class as we need

- Characteristics of an object are represented by instance data.
- Activities of objects are represented by methods in a class.

Class Design

- Classes contain
  - private instance data variables
  - constructors
  - public getters and setters
  - a toString method
  - private and public methods
- Review the AudioItem class

Class Design

- It can be challenging to decide whether or not to create a class to represent something
- If a class becomes too complex, it is often useful to decompose it into smaller classes to distribute the responsibilities.
- You should find a balance between having classes that are so basic that they don’t depend on others and classes that are so complex they have many or complex dependencies

Class Relationships

- Dependency: A *uses* B
- Aggregation or Composition: A *has-a* B
- Inheritance: A *is-a* B

Class Dependency (“Uses”)

- A *dependency* exists when one class relies on another class in some way
  - Examples: creating an object of that class, invoking the methods of that class, etc.

```
public class Driver {
  private static void main(String[] args){
        AudioItem i1 = new AudioItem(…);
```

  - Here our `Driver` class *depends on* the `AudioItem` class
    - `Driver` uses `AudioItem`

Class Aggregation or Composition (“Has”)

- An *aggregate* is an object that is made up of other objects
  - An instance of one object is part of what defines another object
  - Example: A car *has-a* steering wheel
- An aggregate object contains references to other objects *as instance data*.
  - Example: an `AudioItem` object *has one or more* `String` objects that describe it

Practice

- Write a AudioStoreAccount class.
  - name, account ID, balance, list of audio items owned
  - This is aggregation! An AudioStoreAccount *has a* list of AudioItem objects as part of what describe it.
- Update the driver program to create some accounts.

METHOD DESIGN AND THE THIS KEYWORD

Method Visibility

- Public methods can be invoked by clients
  - Also called *service methods*
- Support or helper methods assist a service method
  - Not intended to be used by a client
  - Should be declared private

Method Decomposition

- A method should be relatively small so that it can be understood as a single entity (or as performing a single task)
- A large method should be decomposed into several smaller methods
  - A public service method may call one or more private support methods
  - Private methods might call other private methods

Method Decomposition (cont.)

- How to decompose methods is an important design decision
  - If you will need the same functionality multiple times, it’s best to put that functionality in a support method.
  - If a method starts to do too many things, it’s probably a good idea to break it up.
  - This is more of an art than a science and it takes practice!

Method Overloading Review

- *Method overloading* is the process of giving a single method name multiple definitions
- If a method is overloaded, the name alone is not sufficient to determine which method is being called
- Rather, we need the entire *signature*, including:
  - The number of parameters
  - The type of parameters
  - The order of the parameters

Overloading Constructors

- It is common to overload constructors
- This provides multiple ways to initialize a new object
  - This is commonly used when you have default values for instance data that can be assigned when the user does not supply values

The `this` Reference

- The `this` reference allows an object to refer to itself
- `this` refers to the current object
- When used inside a method, `this` refers to the object through which the method is invoked
- `this` is also used to invoke an overloaded constructor

The `this` Reference (cont.)

- The `this` reference can be used to distinguish between instance variables and formal parameters with the same name, often used in constructors

```
public Account (String name, long acctNumber,
                double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

The `this` Reference (cont.)

- The `this` keyword is also used to call an overloaded constructor
- Note: this statement *must* be the first line in the constructor
- It’s good design practice to fully implement one constructor (the one with the most parameters) and then invoke that constructor (using `this`) from any overloaded constructor sending in the parameters and default values.

Practice

- Write overloaded constructors for the AudioItem and AudioStoreAccount classes that use default values.
  - Use the `this` keyword to invoke the overloaded constructors.
- Update the parameter names of the constructors and setters.
  - Use the `this` keyword to distinguish between the instance data variables and the formal parameters.

STATIC

Invoking Methods Revisited

- Most methods are invoked through an instance of a class:
  - We create an instance with the `new` operator
  - We invoke a method with the dot operator
- Examples:

  -
    ```
    Scanner scan = new Scanner(System.in);
    scan.nextLine()
    ```

  -
    ```
    JPanel panel = new JPanel();
    panel.add(myButton);
    ```

  -
    ```
    Die d1 = new Die();
    d1.roll();
    ```

Static Methods

- Static methods (also called class methods) are invoked **not** through an object but through the **class** name
    - `double answer = Math.sqrt (25)`
    - `double number = Math.random();`

Static Variables

- Static variables (also called class variables) are accessed through the class itself, not through any instance of the class
  - `BorderLayout.CENTER`
  - `JOptionPane.YES`
  - `Integer.MAX_VALUE`
- One copy/version for the whole class!

Static Methods and Variables

- We declare static methods and variables with the `static` keyword
- A static method or variable is associated with the *class itself*, rather than with any individual instantiated object of the class
  - One copy/version for the whole class!
- Determining if a method or variable should be static is an important design decision.

- By convention, visibility modifiers come first
  - `public static` not `static public`

Instance Variables

- For instance variables, each object has its own data space

  ```
  private String firstName;
  ```

  - Each Student has it’s own first name.
- You update instance data through public methods invoked on an object.

  ```
  student1.setFirstName(“Jim”);
  ```

  - Change the first name of the object student1

Static Variables

- For static variables, only **one** copy of the variable exists for *all* objects of that class

```
public static int numberOfStudents;
```

  - There is only one count of the number of students and it is shared by all objects of the Student class.

Static Variables (continued)

- Changing the value of a static changes it for all objects of that class

```
public Student(…)   {
 …
 Student.numberOfStudents++;
}
```

- Memory space for static variables is created when the class is first referenced.

Static Methods (continued)

- Static methods are invoked through the class, not through any object.

```
public static int numberOfStudents;

public static int getNumberOfStudents() {
    return numberOfStudents;
}
public static void setNumberOfStudents n) {
    numberOfStudents = n;
}
```

Static Methods (cont.)

- Static methods:
  - *cannot* reference instance variables
    - Those variables don’t exist until an object exists (and then each object has its own version of them)
  - *can* reference static variables and local variables

- Static methods:
  - *cannot* directly reference other non-static methods
    - Those must be referenced through an object
  - *can* reference other static methods

- You will get a compiler error if you try to do these things!

Accessing Variables and Methods

	Static Variables	Instance Variables
Static Methods	can access	cannot access
Instance Methods	can access	can access

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	**can access**	**cannot access**
Instance Methods	**can access**	**can access**

```
public Student () {
    …
    Student.numStudents++;
}
```

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	**can access**	**cannot access**
Instance Methods	**can access**	**can access**

```
public String getFirstName() {
   return firstName;
}
```

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	**can access**	**cannot access**
Instance Methods	**can access**	**can access**

```
public static int getNumStudents() {
   return Student.numStudents;
}
```

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	**can access**	**cannot access**
Instance Methods	**can access**	**can access**

```
public static int getStudentName() {
   return firstName;
}
```

would be invoked:

```
Student.getStudentName();
```

static methods are invoked through the class, not through an object… so which student's name should be returned??

Using Static Variables

- Shared data (be careful!)
  - Example: a count of objects
- Shared constants
  - Example: MAX_VALUE

Using Static Methods

- Utility or helper functions
  - send input, get a result
  - Example: Math.sqrt
- Accessing static variables
  - Example: updateCount(int n)

Practice

- Modify the AudioItem class to keep track of how many audio items exist.
- Write classes to represent a Donation and a Donor. Write a driver program.
  - Donations are described by an amount and a date.
  - Donors are described by name, phone number, and a list of donations.

OBJECT REFERENCES

References

- Primitive variables contain their actual value.
- Object variables (usually called object *references*) contain a reference/pointer/address to the place in memory where all the information about the object resides.

Object Parameters

- Keep in mind the rules about assigning variables when you have objects as parameters
  - When you use the assignment operator = you create aliases.
  - Formal object parameters are aliases of actual parameters.

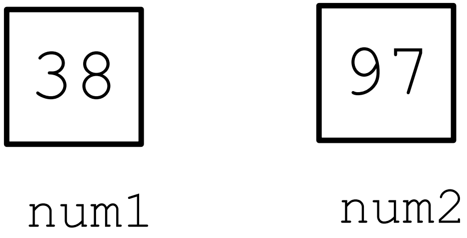
Garbage Collection

- When an object has no more references that point to it, it can no longer be accessed by the program.
- The object is now referred to as *garbage*.
- Java performs *automatic garbage collection* periodically.
  - – Releases an object's memory for future use

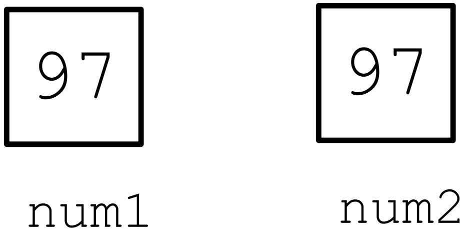
Assignment Revisited

- Assignment takes what is in one variable and places it in the other variable.
  - For primitives, this is the value.

```
int num1 = 38;
int num2 = 97;
```



```
num1 = num2;
```

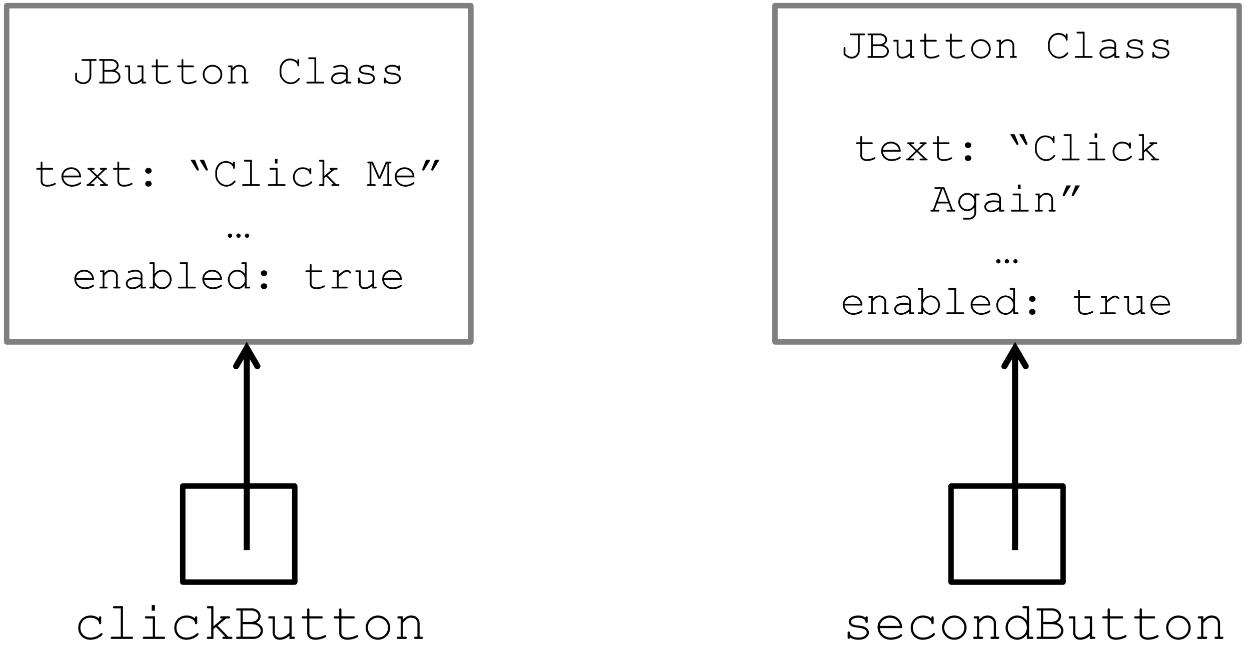


Assignment Revisited (cont.)

- Assignment takes what is in one variable and places it in the other variable.
  - For objects, this is the address.

```
JButton clickButton =
   new JButton(“Click Me”);
JButton secondButton =
   new JButton(“Click Again”);
```

Assignment Revisited (cont.)



JButton Class

text: “Click Me”
…
enabled: true

clickButton

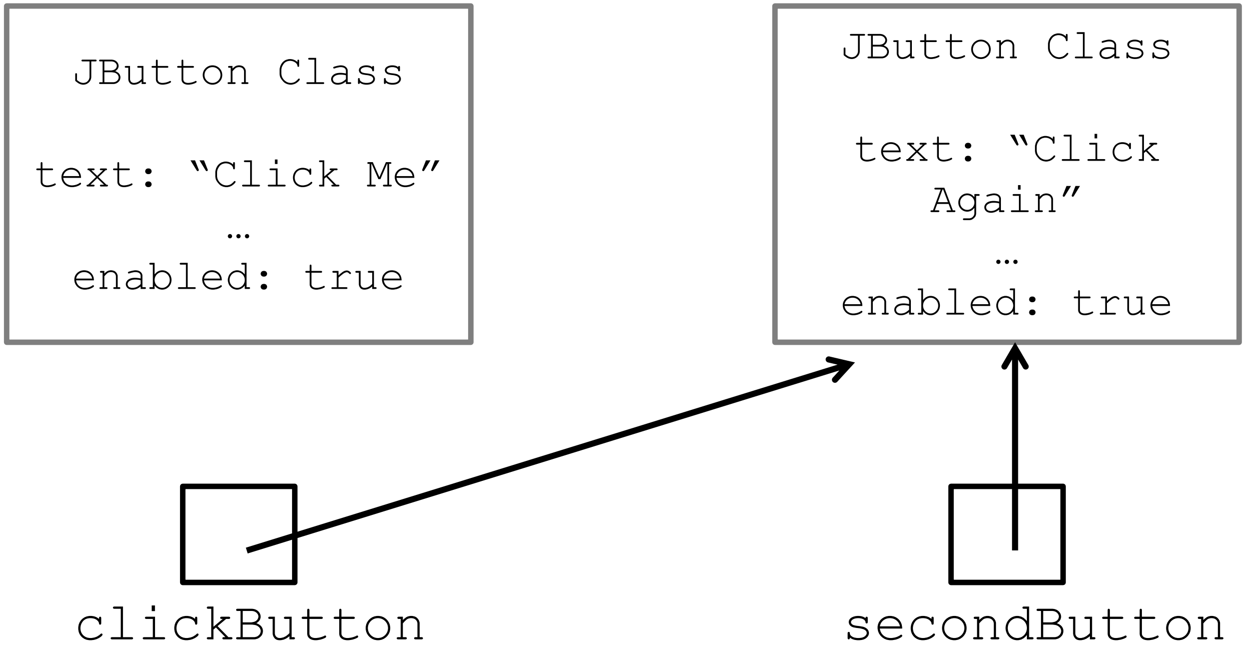
JButton Class

text: “Click Again”
…
enabled: true

secondButton

Assignment Revisited (cont.)

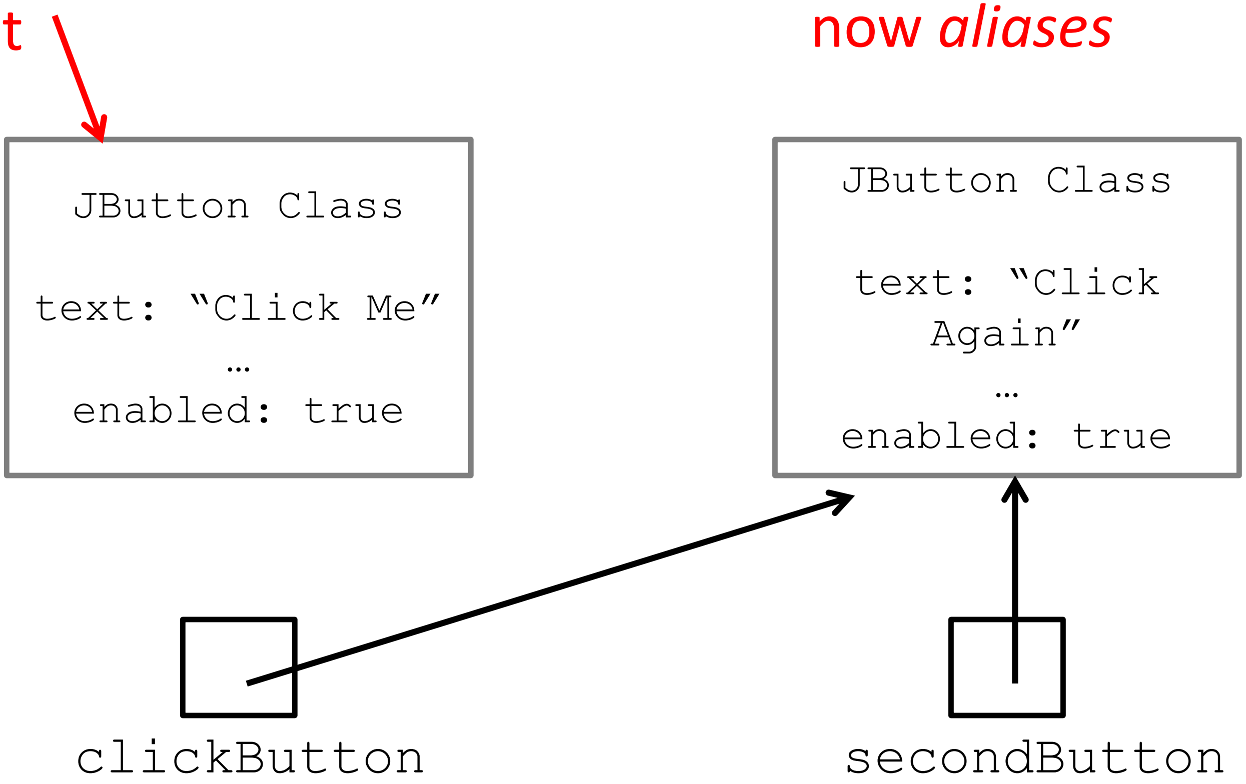
clickButton = secondButton;



Assignment Revisited (cont.)

this object is now lost

clickButton and secondButton are now *aliases*



JButton Class

text: "Click Me"

…

enabled: true

JButton Class

text: "Click Again"

…

enabled: true

clickButton

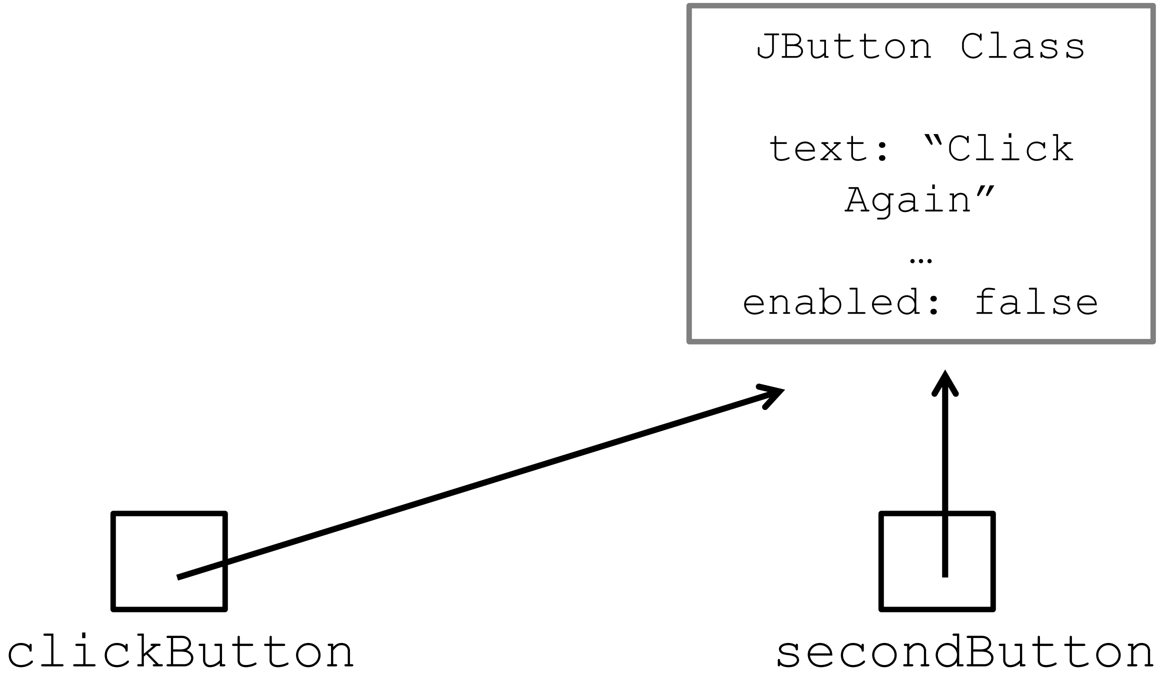
secondButton

Aliases

- Variables that point to the same object are *aliases*
- Changing that object through one reference (i.e., variable name) changes it for all references- because there is only one object

Assignment Revisited (cont.)

secondButton.setEnabled(false);



PASS BY VALUE

Pass By Value

- Parameters in Java are ***passed by value***
- This means that the ***value*** of the actual parameter (the value sent when the method is invoked) is *assigned to* the formal parameter (the parameter listed in the method header)
- For primitives, this is pretty straightforward.

Objects as Parameters

- When we have objects as parameters, it gets a little more tricky!
- Remember what *value* is stored in Java object references… it's a pointer or *reference* to the data that describes that object- the memory address.
  - The memory location is what is passed!
  - So the formal parameter is an *alias*!

Objects as Parameters (cont.)

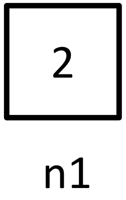
- When an object is passed to a method, the actual parameter and the formal parameter become *aliases* of each other
  - If you change the internal state of an object pointed to by the formal parameter, you change it for the actual parameter as well
  - If you change the formal parameter and have it point to a new object, the original object remains unchanged

Example: Primitive Parameters

```
int n1 = 2;
doubleNumber(n1);
System.out.println("n1 is " + n1);


public static void doubleNumber(int numParam) {
  numParam = numParam * 2;
  System.out.println("inside the doubleNumber
      method, numParam is now " + numParam);
}
```

Example: Primitive Parameters

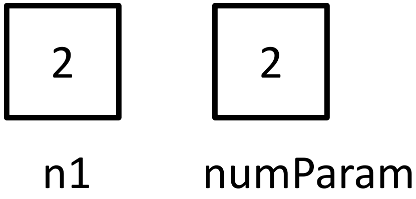
```
int n1 = 2;
doubleNumber(n1);
System.out.println("n1 is " + n1);
```



```
public static void doubleNumber(int numParam) {
  numParam = numParam * 2;
  System.out.println("inside the doubleNumber
      method, numParam is now " + numParam);
}
```

Example: Primitive Parameters

```
int n1 = 2;
doubleNumber(n1);
System.out.println("n1 is " + n1);
```

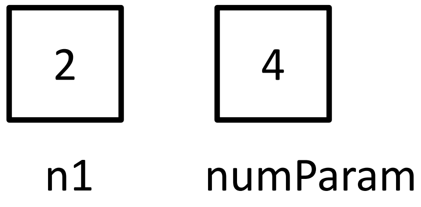


```
public static void doubleNumber(int numParam) {
  numParam = numParam * 2;
  System.out.println("inside the doubleNumber
      method, numParam is now " + numParam);
}
```

behind the scenes, the formal parameter is *assigned* the value of the actual parameter: numParam = n1;

Example: Primitive Parameters

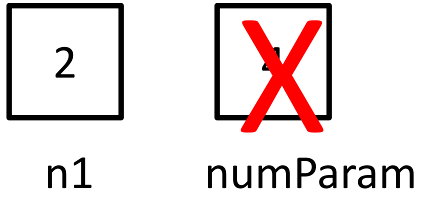
```
int n1 = 2;
doubleNumber(n1);
System.out.println("n1 is " + n1);
```



```
public static void doubleNumber(int numParam) {
  numParam = numParam * 2;
   System.out.println("inside the doubleNumber
     method, numParam is now " + numParam);
}
```

Example: Primitive Parameters

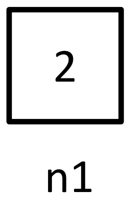
```
int n1 = 2;
doubleNumber(n1);
System.out.println("n1 is " + n1);

public static void doubleNumber(int numParam) {
  numParam = numParam * 2;
   System.out.println("inside the doubleNumber
     method, numParam is now " + numParam);
}
```



method is complete and so formal parameters (and any local variables) are garbage collected

Example: Primitive Parameters

```
int n1 = 2;
doubleNumber(n1);
System.out.println("n1 is " + n1);
```



```
public static void doubleNumber(int numParam) {
  numParam = numParam * 2;
   System.out.println("inside the doubleNumber
     method, numParam is now " + numParam);
}
```

Example: The Rectangle Class

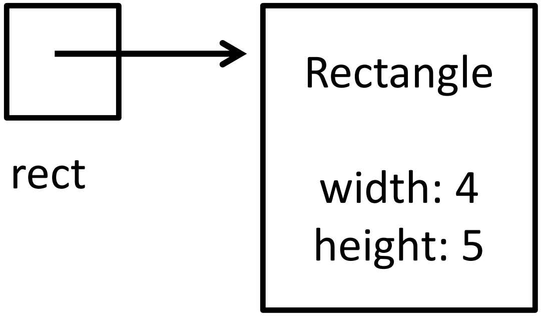
- Java provides a class called Rectangle.
  - Instance data: width and height
  - Methods: getWidth, getHeight, setSize(int, int)

Example: Object Parameters

```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are " +
      rect.getWidth() + " by " + rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are " +
      rect.getWidth() + " by " + rect.getHeight());


public void doubleRectangleDimensions(Rectangle r) {
  int width = (int) r.getWidth();
  int height = (int) r.getHeight();

  r.setSize(width*2, height*2);
}
```

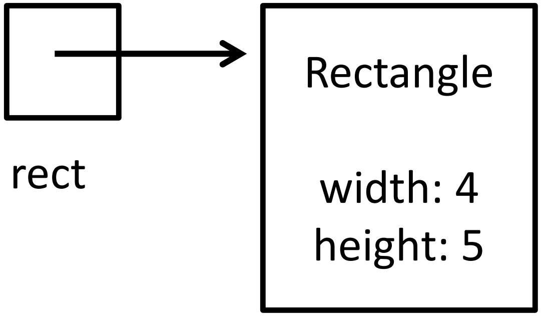
Example: Object Parameters

```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are "
   +     rect.getWidth() + " by " +
   rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are "
   +     rect.getWidth() + " by " +
   rect.getHeight());


public void doubleRectangleDimensions(Rectangle r)
   {
   int width = (int) r.getWidth();
   int height = (int) r.getHeight();

   r.setSize(width*2, height*2);
}
```



Example: Object Parameters

```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are "
   +    rect.getWidth() + " by " +
   rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are "
   +    rect.getWidth() + " by " +
   rect.getHeight());


public void doubleRectangleDimensions(Rectangle r)
   {
   int width = (int) r.getWidth();
   int height = (int) r.getHeight();

   r.setSize(width*2, height*2);
}
```



rect

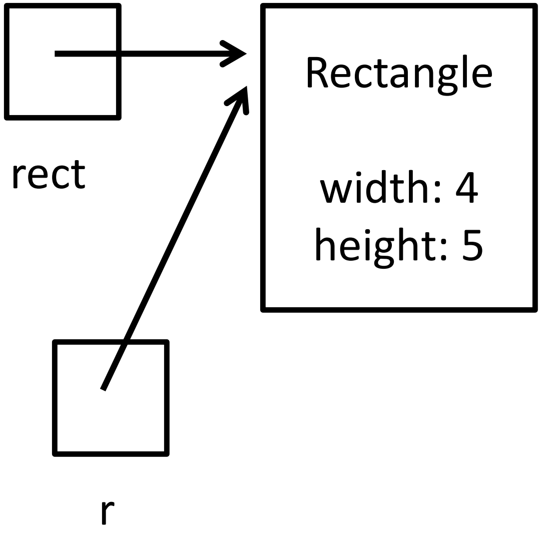
Rectangle

width: 4

height: 5

Example: Object Parameters

```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are "
    +     rect.getWidth() + " by " +
    rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are "
    +     rect.getWidth() + " by " +
    rect.getHeight());


public void doubleRectangleDimensions(Rectangle r)
    {
    int width = (int) r.getWidth();
    int height = (int) r.getHeight();

    r.setSize(width*2, height*2);
}
```



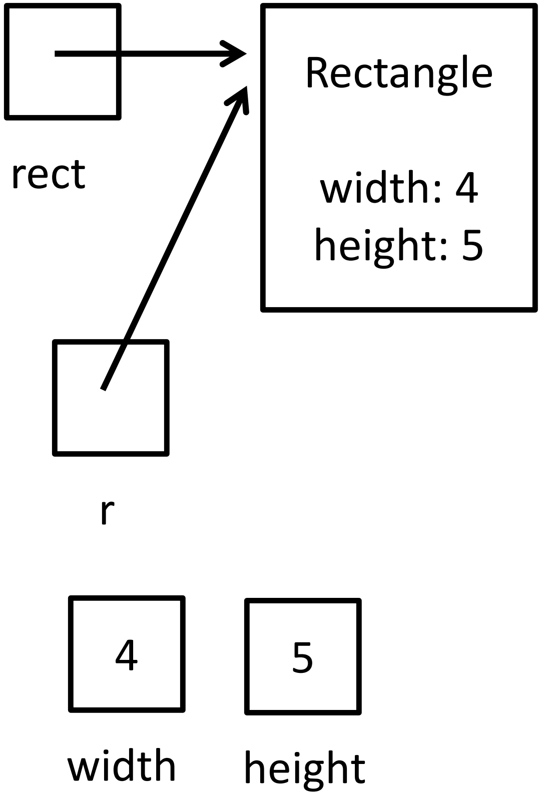
behind the scenes, the formal parameter is *assigned* the value of the actual parameter: r = rect

but rect is an object!! so what is assigned is the memory location

rect and r are now aliases

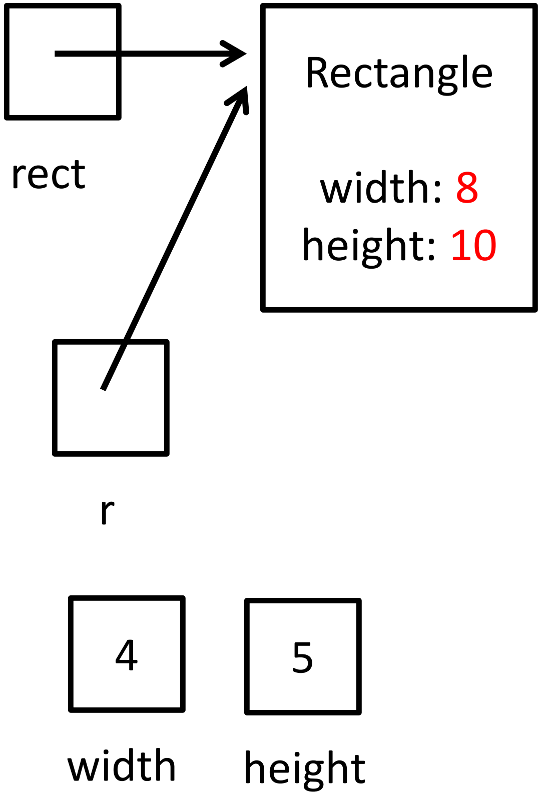
Example: Object Parameters

```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are "
   +     rect.getWidth() + " by " +
   rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are "
   +     rect.getWidth() + " by " +
   rect.getHeight());


public void doubleRectangleDimensions(Rectangle r)
   {
   int width = (int) r.getWidth();
   int height = (int) r.getHeight();

   r.setSize(width*2, height*2);
}
```



Example: Object Parameters

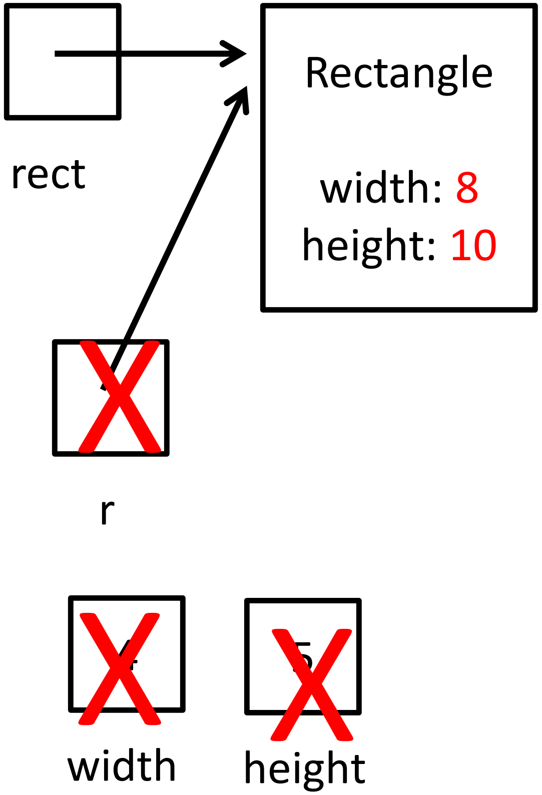
```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are "
   +    rect.getWidth() + " by " +
   rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are "
   +    rect.getWidth() + " by " +
   rect.getHeight());


public void doubleRectangleDimensions(Rectangle r)
   {
   int width = (int) r.getWidth();
   int height = (int) r.getHeight();

    r.setSize(width*2, height*2);
}
```



Example: Object Parameters

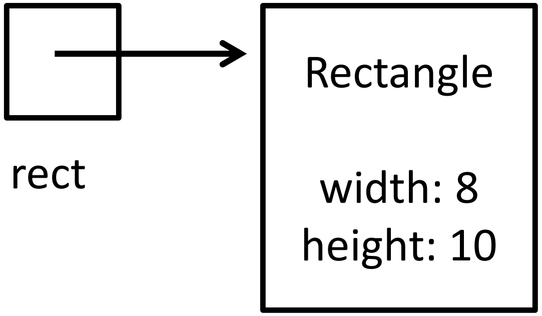
```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are "
   +    rect.getWidth() + " by " +
   rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are "
   +    rect.getWidth() + " by " +
   rect.getHeight());


public void doubleRectangleDimensions(Rectangle r)
   {
   int width = (int) r.getWidth();
   int height = (int) r.getHeight();

    r.setSize(width*2, height*2);
}
```

method is complete and so formal parameters and local variables are garbage collected



Example: Object Parameters

```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are "
   +    rect.getWidth() + " by " +
   rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are "
   +    rect.getWidth() + " by " +
   rect.getHeight());


public void doubleRectangleDimensions(Rectangle r)
   {
   int width = (int) r.getWidth();
   int height = (int) r.getHeight();

    r.setSize(width*2, height*2);
}
```



ENUMS

enum

- A way to provide a restricted set of values
- You can declare variables of this type.
- Examples
  - Sizes: Small, medium, large
  - Suits: Diamonds, hearts, spades, clubs
  - Semesters: Fall, summer, spring

```
enum Size {SMALL, MEDIUM, LARGE};
```

```
Size s1 = Size.LARGE;
// s1 can only hold the values SMALL, MEDIUM,
LARGE, or null

Size s2 = Size.SMALL;
```

Can’t we just use constants?

```
public static final int SMALL = 0;
public static final int MEDIUM = 1;
public static final int LARGE = 2;

public int size = SMALL;
```

- No type safety
  - `public setSize(int size) {`

    // someone could send in -9!
- Allows for illogical results
  - `public static final int FALL = 2;`
  - `FALL == LARGE`. Huh?!
- No easy way to translate to String output
- No way to iterate over all of the choices

Constants vs. enums

- Constants are good things! You should use them in your code.
  - Constants are good for single values like min, max, default values, etc.
- enums are good things! You should use them in your code.
  - enums are best when something has a predefined, finite set of possible values.

enums are Classes!

- You can add constructors, methods, and fields.
  - Constructors are invoked when the enum constants are constructed.
  - Methods and fields are used when you want to associate data or behavior with a constant
- All enums are subclasses of `Enum`.

Example

```
enum Size {
      SMALL("S"), MEDIUM("M"), LARGE("L"),
      EXTRA_LARGE("XL");

      private String abbreviation;

      private Size(String abbreviation) {
            this.abbreviation = abbreviation;
      }
      public String getAbbreviation() {
            return abbreviation;
      }
}
```

Methods for enums

- `toString`
  - Returns the name of the constant
- `static values()` method returns an array of all possible values
  - Example: `Size[] values = Size.values();`
- `ordinal` method returns the position of the constant in the declaration.
  - Starting from 0.
  - Example: `Size.LARGE.ordinal()` returns 2

Practice

- Add an enum to the Employee class to represent whether the employee is full time part time, or inactive.
  - Add data to the enum that represents whether that type of employee gets benefits.
  - Include a toString method.