Classes and Objects

Part I: Designing and Using Your Own Classes, including:

Constructors

Instance Data Variables

Data Scope and Encapsulation

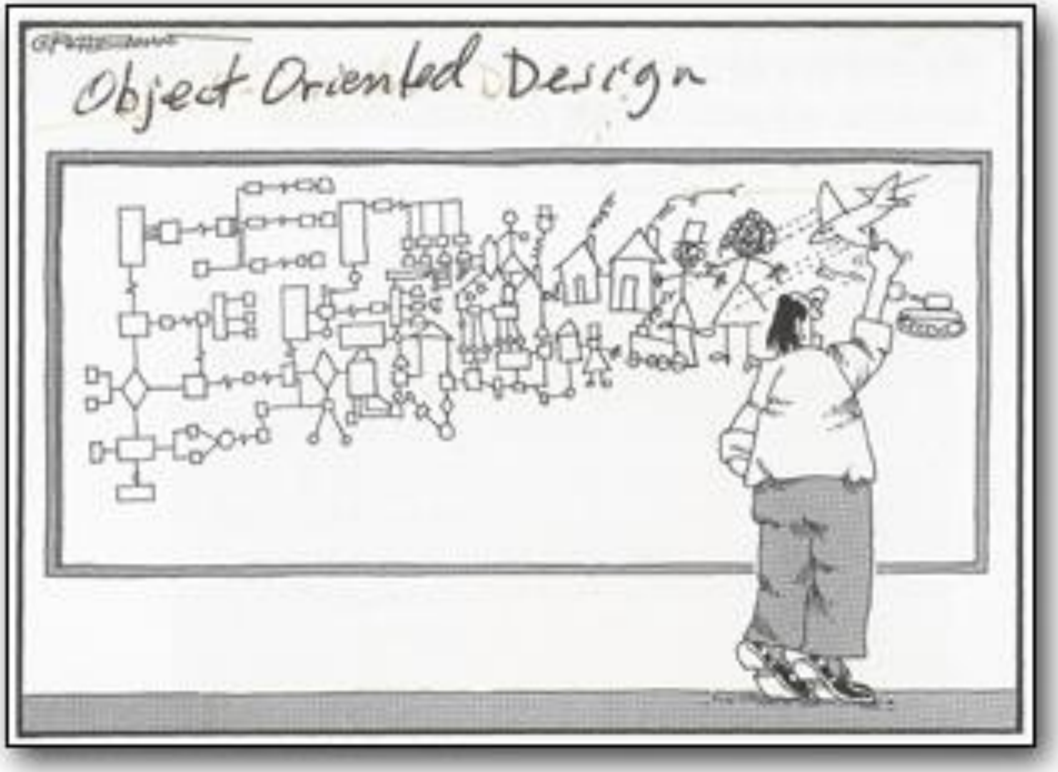
Getters/Setters

toString()

Visibility Modifiers

Invoking Methods

Pass by Value



WRITING YOUR OWN CLASSES

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
- Classes define a blueprint of what all objects of that class will look like.
  - The class is the blueprint. The object is the house.
  - The class is the recipe. The object is the cookie.

Class Design

- Classes define:
  - Characteristics (state)
    - What describes the object
    - Represented by instance data
  - Functionality or actions (behaviors)
    - What the object can do or actions that can be done to the object
    - Represented by methods

Practice

- Write a class to represent a six-sided die.
  - What is the state (characteristics)?
  - What is the behavior?
- Write a program that rolls a die and prints the value.

Class Conventions

- Class names are capital CamelCase
  - PartTimeEmployee, GraduateStudent
- Class names are usually single nouns
  - Die, Student, Employee, String, Scanner
- Class name and file name must be the same
  - Die class is stored in Die.java

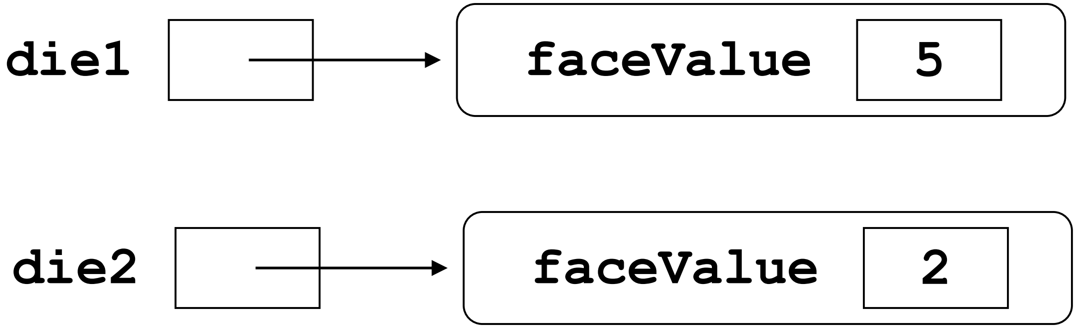
Class Building Blocks

- Instance data variables
- Constructor
- Getters and setters
- toString method
- Other class-specific methods

Instance Data

- Represents the characteristics of the object
- Declared within a class but outside of methods
- Can be accessed by all methods within the class
- Each object has its **own version** of each instance data variable

Instance Data



Constructor

- Create and set up an object
  - An object is an *instance of* the class
  - Initializes the instance data variables using values sent in as parameters or default values defined in the class as constants
- Called when an object is created with the keyword **`new`**
- Has the same name as the class
- Has **no** return type

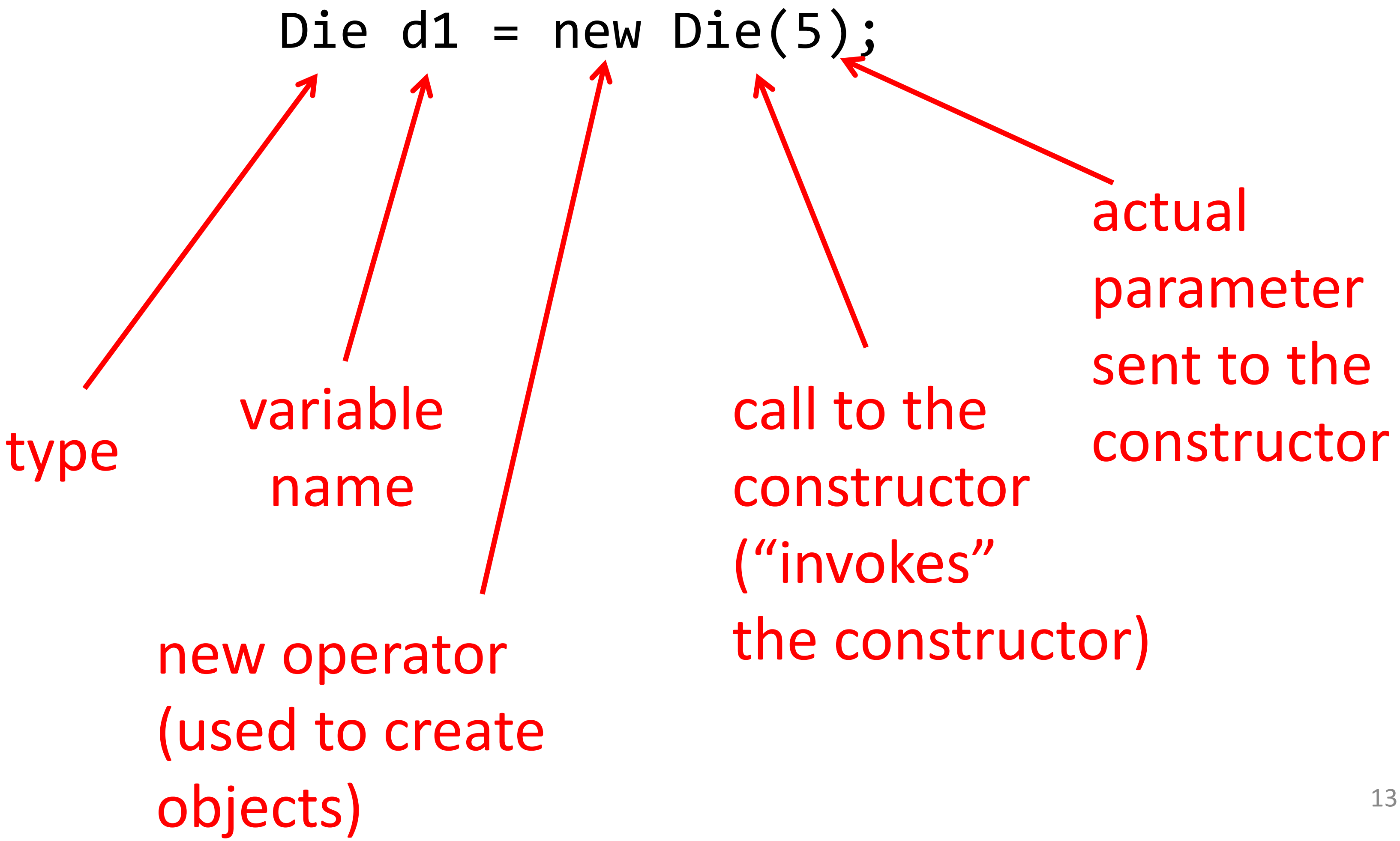
The `new` Keyword

- To create an object, use the `new` keyword and the constructor.
- This is called *instantiating the object* or *creating an instance of* the class.
- `new` does three things:
  - allocates the necessary memory for the object
  - executes the constructor
  - returns the address of (reference to) the object so it can be stored in the variable

Examples: Creating Objects

- Scanner scan = new Scanner(System.in);
  - scan is an *instance of* the Scanner class
- Die d1 = new Die(5);
  - d1 is an *instance of* the Die class

Creating Objects



```
Die d1 = new Die(5);
```

type

variable name

new operator (used to create objects)

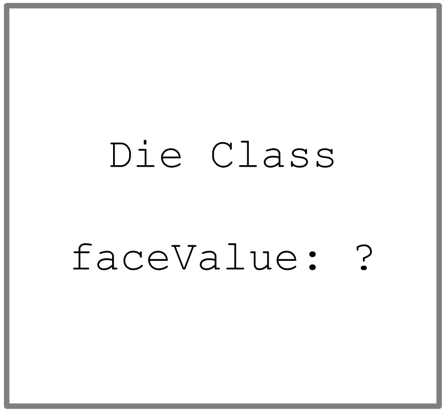
call to the constructor (“invokes” the constructor)

actual parameter sent to the constructor

Creating Objects

Die d1 = new Die(5);

- **new** does three things:
  - allocates the necessary memory for the object
  - executes the code inside of the constructor
  - returns the address of (reference to) the object so it can be stored in the variable



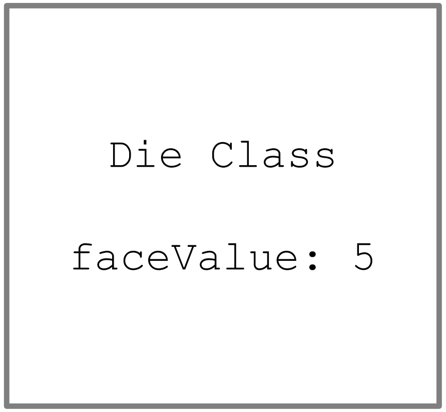
Die Class

faceValue: ?

Creating Objects

```
Die d1 = new Die(5);
```

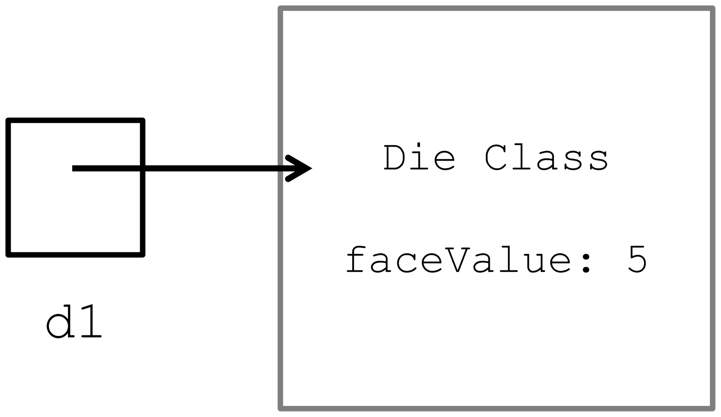
- **new** does three things:
  - allocates the necessary memory for the object
  - executes the code inside of the constructor
  - returns the address of (reference to) the object so it can be stored in the variable



Creating Objects

Die d1 = new Die(5);

- **new** does three things:
  - allocates the necessary memory for the object
  - executes the code inside of the constructor
  - returns the address of (reference to) the object so it can be stored in the variable



The Default or No-Arg Constructor

- If you have *no* constructor in your class, Java will add a default, no-argument constructor behind the scenes:

```
public Die() { }
```

- If you add *any* constructor in your class, that's the only constructors you have. Java doesn't add anything else.
- This will matter later for inheritance!

Visibility Modifiers

- public
  - Can be referenced anywhere
- private
  - Can be referenced within a class
- protected
  - Related to inheritance… more to come later…
- default
  - can be referenced within the package… more to come later…

Visibility Modifiers

use just these for now

- public
  - Can be referenced anywhere
- private
  - Can be referenced within a class
- protected
  - Related to inheritance… more to come later…
- default
  - can be referenced within the package… more to come later…

Encapsulation

- Each objects protects its own information (instance data variables).
- We (the developer of the class) do not let a *client* (someone who uses the class) directly manipulate the instance data variables.
  - If they could, they could modify the object in ways we don’t want!
- A *client* can *change the state* of our object **only** by invoking provided methods.

Encapsulation

- Keep instance data variables private.
- Provide public methods to access and modify the variables appropriately.

Public Variables… DON'T DO IT!

- Public instance variables violate encapsulation
  - Allow a client to modify values directly
- Instance variables should only be modified through provided methods
- Keep instance data variables private!

- It's okay to make constants public because they cannot be changed
  - So this doesn't violate encapsulation

Getters and Setters

- Provides access to the instance data
- Also called *accessors* and *mutators*
- Getter
  - Returns the current value of a variable

```
public TYPE getVARNAME() {
  return VARNAME;
}
```

- Setter
  - Updates the value of a variable
  - Can define parameters for valid values- validity checks

```
public void setVARNAME(TYPE NEWVALUE) {
  VARNAME = NEWVALUE;
}
```

Getters and Setters (cont.)

- Getter

```
public TYPE getVARNAME() {
  return VARNAME;
}
```

- Setter

```
public void setVARNAME(TYPE NEWVALUE) {
  VARNAME = NEWVALUE;
}
```

instance variable

Validity Checks in Setters

```
public void setVARNAME(TYPE NEWVALUE) {
  if( NEWVALUE meets some test) {
      VARNAME = NEWVALUE;
  } else {
      // maybe do nothing
      // maybe print an error message
      // maybe throw an exception
  }
}
```

toString Method

- Returns a text representation of the object
- Automatically called behind-the-scenes when the object is concatenated with a String or put inside a println method
- Has this **exact** header:

```
public String toString() {
    return STRINGVAL;
}
```

Class Specific Methods

- The method header is used to declare the method:
  - Visibility
  - Return type
  - Method name
  - Formal parameters
    - List of type and name separated by commas
- Methods represent the functionality for the class.
- Methods often *change the state* of the object (i.e., update the value of the instance data variables).

Summary of Classes

- Blueprints of what objects look like
- Contain:
  - private instance variables
    - Characteristics of objects
  - Constructors
    - Set up the object
  - getters and setters
    - Provide access to private instance data variables
  - toString method
    - Provides a text representation of the object
  - public and private methods
    - Specific to the class functionality

Data Scope

- The *scope* of a variable is the area where it can be used or referenced
    - Determined by where the variable is **declared** (where we specify type and name)
- Instance data variables
    - Declared in the class
    - Can be used anywhere in the class
    - Used to describe an object
    - Live as long as the object lives
- Local data
    - Declared inside of a method
    - Can be used only in that method
    - Are garbage collected when the method ends

Parameters

- Formal parameters
  - Declared in the method header
  - Visible within the method
  - Garbage collected when the method ends
  - Use when the information is needed from somewhere outside the method
- Actual parameters
  - Values passed during method invokation
  - Used when sending information to a method

Variable Summary

	What It Is	When to Use It	Where It Can Be Used
instance data variable	declared inside the class, outside of any method	when it describes an object’s characteristics	anywhere in the class
local variable	declared inside of a method	when you only need the variable for the method	inside the method only; garbage collected when the method ends
formal parameter	listed in the method header	when you need input from elsewhere to accomplish a task	inside the method only; garbage collected when the method ends
actual parameter	included in the method invokation	when you need to send input to a method	

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
- Local variables and formal parameters are garbage collected when the method ends.

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

Visibility Modifiers

	`public`	`private`
Variables	**Violate encapsulation**	**Enforce encapsulation**
Methods	**Provide services to clients**	**Support other methods in the class**

INVOKING METHODS

Method Invocation

- Invoking a method is like asking an object to “do something.”
- When invoked, some methods return a value.
  - The returned value can be saved, used, or ignored.
- void methods do not return a value.

Method Invocation

- To invoke a method, specify:
  - invoking object (or class) followed by dot operator
  - method name
  - actual parameters
- Examples:

```
d1.roll();
d1.setFaceValue(6);
```

actual parameter

Method Invocation

- We will deal with these special cases later, but I include them here now just for accuracy!

- If the method invoked is in the same class, you only need the method name.
  - You can omit the invoking object.
  - Or you can use `this` as the invoking object.
- If the method is `static`, you invoke it with the name of the class, rather than with an object.

Declaration vs. Invocation

- Declaring a method specifies what happens when it runs.
- Invoking a method actually *makes* it run.

Practice

- Write a Dice class that represents a pair of dice.

Practice

- Write a class to represent an item sold at an audio store (e.g., music, audio book, etc.).
  - All items sold are described by title, price, duration.

OBJECT REFERENCES

What is Stored in Memory

- Primitive variables
  - The actual value- the data
- Object variables (also called object *references*)
  - A reference/ pointer/ memory address to the place in memory where all the information about the object resides

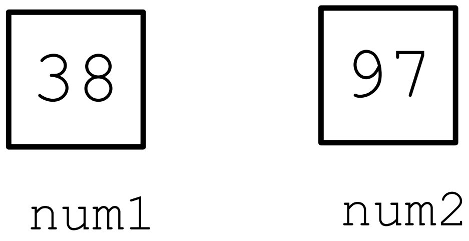
Assignment Statements

- Assignment takes the **value** on the right and stores it in the variable on the left.
- Think about what **the value** is!
  - It’s different for primitives and objects!

Assignment- Primitives

- Assignment takes the value on the right and stores it in the variable on the left.
  - For primitives, the value is just the data!

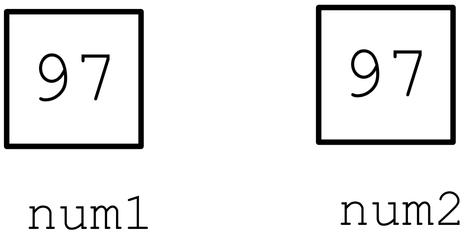
```
int num1 = 38;

int num2 = 97;
```



Assignment- Primitives

```
num1 = num2;
```

- What is the **value** of num2?
- Because it’s a primitive, the value is just the data! So the data- the actual number- is placed into num1.



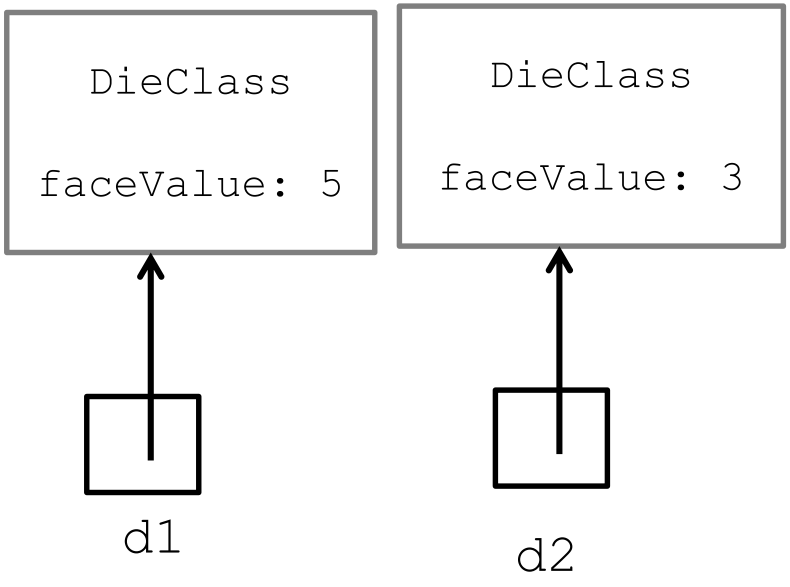
97 97

num1 num2

Assignment- Objects

- Assignment takes the value on the right and stores it in the variable on the left.
  - For objects, the value is a memory address!

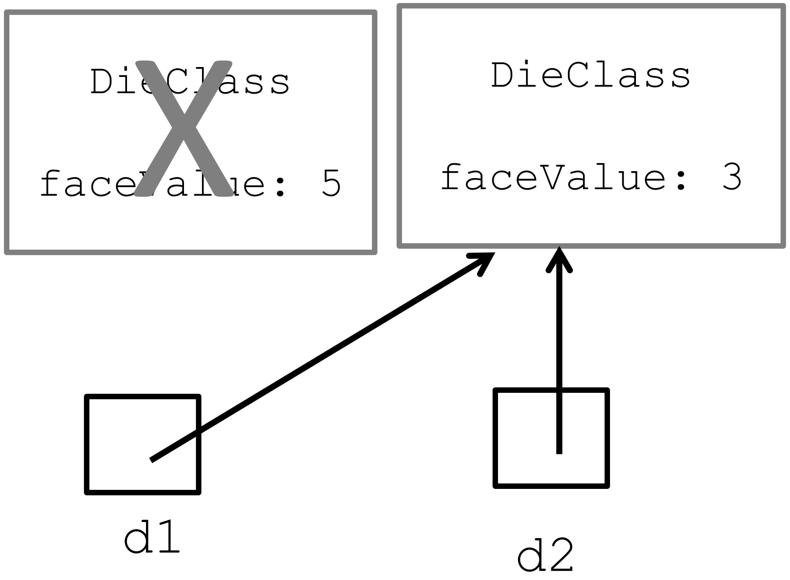
```
Die d1 = new Die(5);
Die d2 = new Die(3);
```



Assignment- Objects

```
d1 = d2;
```

- What is the **value** of d2?
- Because it’s an object, the value is the address!
- So now d1 and d2 point to the exact same place in memory- the same address!
- Because no reference points to the other Die object, it gets garbage collected.

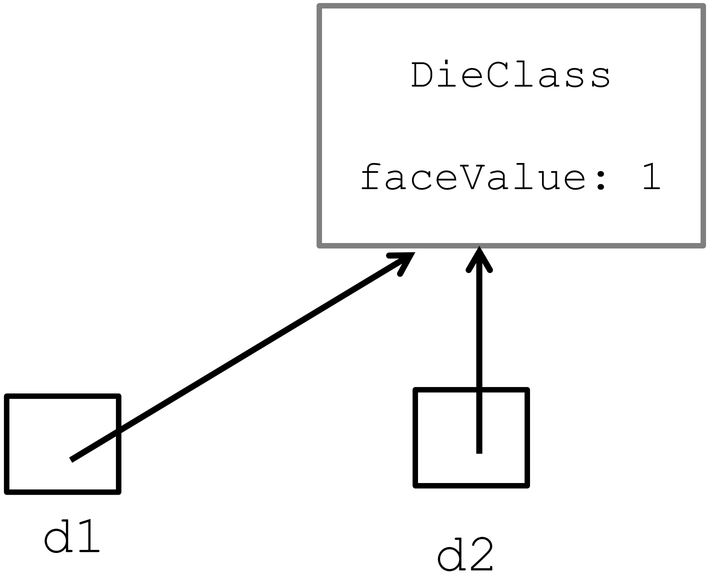


Aliases

- d1 and d2 are now **aliases**.
- Variables that point to the same object (the same place in memory) are *aliases*
- Changing that object through one reference (i.e., one variable name) changes it for *all* references- because there is only **one** object!

Aliases

d2.setFaceValue(1);



INVOKING METHODS AND PASS BY VALUE

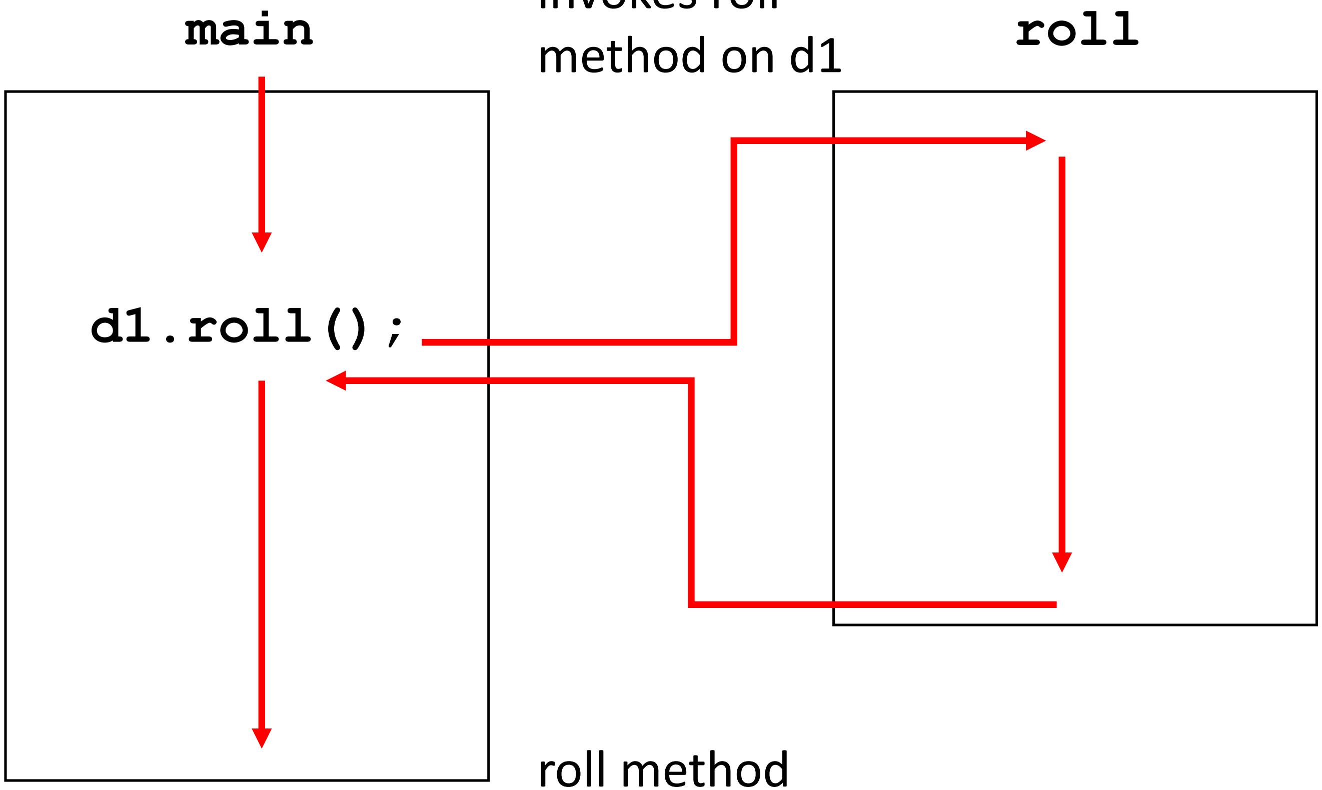
Flow of Control

- Java executes statements in order, starting with the first statement in the main method.
- When a statement contains a method invokation, the flow of control jumps to that method and executes that method line by line.
  - You can think of the original method as being *put on pause* until the invoked method completes.

Flow of Control

- An invoked method is complete when you reach:
  1. a `return` statement or
  2. the end of the method
- When the invoked method is complete, control returns to where the method was called and continues.

Flow of Control Example



main

roll

`d1.roll();`

main method invokes roll method on d1

roll method returns control to main method

Invoking Methods with Parameters

- Formal parameters are defined in the method header
  - They last as long as the method lasts.
  - When the method is over, these parameters are gone!
- Actual parameters are the values sent when the method is invoked.

Pass By Value

- Parameters in Java are ***passed by value***
- This means that the ***value*** of the actual parameter is *assigned to* the formal parameter.
  - But remember how assignment works for primitives vs objects!

Passing Parameters

- When a method is invoked, it’s as if there is assignment statement executed behind the scenes:

```
    formalParam = actualParam;
```

- This is an assignment statement!
  - When you use the assignment operator with objects, you create **aliases**.
  - Formal object parameters are **aliases** of actual parameters.

Objects as Parameters

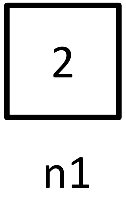
- When an object is passed to a method, the actual parameter and the formal parameter become *aliases* of each other
  - If you change the internal state of the formal parameter by invoking a method, you change it for the actual parameter as well

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);
```



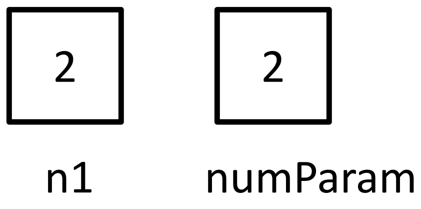
2

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);
```

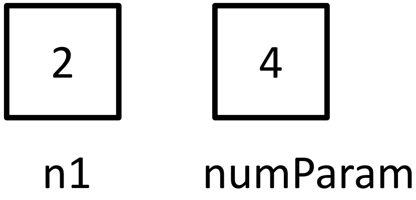


```
public static void doubleNumber(int numParam) {
  numParam = numParam * 2;
  System.out.println("inside the doubleNumber
      method, numParam is now " + numParam);
}
```

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);
```



2

4

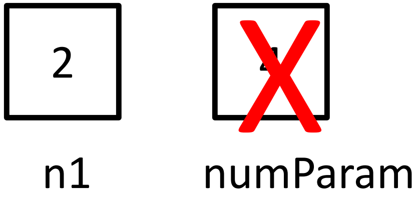
n1

numParam

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

Example: Primitive Parameters

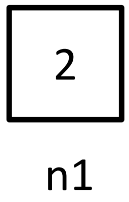
```
int n1 = 2;
doubleNumber(n1);
System.out.println("n1 is " + n1);

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

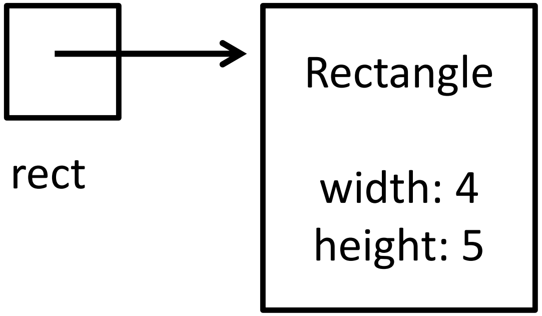
- Java provides a class called Rectangle.
  - Instance data: width and height
  - Methods: getWidth, getHeight, setSize(int, int)

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);
}
```

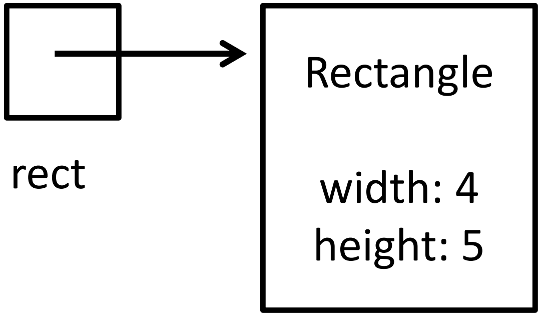
Example: Object Parameters

```
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);
}
```



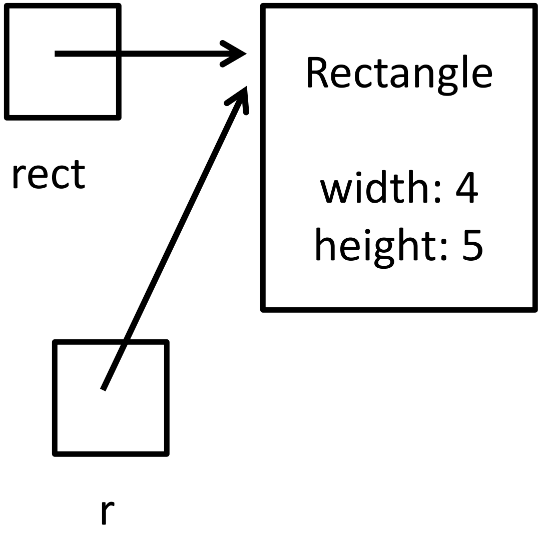
Example: Object Parameters

```
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);
}
```



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);
}
```



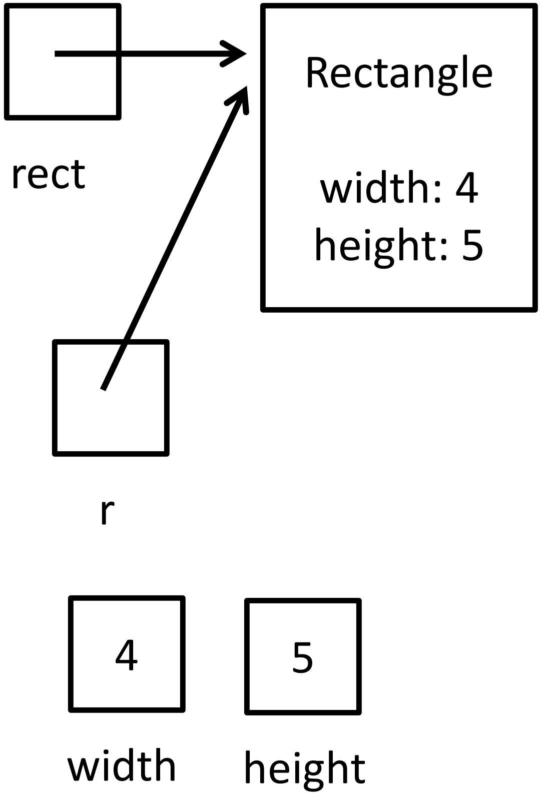
behind the scenes, the formal parameter is *assigned* the value of the actual parameter: r = rect

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

rect and r are now aliases

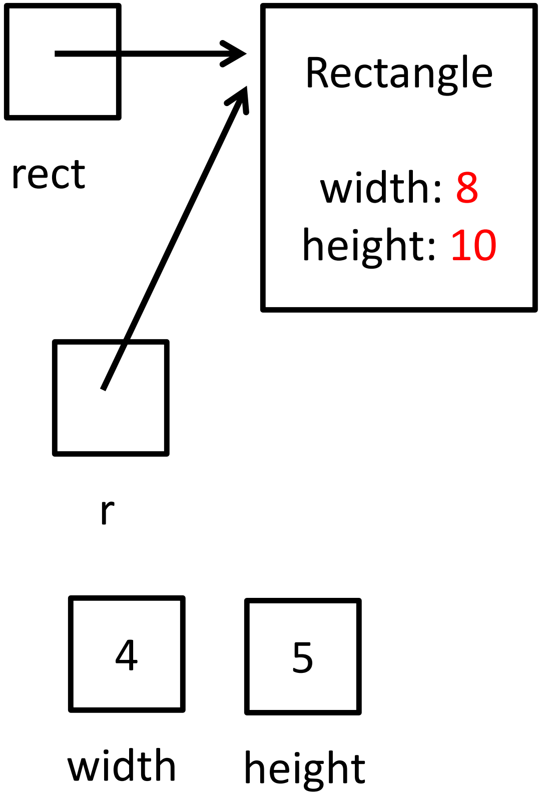
Example: Object Parameters

```
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

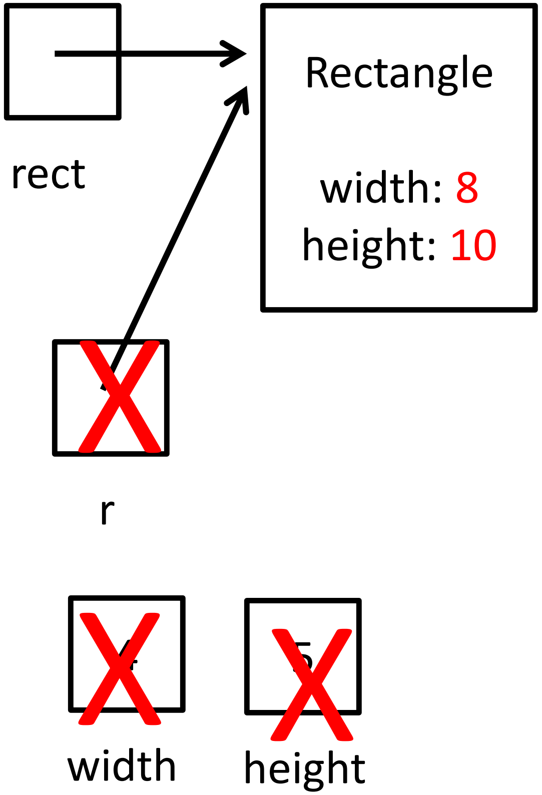
```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are "
   +    rect.getWidth() + " by " +
   rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are "
   +    rect.getWidth() + " by " +
   rect.getHeight());


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

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



Example: Object Parameters

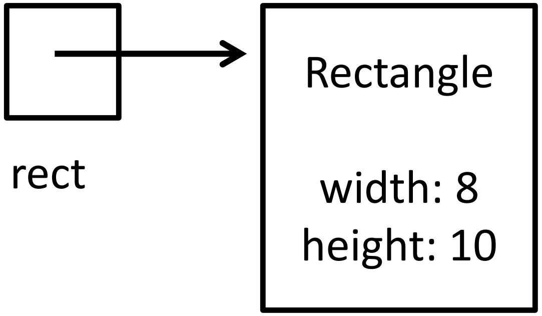
```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are "
   +     rect.getWidth() + " by " +
   rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are "
   +     rect.getWidth() + " by " +
   rect.getHeight());


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);
}
```

One Last Note on Objects as Parameters

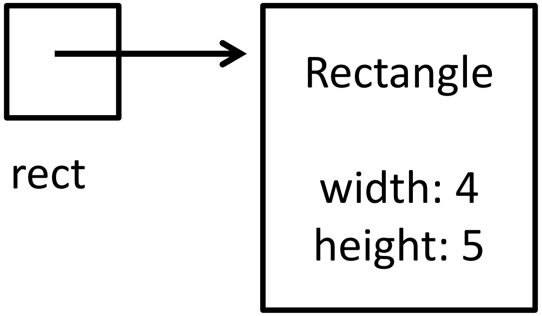
- If you change the formal parameter and have it point to a new object, you break the alias link.
  - So changes made to the formal parameter no longer affect the actual parameter.
  - You rarely want to do this- it’s most often an error.

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) {
  r = new Rectangle(1, 3);
  r.setSize(2, 2);
}
```

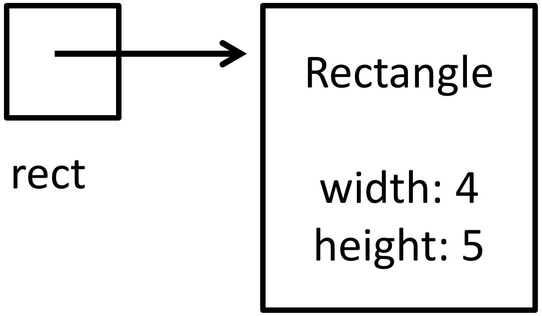
Example: Object Parameters

```
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)
   {
    r = new Rectangle(1, 3);
   r.setSize(2, 2);
}
```



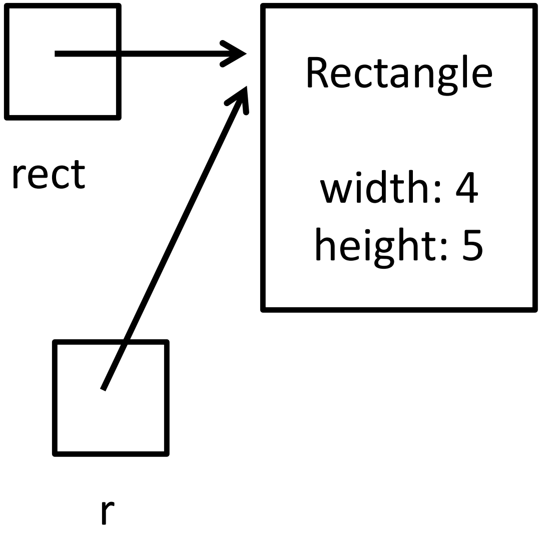
Example: Object Parameters

```
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)
   {
   r = new Rectangle(1, 3);
   r.setSize(2, 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)
   {

   r.setSize(2, 2); ();
   int height = (int) r.getHeight();

}
```



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

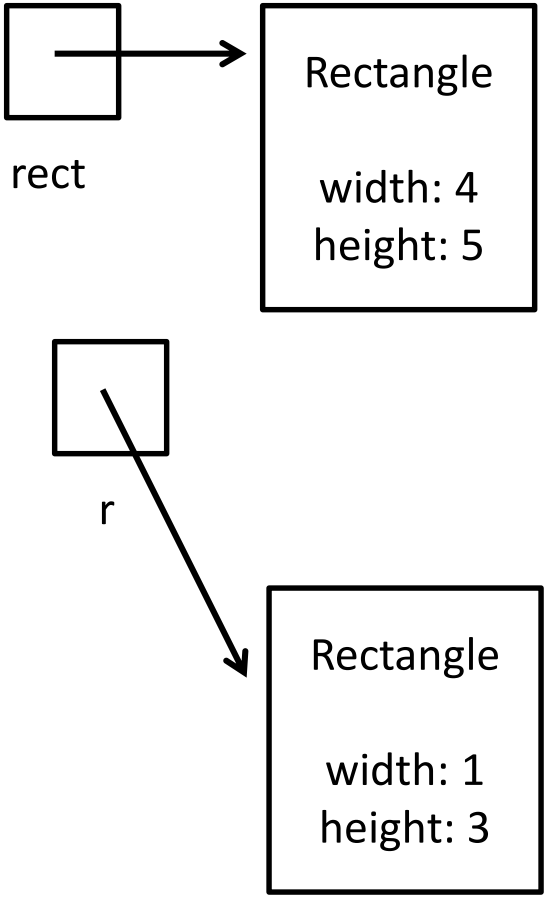
but rect is an object!! so what is assigned is the memory location

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)
   {
   r = new Rectangle(1, 3);

   r.setSize(2, 2);

}
```

the alias link is broken!



rect

Rectangle

width: 4
height: 5

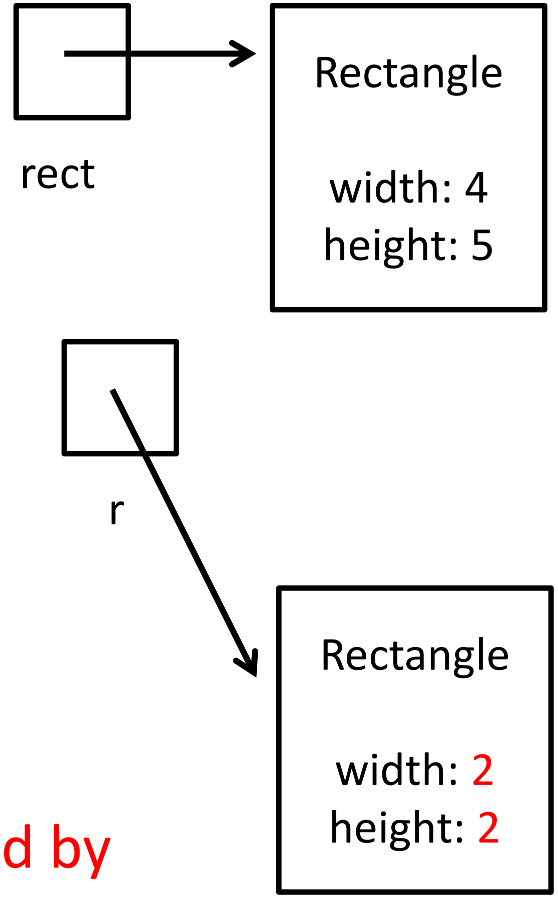
r

Rectangle

width: 1
height: 3

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)
   {
   r = new Rectangle(1, 3);

   r.setSize(2, 2);
}
```

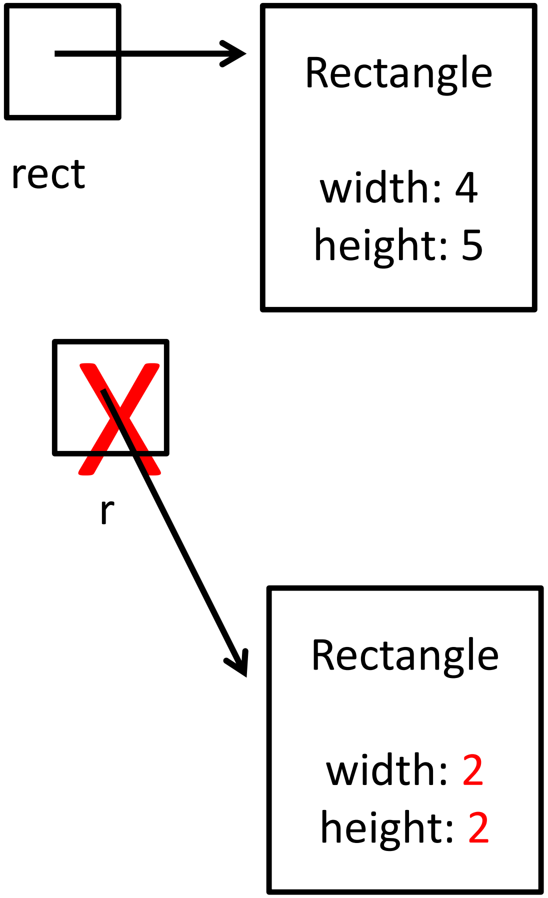


the alias link is broken so rect is unaffected by any changes made inside the method!

Example: Object Parameters

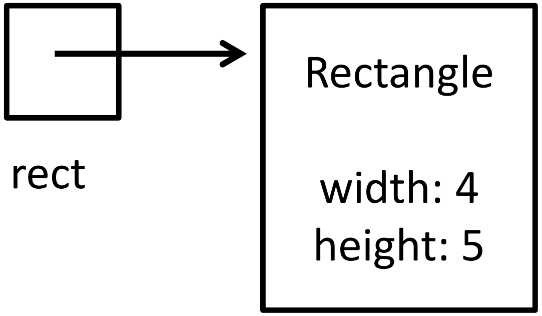
```
Rectangle rect = new Rectangle(4, 5);
System.out.println("The rectangle dimensions are "
   +     rect.getWidth() + " by " +
   rect.getHeight());
doubleRectangleDimensions(rect);
System.out.println("The rectangle dimensions are "
   +     rect.getWidth() + " by " +
   rect.getHeight());


public void doubleRectangleDimensions(Rectangle r)
   {
   r = new Rectangle(1, 3);

   r.setSize(2, 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)
   {
   r = new Rectangle(1, 3);



   r.setSize(2, 2);
}
```



Bottom Line

- When passing objects as parameters, the actual and formal parameters are aliases.
  - Methods invoked on the formal parameter (using the dot operator) affect the actual parameter- because they are the same object!
- Reassigning the formal parameter breaks the alias link.
  - This is usually a mistake.

Practice

- Write an Employee class that represents employees at a company.
  - An employee is represented by a name, id, and phone number.