public class Student
{
    private String name;
    private int age;

    public void setName(String studentName) {
        name = studentName;
    }
    public void setAge(int studentAge) {
        age = studentAge;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}
return Statement

• A method that returns a value must have at least one return statement
• Terminates the method, and returns a value
• Syntax:
  – return Expression;
• Expression can be any expression that produces a value of type specified by the return type in the method heading
Methods

- **getMajor()**
  ```java
  public String getMajor()
  {
    return major;
  }
  ```

  **returns a String**

- **increaseYear()**
  ```java
  public void increaseYear()
  {
    classYear++;
  }
  ```

  **returns nothing**
Methods that Return a Value

As usual, inside a block (defined by braces), you can have multiple statements

```java
public String getClassYear()
{
    if (classYear == 1)
        return "Freshman";
    else if (classYear == 2)
        return "Sophomore";
    else if ...
}
```
return Statement

• Can also be used in methods that return nothing
• Simply terminates the method
• Syntax:
  – return;

```java
public void increaseYear()
{
    if (classYear >= 4)
        return;
    classYear++;
}
```
Methods with Parameters

• Parameters are used to hold the value that you pass to the method

• Parameters can be used as (local) variables inside the method

```java
public int square(int number) {
    return number * number;
}
```

Parameters go inside the parentheses of method header
public static void main(String[] args)
{
    Student jack = new Student();
    jack.setName("Jack Smith");
    jack.setClassYear(3);
}

Parameters/Arguments
Methods with Multiple Parameters

• Multiple parameters separated by commas
  
  ```java
  public double getTotal(double price, double tax)
  {
      return price + price * tax;
  }
  ```

• When calling a method, the order, type, and number of arguments must match parameters specified in method heading
Today’s Topics

• Constructors
• Overloading methods
• Static variables and methods
Constructors

• Create and initialize new objects
• Special methods that are called when (and **only** when) creating a new object

```
Student jack = new Student();
```

Calling a constructor
Creating an Object

Create an object *jack* of class *Student*

\[
\text{Student jack} = \text{new Student}();
\]

Create an object *keyboard* of class *Scanner*

\[
\text{Scanner keyboard} = \text{new Scanner(System.in)};
\]

Create an object *keyboard* of class *Scanner*
Constructors

• Can perform any action you write into a constructor’s definition
  – There are no specific rules about what’s in a constructor
• Meant to perform initializing actions
  – Usually, initializing values of instance variables by the creator of the object
Similar to Setter Methods

• However, constructors *create* an object in addition to setting the values of instance variables
• Like methods, constructors can have parameters
public class Pet
{
    private String name;
    private int age;
    private double weight;

    public Pet()
    {
        name = "No name yet.";
        age = 0;
        weight = 0;
    }

    public static void main(String[] args)
    {
        Pet p = new Pet();
    }
}
public class Pet {
    private String name = "No name yet.";
    private int age = 0;
    private double weight = 0;

    public static void main(String[] args) {
        Pet p = new Pet();
    }
}
Default Constructor

• Constructor that takes no parameters

```java
public Pet()
{
    name = "No name yet."
    age = 0;
    weight = 0;
}
```

• Java automatically defines a default constructor if you do not define any constructors
  – You’ve never written a constructor but you can still create objects
Constructors with Parameters

```java
public class Pet {
    private String name;
    private int age;
    private double weight;

    public Pet(String initName, int initAge, double initWeight) {
        name = initName;
        age = initAge;
        weight = initWeight;
    }

    public void setPet(String newName, int newAge, double newWeight) {
        name = newName;
        age = newAge;
        weight = newWeight;
    }
}
```

Another version of constructor that has parameters
public Pet(String initName, int initAge, double initWeight) {
    name = initName;
    age = initAge;
    weight = initWeight;
}
Constructors with Parameters

• If you define at least one constructor, a default constructor will not be created for you

• Now you must create a Pet object like this:
  – Pet odie = new Pet(“Odie”, 3, 8.5);
  – Pet odie = new Pet(); // WRONG! No default constructors!

```java
public class Pet {
    private String name;
    private int age;
    private double weight;
    public Pet(String initName, int initAge, double initWeight)
    {
        name = initName; age = initAge; weight = initWeight;
    }
}
```
Multiple Constructors

• You can have several constructors per class
  – They all have the same name, just different parameters
    • Remember that the name is the same as the class name
  – The methods (with the same name) will be called according to its parameters
public class Pet {
    private String name;
    private int age;
    private double weight;

    public Pet() {
        name = "No name yet.";
        age = 0;
        weight = 0;
    }

    public Pet(String initName, int initAge, double initWeight) {
        name = initName;
        age = initAge;
        weight = initWeight;
    }

    public static void main(String[] args) {
        Pet p = new Pet();
        Pet q = new Pet("Garfield", 3, 10);
    }
}
Multiple Constructors

```java
public class Pet {
    private String name = "No name yet."
    private int age = 0;
    private double weight = 1; // The instance variables are initialized

    public Pet() {
        name = "No name yet."
        age = 0;
        weight = 0;
    }

    public Pet(String initName, int initAge, double initWeight) {
        name = initName;
        age = initAge;
        weight = initWeight;
    }

    public Pet(String initName) {
        name = initName;
    }

    public static void main(String[] args) {
        Pet p = new Pet(); // p.weight is 0 - it is overwritten by constructor
        Pet q = new Pet("Garfield", 3, 10);
        Pet w = new Pet("Odie"); // w.weight is 1, as only one constructor can be called. Variables will get initial value if not set in constructor.
    }
}
```
Calling a Constructor

• A constructor can be only called once when the object is created
  – Pet odie = new Pet("Odie", 3, 8.5);
• You can not invoke a constructor from an object
  – odie.Pet("Odie", 3, 8.5);
    // Wrong! A constructor can not be invoked this way
  – odie.setPet("Odie", 3, 8.5);
    // Yes. You can use a setter instead
public class Pet
{
    private String name;
    private int age;
    private double weight;

    public Pet(String initName, int initAge, double initWeight)
    {
        setPet(initName, initAge, initWeight);
    }

    public void setPet(String newName, int newAge, double newWeight)
    {
        name = newName;
        age = newAge;
        weight = newWeight;
    }
}
Initializing and Setting Instance Variables

• Initialization values give values to instance variables that are the same (or commonly the same) for all objects
• Constructors give values to instance variables that should be decided for each object
• Setters give values to instance variables that can be changed during time
  – If a value is never going to be changed, no setter is needed
public class Pet {
    private String name;
    private int age = 0;
    // Age is always 0 (assuming newly-born pets are registered immediately)
    private double weight;

    public Pet(String initName, double initWeight){
        name = initName;
        weight = initWeight;
        // Name is given when registering, and can not be changed
    }

    public void setPetWeight(double newWeight) {
        weight = newWeight;
        // Weight changes every time you weight your pet
    }

    public void setPetAge(double newAge) {
        age = newAge;
        // Surely age can change, too
    }
}
Summary: Constructor

• A special method with the same name as the class, and no return type
• Called only when an object is created
• It can take parameters to initialize instance variables
• You can define multiple constructors with different parameter lists
Methods Overloading

- We’ve seen that a class can have multiple constructors. Notice that they have the same name

```java
public class Pet {
    public Pet() {...}
    public Pet(String initName, int initAge, double initWeight) {...}
    public Pet(String initName) {...}
    public static void main(String[] args) {
        Pet p = new Pet(); // First constructor will be called
        Pet q = new Pet("Garfield", 3, 10); // Second constructor
        Pet w = new Pet("Odie"); // Third constructor
        Pet u = new Pet("Nermal", 2); // Wrong - no matching method
    }
}```
Overloading

- Using the same method name for two or more methods within the same class
  - It’s not only for constructors
- Parameter lists must be different
  - public double average(int n1, int n2)
  - public double average(double n1, double n2)
  - public double average(double n1, double n2, double n3)
- Java knows what to use based on the number and types of the arguments
Overloading

• Java knows what to use based on the number and types of the arguments
  – You’ve used overloading before
    • System.out.println(“The result is”); // String type parameter
    • System.out.println(20); // int type parameter

• Java makes the decision based on a method’s signature
Method Signature

• The signature includes a **method’s name** and the **number and types** of its **parameters**
  
  – Pet q = new Pet("Garfield", 3, 10);
  – Pet w = new Pet("Odie");

• Signature does NOT include return type
  
  – Cannot have two methods with the same signature in the same class
  
  – **public double average(int n1, int n2)**
  
  – **public int average(int n1, int n2)** // **Wrong overloading**

  – Java won’t know what method to call if average(1,2) is invoked
Overloading and Type Conversion

- Java always tries to find an exactly matching method. If it fails, it tries type conversion.
  - If a class has the following two methods:
    - `public double average(int n1, int n2)`
    - `public double average(double n1, double n2)`
      - If the method call is `average(3,3)`, the first method will be called.
  - However, if a class only have this method:
    - `public double average(double n1, double n2)`
      - If the method call is `average(3,3)`, it will be converted to `average(3.0,3.0)` and call the (only) method.
  - Recall: `byte->short->int->long->float->double`
How to Use Overloading

• Use it only if two or more methods are performing exactly the same function
  – public void setPet(String newName)
  – public void setPet(String newName, int newAge, double newWeight)

• It is a very bad idea to create methods that have the same name but do different things
  – public void setPet(int newAge)
  – public void setPet(double newWeight)
  – What happens if we call setPet(3)? What about setPet(3.0)?
    • Use setAge() and setWeight() instead
    • Usually we do not overload methods if parameters can be converted
Summary: Overloading

• Overloading means several methods share the same name but have different parameters.

• Java calls the methods according to the parameter numbers and types:
  – The name, parameter number and parameter type form the method signature.

• Make sure that they do the same thing. Otherwise the user will be confused.
Static Variables and Methods

• Instance variables

private int age;
private String name;

• Methods

public int getAge()
{
    return age;
}

• Calling methods on objects

Student std = new Student();
std.setAge(20);
System.out.println(std.getAge());
Static Variables and Methods

• Recall that “classes do not have data; individual objects have data”

• This is not always true – classes can have data, too
  – static variables and methods belong to a class as a whole, not to an individual object
  – When would you want a method that does not need an object?
    • If the method perform a general function instead of actions on an object
// Returns x raised to the yth power, where y >= 0.
public int pow(int x, int y) {
    int result = 1;
    for (int i = 0; i < y; i++) {
        result *= x;
    }
    return result;
}

Do we need an object to call this method?
Static Variables and Methods

• We have seen static variables and methods before
  – private static final int FACE_DIAMETER = 200;
    • Recall that “final” means “not changable”
  – public static void main(String[] args)
    – Static can describe more than constants and main method
      • Static variables are sometimes referred as “global variables”, which record the global status of all objects in the same class
      • Static methods are used for actions that do not relate to a certain object
        – main method is a static method because if you execute a program, this entrance is not owned by an object
Instance vs. Static

• Instance variables and methods
  – private int name;
  – public void setName(String newName){}

• Static variables and methods
  – private static int totalNumber;
  – public static int getTotalNumber(){}
Instance vs. Static

• In an instance method
  – Instance variables/methods can be called
  – Static variables/methods can also be called
    • Eg: you can call a static method `pow(x, y)` anywhere in a class

• In a static method
  – Only static variables/methods can be called
  – Instance variables/methods can be only called if they are invoked from an object
    • Instance variables include “this”
Invoking Instance and Static Methods

• From an object, both instance and static variables/methods can be invoked
  – `ObjectName.var;`
  – However, static variables/methods keep the same for the same type objects

• From a class, only static variables/methods can be invoked
  – `ClassName.var;`
  – You are suggested to call static variables/methods this way
Example: Static Variables and Methods

```java
public class Pet {
    private String name;
    private static int totalNumber = 0;
    // totalNumber is initialized when the first object is created

    public Pet(String initName) {
        this.name = initName;
        // Recommended: use "this" to call instance variables
        totalNumber++; // totalNumber can be accessed in an instance method
        System.out.println("Total pet number is " + Pet.getTotalNumber());
        // Recommended: use class name to call static variables
    }

    public static int getTotalNumber() {
        return totalNumber;
        // You can not access "name" or "this" in a static method
    }

    public static void main(String[] args) {
        Pet a = new Pet("Odie");
        Pet b = new Pet("Garfield");
        Pet c = new Pet("Nermal");
        // Three objects are created, so totalNumber is increased for three times
        System.out.println("Total pet number is " + a.getTotalNumber());
        System.out.println("Total pet number is " + b.getTotalNumber());
        // You can invoke a static method from an object. However they perform the same.
        // You are recommended to call it as Pet.getTotalNumber();
    }
}
```
Example: The Output

- Total pet number is 1
- Total pet number is 2
- Total pet number is 3
- Total pet number is 3
- Total pet number is 3
Summary: Static Variables/Methods

- Static variables and methods belong to a class instead of an object.
- Every object has its own instance variables; all objects in the same type share the same static variables.
- Pay attention to: what can be accessed in different methods.