Unit 5, Part 1

Classes as Blueprints: How to Define New Types of Objects

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Types of Decomposition

- When writing a program, it's important to decompose it into manageable pieces.
- We've already seen how to use *procedural* decomposition.
  - break a task into smaller subtasks, each of which gets its own method
- Another way to decompose a program is to view it as a collection of *objects.*
  - referred to as *object-oriented programming*

Review: What is an Object?

- An object groups together:
  - one or more data values (the object's *fields*)
  - a set of operations that the object can perform (the object's *methods*)

Review: Using an Object's Methods

- An object's methods are different from the static methods that we've been writing thus far.
  - they're called *non-static* or *instance* methods
- When using an instance method, we specify the object to which the method belongs by using dot notation:

```
String firstName = "Perry";
int len = firstName.length();
```

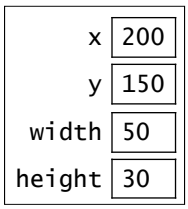
- Using an instance method is like sending a message to an object, asking it to perform an operation.
- We refer to the object on which the method is invoked as either:
  - the *called object*
  - the *current object*

Review: Classes as Blueprints

- We've been using classes as containers for our programs.
- A class can also serve as a blueprint – as the definition of a new type of object.
  - specifying the fields and methods that objects of that type will have
- The objects of a given class are built according to its blueprint.
- Objects of a class are referred to as *instances* of the class.

`Rectangle` Objects

- Java comes with a built-in `Rectangle` class.
  - in the `java.awt` package
- Each `Rectangle` object has the following fields:
  - `x` – the `x` coordinate of its upper left corner
  - `y` – the `y` coordinate of its upper left corner
  - `width`
  - `height`
- Here's an example of one:



`Rectangle` Methods

- A `Rectangle`'s methods include:

```
void grow(int h, int v)
void translate(int x, int y)
double getWidth()
double getHeight()
double getX()
double getY()
```

Writing a "Blueprint Class"

- To illustrate how to define a new type of object, let's write our own class for `Rectangle` objects.

```
public class Rectangle {
    ...
```

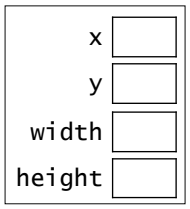
- As always, the class definition goes in an appropriately named text file.
  - in this case: `Rectangle.java`

Using Fields to Capture an Object's State

- Here's the first version of our `Rectangle` class:

```
public class Rectangle {
    int x;
    int y;
    int width;
    int height;
}
```

  - it declares four fields, each of which stores an `int`
  - each `Rectangle` object gets its own set of these fields



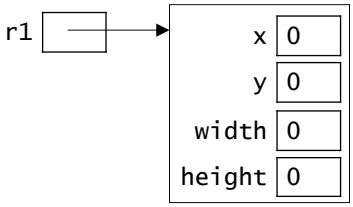
- Another name for a field is an *instance variable.*

Using Fields to Capture an Object's State (cont.)

- For now, we'll create `Rectangle` objects like this:

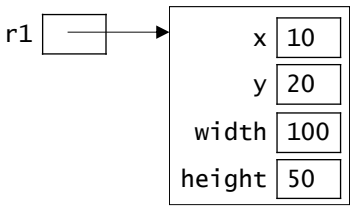
```
Rectangle r1 = new Rectangle();
```

- The fields are initially filled with the default values for their types.
  - just like array elements



- Fields can be accessed using dot notation:

```
r1.x = 10;
r1.y = 20;
r1.width = 100;
r1.height = 50;
```



Client Programs

- Our `Rectangle` class is *not* a program.
  - it has no `main` method
- Instead, it will be used by code defined in other classes.
  - referred to as *client programs* or *client code*
- More generally, when we define a new type of object, we create a building block that can be used in other code.
  - just like the objects from the built-in classes: `String`, `Scanner`, `File`, etc.
  - our programs have been clients of those classes

Initial Client Program

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.x = 10;       r1.y = 20;
        r1.width = 100;  r1.height = 50;

        Rectangle r2 = new Rectangle();
        r2.x = 50;       r2.y = 100;
        r2.width = 20;   r2.height = 80;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        int area1 = r1.width * r1.height;
        System.out.println("area = " + area1);

        System.out.println("r2: " + r2.width + " x " + r2.height);
        int area2 = r2.width * r2.height;
        System.out.println("area = " + area2);

        // grow both rectangles
        r1.width += 50;  r1.height += 10;
        r2.width += 5;   r2.height += 30;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Using Methods to Capture an Object's Behavior

- It would be useful to have a method for growing a `Rectangle`.
- One option would be to define a static method:

```
public static void grow(Rectangle r, int dWidth, int dHeight) {
    r.width += dWidth;
    r.height += dHeight;
}
```

- This would allow us to replace the statements

```
r1.width += 50;
r1.height += 10;
```

with the method call

```
Rectangle.grow(r1, 50, 10);
```

Using Methods to Capture an Object's Behavior

- It would be useful to have a method for growing a `Rectangle`.
- One option would be to define a static method in our `Rectangle` class:

```
public static void grow(Rectangle r, int dWidth, int dHeight) {
    r.width += dWidth;
    r.height += dHeight;
}
```

- This would allow us to replace these statements in the client

```
r1.width += 50;
r1.height += 10;
```

with the method call

```
Rectangle.grow(r1, 50, 10);
```

(Note: We need to use the class name, because we're calling the method from outside the `Rectangle` class.)

Using Methods to Capture an Object's Behavior (cont.)

- A better approach is to give each `Rectangle` object the ability to grow itself.
- We do so by defining a non-static or instance method.
- We'll use dot notation to call the instance method:

```
r1.grow(50, 10);
```

instead of `Rectangle.grow(r1, 50, 10);`

- This is like sending a message to `r1`, asking it to grow itself.

Using Methods to Capture an Object's Behavior (cont.)

- Here's our `grow` instance method:

```
public void grow(int dWidth, int dHeight) { // no static
    this.width += dWidth;
    this.height += dHeight;
}
```

- We don't pass the `Rectangle` object as an explicit parameter.
- Instead, the Java keyword `this` gives us access to the called object.
  - every instance method has this special variable
  - referred to as the *implicit parameter*
- Example: `r1.grow(50, 10)`
  - `r1` is the called object
  - `this.width` gives us access to `r1`'s `width` field
  - `this.height` gives us access to `r1`'s `height` field

Comparing the Static and Non-Static Versions

- Static:

```
public static void grow(Rectangle r, int dWidth, int dHeight) {
    r.width += dWidth;
    r.height += dHeight;
}
```

  - sample method call: `Rectangle.grow(r1, 50, 10);`

- Non-static:

```
public void grow(int dWidth, int dHeight) {
    this.width += dWidth;
    this.height += dHeight;
}
```

  - there's no keyword `static` in the method header
  - the `Rectangle` object is not an explicit parameter
  - the implicit parameter `this` gives access to the object
  - sample method call: `r1.grow(50, 10);`

Omitting the Keyword `this`

- The use of `this` to access the fields is optional.
  - example:

```
public void grow(int dWidth, int dHeight) {
    width += dWidth;
    height += dHeight;
}
```

Another Example of an Instance Method

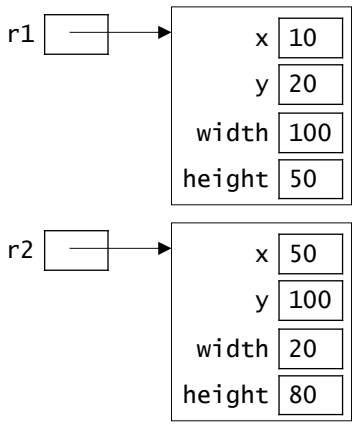
- Here's an instance method for getting the area of a `Rectangle`:

```
public int area() {
    return this.width * this.height;
}
```

- Sample method calls:

```
int area1 = r1.area();
int area2 = r2.area();
```

  - we're asking `r1` and `r2` to give us their areas
  - no explicit parameters are needed because the necessary info. is in the objects' fields!



Types of Instance Methods

- There are two main types of instance methods:
  - *mutators* – methods that change an object's internal state
  - *accessors* – methods that retrieve information from an object without changing its state
- Examples of mutators:
  - `grow()` in our `Rectangle` class
- Examples of accessors:
  - `area()` in our `Rectangle` class
  - `String` methods: `length()`, `substring()`, `charAt()`

Second Version of our `Rectangle` Class

```
public class Rectangle {
    int x;
    int y;
    int width;
    int height;

    public void grow(int dWidth, int dHeight) {
        this.width += dWidth;
        this.height += dHeight;
    }

    public int area() {
        return this.width * this.height;
    }
}
```

Which method call increases `r`'s height by 5?

```
public class Rectangle {
    int x;
    int y;
    int width;
    int height;

    public void grow(int dWidth, int dHeight) {
        this.width += dWidth;
        this.height += dHeight;
    }

    public int area() {
        return this.width * this.height;
    }
}
```

- Consider this client code:

```
Rectangle r = new Rectangle();
r.width = 10;
r.height = 15;

______???______;
```

Initial Client Program

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.x = 10;       r1.y = 20;
        r1.width = 100;  r1.height = 50;

        Rectangle r2 = new Rectangle();
        r2.x = 50;       r2.y = 100;
        r2.width = 20;   r2.height = 80;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        int area1 = r1.width * r1.height;
        System.out.println("area = " + area1);

        System.out.println("r2: " + r2.width + " x " + r2.height);
        int area2 = r2.width * r2.height;
        System.out.println("area = " + area2);

        // grow both rectangles
        r1.width += 50;  r1.height += 10;
        r2.width += 5;   r2.height += 30;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Revised Client Program

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.x = 10;       r1.y = 20;
        r1.width = 100;  r1.height = 50;

        Rectangle r2 = new Rectangle();
        r2.x = 50;       r2.y = 100;
        r2.width = 20;   r2.height = 80;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("area = " + r1.area());

        System.out.println("r2: " + r2.width + " x " + r2.height);
        System.out.println("area = " + r2.area());

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Practice Defining Instance Methods

- Add a mutator method that moves the rectangle to the right by a specified amount.

```
public ______ moveRight(__________________) {


}
```

- Add an accessor method that determines if the rectangle is a square (true or false).

```
public __________ isSquare(____________) {




}
```

Defining a Constructor

- Our current client program has to use several lines to initialize each `Rectangle` object:

```
Rectangle r1 = new Rectangle();
r1.x = 10;       r1.y = 20;
r1.width = 100;  r1.height = 50;
```

- We'd like to be able to do something like this instead:

```
Rectangle r1 = new Rectangle(10, 20, 100, 50);
```

- To do so, we need to define a *constructor*, a special method that initializes the state of an object when it is created.

Defining a Constructor (cont.)

- Here it is:

```
public Rectangle(int initialX, int initialY,
  int initialWidth, int initialHeight) {
    this.x = initialX;
    this.y = initialY;
    this.width = initialWidth;
    this.height = initialHeight;
}
```

- General syntax for a constructor:

```
public <class name>(<parameter list>) {
    body of the constructor
}
```

- Note that a constructor has no return type.

Third Version of our `Rectangle` Class

```
public class Rectangle {
    int x;
    int y;
    int width;
    int height;

    public Rectangle(int initialX, int initialY,
      int initialWidth, int initialHeight) {
        this.x = initialX;
        this.y = initialY;
        this.width = initialWidth;
        this.height = initialHeight;
    }

    public void grow(int dWidth, int dHeight) {
        this.width += dWidth;
        this.height += dHeight;
    }

    public int area() {
        return this.width * this.height;
    }
}
```

Revised Client Program

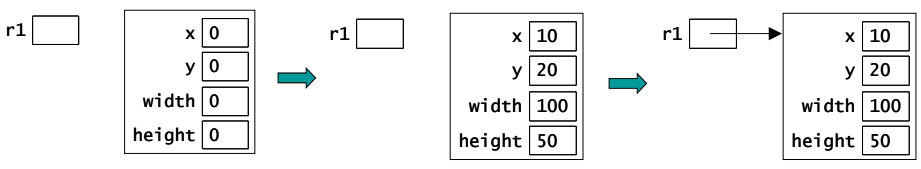
```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("area = " + r1.area());

        System.out.println("r2: " + r2.width + " x " + r2.height);
        System.out.println("area = " + r2.area());

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

A Closer Look at Creating an Object

- What happens when the following line is executed?

  `Rectangle r1 = new Rectangle(10, 20, 100, 50);`

- Several different things actually happen:
  1) a new `Rectangle` object is created
     - initially, all fields have their default values
  2) the constructor is then called to assign values to the fields
  3) a reference to the new object is stored in the variable r1



Limiting Access to Fields

- The current version of our `Rectangle` class allows clients to directly access a `Rectangle` object's fields:

```
r1.width = 100;
r1.height += 20;
```

- This means that clients can make inappropriate changes:

```
r1.width = -100;
```

- To prevent this, we can declare the fields to be *private*:

```
public class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;
    ...
}
```

- This indicates that these fields can only be accessed or modified by methods that are part of the `Rectangle` class.

Limiting Access to Fields (cont.)

- Now that the fields are private, our client program won't compile:

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("area = " + r1.area());

        System.out.println("r2: " + r2.width + " x " + r2.height);
        System.out.println("area = " + r2.area());

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Adding Accessor Methods for the Fields

```
public class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;
    ...
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
    public int getWidth() {
        return this.width;
    }
    public int getHeight() {
        return this.height;
    }
}
```

- These methods are *public*, which indicates that they can be used by code that is outside the `Rectangle` class.

Revised Client Program

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);

        System.out.println("r1: " + r1.getWidth() + " x " +
          r1.getHeight());
        System.out.println("area = " + r1.area());

        System.out.println("r2: " + r2.getWidth() + " x " +
          r2.getHeight());
        System.out.println("area = " + r2.area());

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.getWidth() + " x " +
          r1.getHeight());
        System.out.println("r2: " + r2.getWidth() + " x " +
          r2.getHeight());
    }
}
```

Access Modifiers

- `public` and `private` are known as *access modifiers*.
  - they specify where a class, field, or method can be used
- A class is usually declared to be `public`:

```
public class Rectangle {
```

  - indicates that objects of the class can be used anywhere, including in other classes
- Fields are usually declared to be `private`.
- Methods are usually declared to be `public`.
- We occasionally define `private` methods.
  - serve as *helper methods* for the `public` methods
  - cannot be invoked by code that is outside the class

Allowing Appropriate Changes

- To allow for appropriate changes to an object, we add whatever mutator methods make sense.
- These methods can prevent inappropriate changes:

```
public void setLocation(int newX, int newY) {
    if (newX < 0 || newY < 0) {
        throw new IllegalArgumentException();
    }

    this.x = newX;
    this.y = newY;
}
```

Allowing Appropriate Changes (cont.)

- Here are two other mutator methods:

```
public void setWidth(int newWidth) {
    if (newWidth <= 0) {
        throw new IllegalArgumentException();
    }
    this.width = newWidth;
}

public void setHeight(int newHeight) {
    if (newHeight <= 0) {
        throw new IllegalArgumentException();
    }
    this.height = newHeight;
}
```

Instance Methods Calling Other Instance Methods

- Here's another mutator method that we already had:

```
public void grow(int dWidth, int dHeight) {
    this.width += dWidth;
    this.height += dHeight;
}
```

- However, it doesn't prevent inappropriate changes.
- Rather than adding error-checking to it, we can have it call the new mutator methods:

```
public void grow(int dWidth, int dHeight) {
    this.setWidth(this.width + dWidth);
    this.setHeight(this.height + dHeight);
}
```

Revised Constructor

- To prevent invalid values in the fields of a `Rectangle` object, we also need to modify our constructor.
- Here again, we take advantage of the error-checking code that's already present in the mutator methods:

```
public Rectangle(int initialX, int initialY,
  int initialWidth, int initialHeight)
{
    this.setLocation(initialX, initialY);
    this.setWidth(initialWidth);
    this.setHeight(initialHeight);
}
```

  - `setLocation`, `setWidth`, and `setHeight` operate on the newly created `Rectangle` object

Encapsulation

- *Encapsulation* is one of the key principles of object-oriented programming.
  - another name for it is *information hiding*
- It refers to the practice of "hiding" the implementation of a class from users of the class.
  - prevent *direct* access to the internals of an object
    - making the fields private
  - provide *limited*, *indirect* access through a set of methods
    - making them public
- In addition to preventing inappropriate changes, encapsulation allows us to change the implementation of a class without breaking the client code that uses it.

Abstraction

- *Abstraction* involves focusing on the essential properties of something, rather than its inner or low-level details.
  - an important concept in computer science
- Encapsulation leads to abstraction.
  - example: rather than treating a `Rectangle` as four `int`s, we treat it as an object that's capable of growing itself, changing its location, etc.

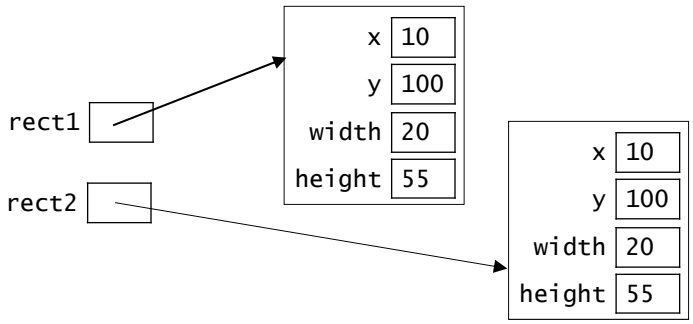
Practice Defining Instance Methods

- Add a mutator method that scales the dimensions of a `Rectangle` object by a specified factor.
  - make the factor a `double`, to allow for fractional values
  - take advantage of existing mutator methods
  - use a type cast to turn the result back into an integer
- Add an accessor method that gets the perimeter of a `Rectangle` object.

Testing for Equivalent Objects

- Let's say that we have two different `Rectangle` objects, both of which represent equivalent rectangles:

```
Rectangle rect1 = new Rectangle(10, 100, 20, 55);
Rectangle rect2 = new Rectangle(10, 100, 20, 55);
```



- What is the value of the following condition?

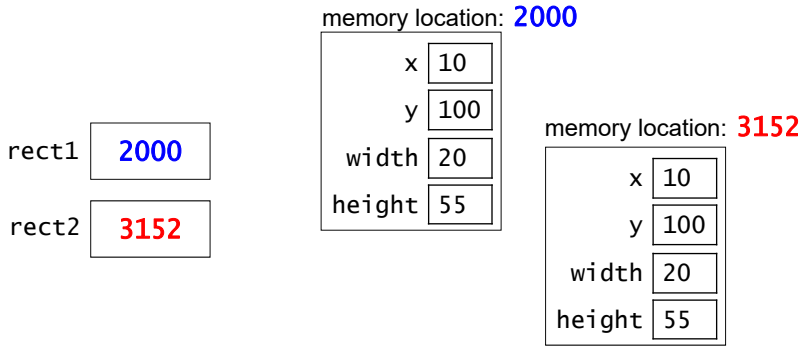
```
rect1 == rect2
```

Testing for Equivalent Objects (cont.)

- The condition

```
rect1 == rect2
```

compares the *references* stored in `rect1` and `rect2`.



- It doesn't compare the objects themselves.

Testing for Equivalent Objects (cont.)

- Recall: to test for equivalent objects, we need to use the `equals` method:

```
rect1.equals(rect2)
```

- Java's built-in classes have `equals` methods that:
  - return `true` if the two objects are equivalent to each other
  - return `false` otherwise

Default `equals()` Method

- If we don't write an `equals()` method for a class, objects of that class get a default version of this method.
- The default `equals()` just tests if the memory addresses of the two objects are the same.
  - the same as what == does!
- To ensure that we're able to test for equivalent objects, we need to write our own `equals()` method.

`equals()` Method for Our `Rectangle` Class

```
public boolean equals(Rectangle other) {
    if (other == null) {
        return false;
    } else if (this.x != other.x) {
        return false;
    } else if (this.y != other.y) {
        return false;
    } else if (this.width != other.width) {
        return false;
    } else if (this.height != other.height) {
        return false;
    } else {
        return true;
    }
}
```

- **Note:** The method is able to access the fields in `other` directly (without using accessor methods).
- Instance methods can access the private fields of *any* object from the same class as the method.

`equals()` Method for Our `Rectangle` Class (cont.)

- Here's an alternative version:

```
public boolean equals(Rectangle other) {
    return (other != null
         && this.x == other.x
         && this.y == other.y
         && this.width == other.width
         && this.height == other.height);
}
```

Converting an Object to a String

- The `toString()` method allows objects to be displayed in a human-readable format.
  - it returns a string representation of the object
- This method is called implicitly when you attempt to print an object or when you perform string concatenation:

```
Rectangle r1 = new Rectangle(10, 20, 100, 80);
System.out.println(r1);

// the second line above is equivalent to:
System.out.println(r1.toString());
```

- If we don't write a `toString()` method for a class, objects of that class get a default version of this method.
  - here again, it usually makes sense to write our own version

`toString()` Method for Our `Rectangle` Class

```
public String toString() {
    return this.width + " x " + this.height;
}
```

- Note: the method does not do any printing.
- It returns a `String` that can then be printed.

Revised Client Program

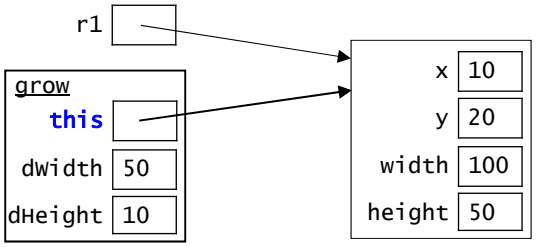
```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);

        System.out.println("r1: " + r1);
        System.out.println("area = " + r1.area());

        System.out.println("r2: " + r2);
        System.out.println("area = " + r2.area());

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1);
        System.out.println("r2: " + r2);
    }
}
```

Conventions for Accessors and Mutators

- Accessors:
  - usually have no parameters
    - all of the necessary info. is inside the called object
  - have a non-`void` return type
  - often have a name that begins with "get" or "is"
    - examples: `getWidth()`, `isSquare()`
    - but not always: `area()`, `perimeter()`
- Mutators:
  - usually have one or more parameter
  - usually have a `void` return type
  - often have a name that begins with "set"
    - examples: `setLocation()`, `setWidth()`
    - but not always: `grow()`, `scale()`

The Implicit Parameter and Method Frames

- When we call an instance method, the implicit parameter is included in its method frame.
  - example: `r1.grow(50, 10)`



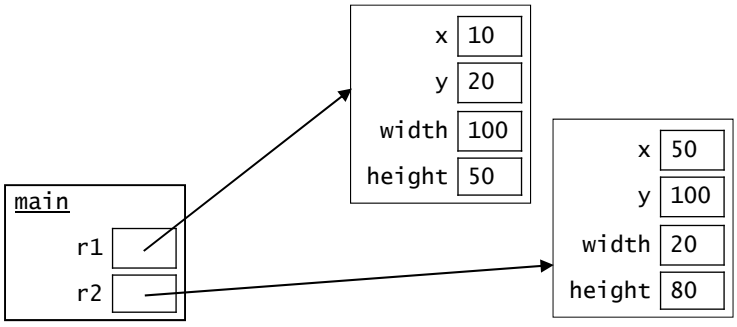
- The method uses `this` to access the fields in the called object.
  - even if the code doesn't explicitly use it

```
width += dWidth;          this.width += dWidth;
height += dHeight;        this.height += dHeight;
```

Example: Method Frames for Instance Methods

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

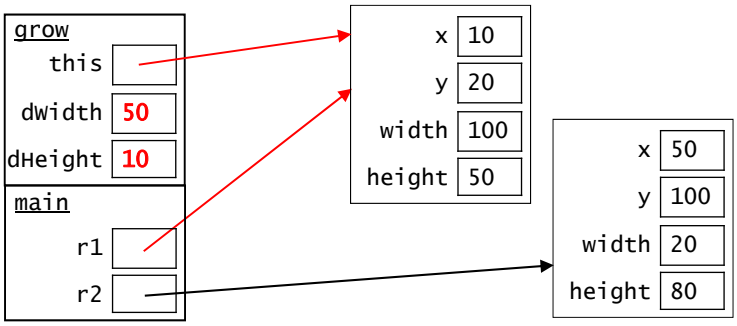
- After the objects are created:



Example: Method Frames for Instance Methods

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

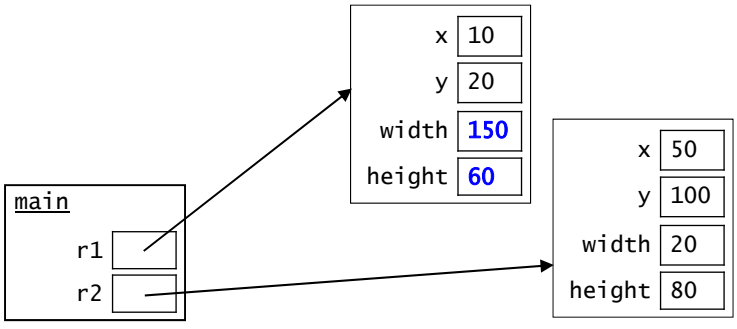
- During the method call `r1.grow(50, 10)`:



Example: Method Frames for Instance Methods

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

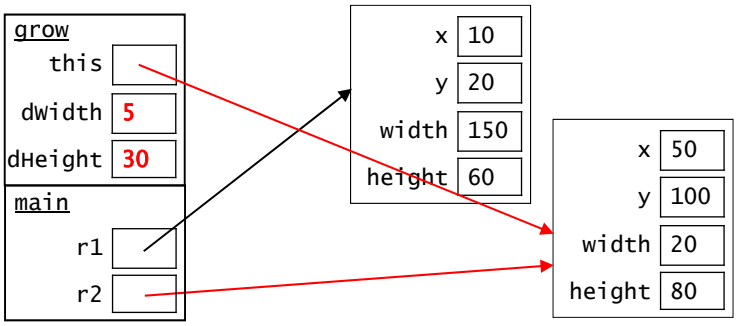
- After the method call `r1.grow(50, 10)`:



Example: Method Frames for Instance Methods

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

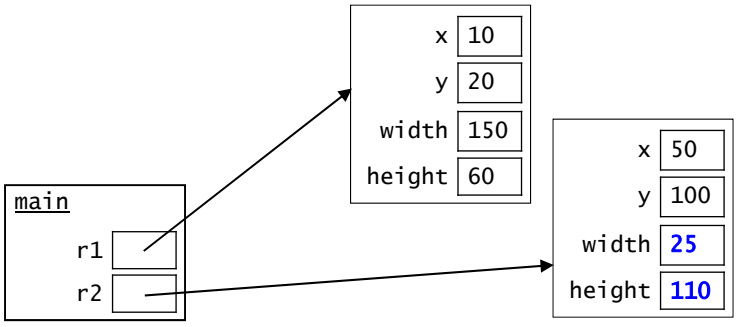
- During the method call `r2.grow(5, 30)`:



Example: Method Frames for Instance Methods

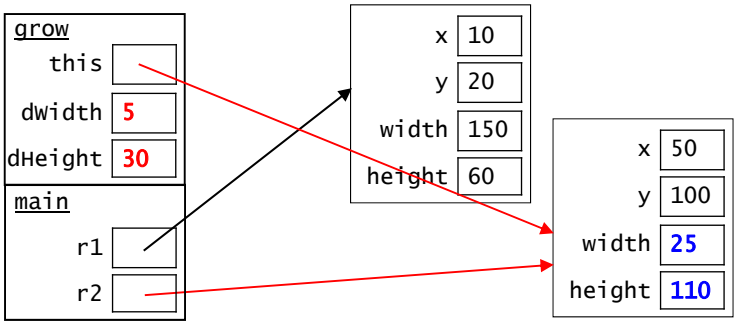
```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

- After the method call `r2.grow(5, 30)`:



Why Mutators Don't Need to Return Anything

- A mutator operates directly on the called object, so any changes it makes will be there after the method returns.
  - example: the call `r2.grow(5, 30)` from the last slide



  - during this call, `grow` gets a copy of the reference in `r2`, so it changes the object to which `r2` refers

Variable Scope: Static vs. Non-Static Methods

```
public class Foo {
    private int x;

    public static int bar(int b, int c, Foo f) {
        c = c + this.x;          // would not compile
        return 3*b + f.x;        // would compile
    }

    public int boo(int d, Foo f) {
        d = d + this.x + f.x;    // would compile
        return 2 * d;
    }
}
```

- Static methods (like `bar` above) do *NOT* have a called object, so they can't access its fields.
- Instance/non-static methods (like `boo` above) *do* have a called object, so they *can* access its fields.
- *Any* method of a class can access fields in an object of that class that is passed in as a parameter (like the parameter `f` above).

A Common Use of the Implicit Parameter

- Here's our `setLocation` method:

```
public void setLocation(int newX, int newY) {
    if (newX < 0 || newY < 0) {
        throw new IllegalArgumentException();
    }
    this.x = newX;
    this.y = newY;
}
```

- Here's an equivalent version:

```
public void setLocation(int x, int y) {
    if (x < 0 || y < 0) {
        throw new IllegalArgumentException();
    }
    this.x = x;
    this.y = y;
}
```

  - When the parameters have the same names as the fields, we *must* use `this` to access the fields.

Defining a Second Constructor

- Here's our `Rectangle` constructor:

```
public Rectangle(int initialX, int initialY,
  int initialWidth, int initialHeight) {
    this.setLocation(initialX, initialY);
    this.setWidth(initialWidth);
    this.setHeight(initialHeight);
}
```

- It requires four parameters:

```
Rectangle r1 = new Rectangle(10, 20, 100, 50);
```

- A class can have an arbitrary number of constructors, provided that each of them has a distinct parameter list.

Defining a Second Constructor (cont.)

- Here's a constructor that only takes values for width and height:

```
public Rectangle(int width, int height) {
    this.setWidth(width);
    this.setHeight(height);
    this.x = 0;
    this.y = 0;
}
```

  - it puts the rectangle at the location (0, 0)

- Equivalently, we can call the original constructor, and let it perform the actual assignments:

```
public Rectangle(int width, int height) {
    this(0, 0, width, height);  // call other constr.
}
```

  - we use the keyword `this` instead of `Rectangle`
  - this is the way that one constructor calls another

Practice Exercise: Writing Client Code

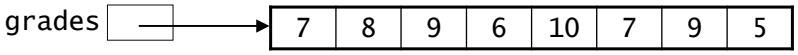
- Write a static method called `processRectangle()` that:
  - takes a `Rectangle` object (call it `r`) and an integer (call it `delta`) as parameters
  - prints the existing dimensions and area of the `Rectangle` (*hint:* take advantage of the `toString()` method)
  - increases both of the `Rectangle`'s dimensions by `delta`
  - prints the new dimensions and area

Collections of Data

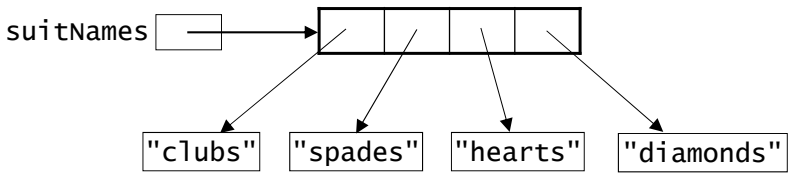
- There are many situations in which we need a program to maintain a collection of data.
- Examples include:
  - all of the grades on a given assignment/exam
  - a simple database of song info (e.g., in a music player)

Using an Array for a Collection

- We've used an array to maintain a collection of primitive data values.



- It's also possible to have an array of objects:



A Class for a Collection

- Rather than just using an array, it's often helpful to create a blueprint class for the collection.
- Example: a `GradeSet` class for a collection of grades from a single assignment or exam
  - possible field definitions:

```
public class GradeSet {
    private String name;
    private int possiblePoints;
    private double[] grades;
    private int gradeCount;
```

- The array of values is "inside" the collection object, along with other relevant information associated with the collection.
- In addition, we would add methods for maintaining and processing the collection.

A Blueprint Class for Grade Objects

- Rather than just representing the grades as `int`s or `double`s, we'll use a separate blueprint class for a single grade:

```
public class Grade {
    private double rawScore;
    private int latePenalty;  // as a percent
```

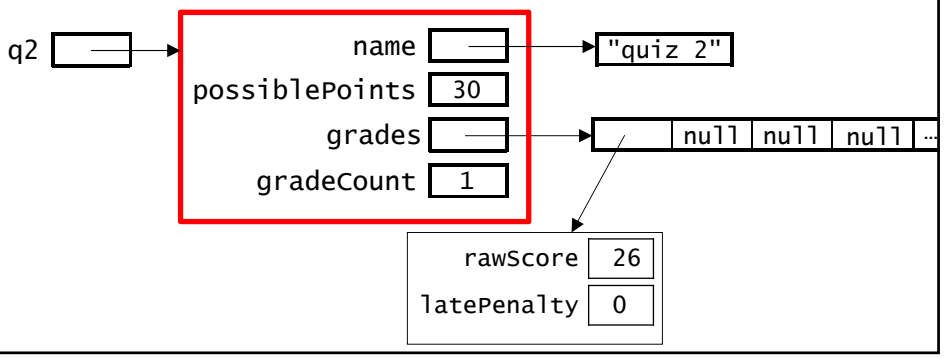
- This allows us to store both the raw score and the late penalty (if any).
- Constructors and methods include:

```
Grade(double raw, int late)
Grade(double raw)
getRawScore()
getLatePenalty()
setRawScore(double newScore)
setLatePenalty(int newPenalty)
getAdjustedScore()   // with late penalty
```

Revised GradeSet Class

```
public class GradeSet {
    private String name;
    private int possiblePoints;
    private Grade[] grades;
    private int gradeCount;
```

- Here's what one of these objects would look like in memory:



GradeSet Constructor/Methods

- Constructor:
  `GradeSet(String name, int possPts, int numGrades)`

- Accessor methods:
  ```
  String getName()
  int getPossiblePoints()
  int getGradeCount()
  Grade getGrade(int i) // get grade at position i
  double averageGrade(boolean includePenalty)
  ```

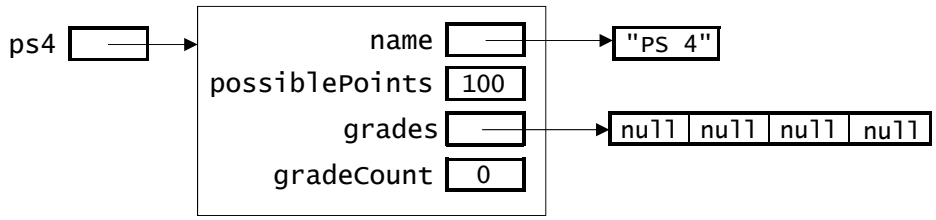
- Mutator methods:
  ```
  void setName(String name)
  void setPossiblePoints(int possPoints)
  void addGrade(Grade g)
  Grade removeGrade(int i) // remove grade at posn i
  ```

- Let's review the code for these, and write some of them together.

`GradeSet` Constructor/Methods

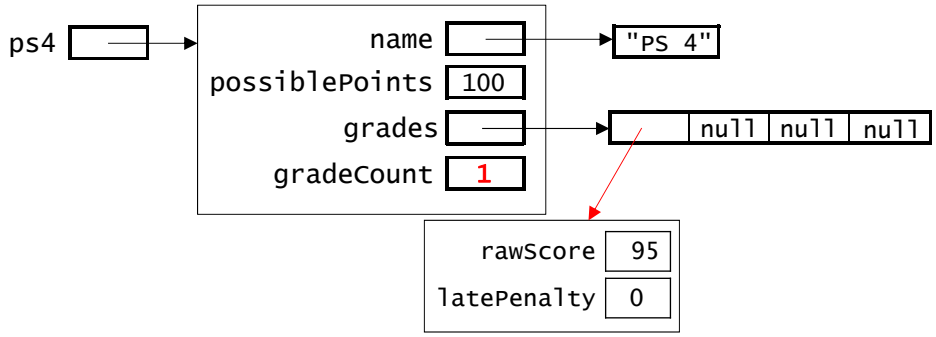
`GradeSet` Constructor/Methods

GradeSet: Adding a Grade



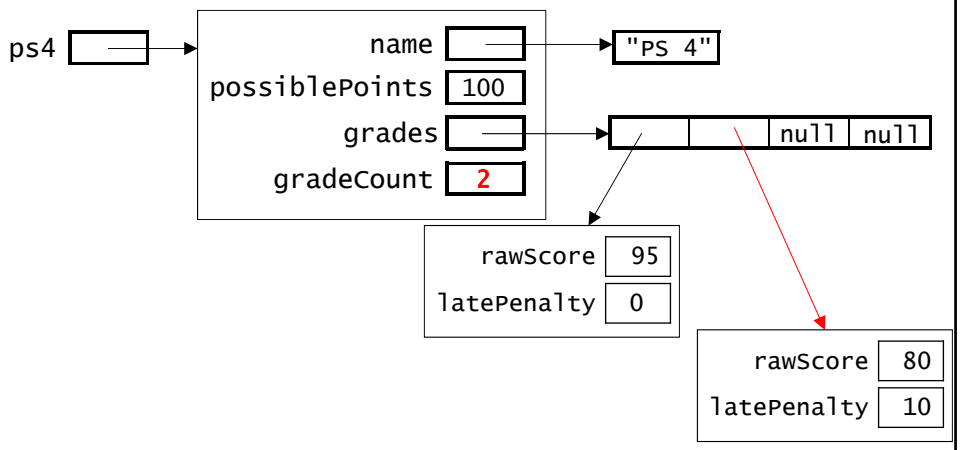

```
GradeSet ps4 = new GradeSet("PS 4", 100, 4);
ps4.addGrade(new Grade(95, 0));
ps4.addGrade(new Grade(80, 10));
```

GradeSet: Adding a Grade




```
GradeSet ps4 = new GradeSet("PS 4", 100, 4);
ps4.addGrade(new Grade(95, 0));
ps4.addGrade(new Grade(80, 10));
```

GradeSet: Adding a Grade




```
GradeSet ps4 = new GradeSet("PS 4", 100, 4);
ps4.addGrade(new Grade(95, 0));
ps4.addGrade(new Grade(80, 10));
```