Outline

- Classes and objects
- Methods and constructors
  - definition and passing
  - method **overloading**
- Class variables, constants and methods
- Inheritance
  - inheritance hierarchy
  - method **overriding**
A **class** is a collection of **data** and **methods** that operate on that data.

An example class: **Circle**

```java
public class Circle {
    private double radius = 1.0;

    // Method that returns the area of the circle
    public double area() {
        return radius * radius * 3.14159;
    }
}
```
Objects are instances of a class

- To declare a variable of Circle type:
  - `Circle c;`
- This variable c is simply a name that refers to a circle object; it is not an object itself.
- To create an instance of our Circle class, and have it assigned to the variable c:
  - `c = new Circle();`

Access object data

- After we have created an object, we can use its data fields. The syntax is similar with C++.
  - `Circle c = new Circle();`
  - `/initialize our circle to have radius 2.0.
  - `c.radius = 2.0;`
Using objects methods

- To access the methods of an objects:
  - `Circle c = new Circle();`
  - `double a;`
  - `c.radius = 2.5;`
  - `a = c.area();`

Differences between variables of primitive data types and object types

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive type</td>
<td>int i = 1</td>
<td>i = 1</td>
</tr>
<tr>
<td>Object type</td>
<td>Circle c</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reference</td>
</tr>
</tbody>
</table>

Created using `new Circle()`
Copying Variables of Primitive Data Types and Object Types

**Primitive type assignment**

\[ i \leftarrow j \]

**Object type assignment**

\[ c_1 \leftarrow c_2 \]

---

**Garbage Collection**

As shown in the previous figure, after the assignment statement \( c_1 = c_2 \), \( c_1 \) points to the same object referenced by \( c_2 \). The object previously referenced by \( c_1 \) is no longer useful. This object is known as garbage. Garbage is automatically collected by JVM.
Introducing Methods

A method is a collection of statements that are grouped together to perform an operation.

Introducing Methods, cont.

- **parameter profile** refers to the type, order, and number of the parameters of a method.

- **method signature** is the combination of the method name and the parameter profiles.

- The parameters defined in the method header are known as **formal parameters**.

- When a method is invoked, its formal parameters are replaced by variables or data, which are referred to as **actual parameters**.
Ambiguous Invocation

Sometimes there may be two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*. Ambiguous invocation is a compilation error.

```java
public class AmbiguousOverloading {
    public static void main(String[] args) {
        System.out.println(max(1, 2));
    }
    public static double max(int num1, double num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
    public static double max(double num1, int num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
}
```
Constructors

```java
Circle(double r) {
    radius = r;
}

Circle() {
    radius = 1.0;
}
```

Constructors are a special kind of methods that are invoked to construct objects.

```java
myCircle = new Circle(5.0);
```

Constructors, cont.

A constructor with no parameters is referred to as a default constructor.

Constructors must have the same name as the class itself. Constructors do not have a return type—not even void. Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.
**Defining a constructor**

```java
public class Circle {
    public double x, y, r;
    public Circle(double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
    public double circumference() {
        return r * 2 * 3.14159;
    }
    public double area() {
        return r * r * 3.14159;
    }
}
```

**Defining a constructor, cont.**

- With the default constructor, we had to write:
  - `Circle c = new Circle();`
  - `c.x = 1.414;`
  - `c.y = -1.0;`
  - `c.r = .25;`
- With the new constructor, the initialization becomes part of the object creation step:
  - `Circle c = new Circle(1.414, -1.0, .25);`
The *this* keyword

- *this* is a reference to the **current calling object** and may be used as an object variable (you may not declare it).

---

**Multiple constructors**

- Sometimes you want to be able to initialize an object in a number of different ways.
- No problem: A class can have any number of constructor methods.
- Example in next slide.
public class Circle {
    public double x, y, r;
    public Circle(double x, double y, double r)
        {this.x = x; this.y = y; this.r = r;}
    public Circle(double r)
        {this.x = 0.0; this.y = 0.0; this.r = r;}
    public Circle(Circle c)
        {this.x = c.x; this.y = c.y; this.r = c.r;}
    public Circle()
        {this.x = 0.0; this.y = 0.0; this.r = 1.0;}
    public double circumference(){return r * 2 * 3.14159; }
    public double area() {return r * r * 3.14159; }
}

---

Method overloading

- In this example, all the constructor methods have the same name!
- In Java, a method is distinguished by its name, and by the number, type and position of the arguments.
- When you call a method and there is more than one method with the same name, the compiler automatically picks the one that matches.
Visibility Modifiers

By default, the class, variable, or data can be accessed by any class in the same package.

public
The class, data, or method is visible to any class in any package.

private
The data or methods can be accessed only by the declaring class.
The get and set methods are used to read and modify private properties.

Visibility Modifiers Cont.

The instance variables of a class should normally be declared private, and the class methods should be used to provide a standard interface to the class.

Use set methods to check the validity and consistency of input data before it is stored in an object’s instance variable.

get methods are used to retrieve information from the instance variables and to format it properly for presentation to the outside world.
Passing Objects to Methods

- **Passing by value** (the value is the reference to the object)

Passing Objects to Methods, cont.

```
main method

n
5

printAreas method
times

5

\( c \)

myCircle
Reference

\( myCircle: Circle \)

radius = 1

\( c \)

Pass by value (here the value is 5)

Pass by value (here the value is the reference for the object)
```
Class variables

- Sometimes, we want a variable of which there is only one copy -- something like a global variable in C.
- The problem is Java does not allow global variables! Every variable in Java must be declared inside a class.
- Java uses the static keyword to indicate that a particular variable is a class variable rather than an instance variable.

Class Variables, Constants, and Methods

Class variables are shared by all the instances of the class.

Class methods are not tied to a specific object.

Class constants are final variables shared by all the instances of the class.
Scope of Variables

- The scope of instance and class variables is the entire class. They can be declared anywhere inside a class.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.

```java
public class Circle {
    static int num_circles = 0;
    public double x, y, r;
    public Circle(double x, double y, double r) {
        this.x = x; this.y = y; this.r = r;
        num_circles++;
    }
    public Circle(double r) {
        this.x = 0.0; this.y = 0.0; this.r = r;
    }
    public Circle(Circle c) {
        this.x = c.x; this.y = c.y; this.r = c.r;
    }
    public Circle() {
        this.x = 0.0; this.y = 0.0; this.r = 1.0;
    }
    public double circumference() { return r * 2 * 3.14159; }
    public double area() { return r * r * 3.14159; }
}
```
**Accessing class variables**

- Because *static* variables are associated with the class rather than with an instance, we access them through the *class* rather than through the instance. For example:

  ```java
  System.out.println("Number of circles created: "+
          Circle.num_circles);
  ```

---

**Another class variable example**

- When computing the area and circumference of circles, we use the value *PI*. Since we use the value frequently, we do not want to keep typing out **3.14159**, so we define it as a class variable.

  ```java
  public class Circle {
      public static final double PI=3.1415926535897932;
      public double x, y, r;
      // ... etc ...
  }
  ```

  We use *final* keyword, which means that this variable can never be changed.
The final keyword, prevents you from doing anything stupid like:

- `Circle.PI = 4;`

The Java compiler is smart about variables declared static and final -- it knows they have constant values. So when you have:

- `double circumference = 2 * Circle.PI * radius;`

The compiler precomputes the value `2 * Circle.PI`.

Java does not have a preprocessor with a C-style `#define` directive. The static final variables are Java’s substitute for C’s `#define` constants.

---

A class method for circles

Class methods are the closest Java comes to “global” methods.

```java
public class Circle {
    public double x, y, r;
    // an instance method. Returns the bigger of two circles.
    public Circle bigger(Circle c){
        if(c.r > r) return c; else return this;
    }
    // an class method. Returns the bigger of two circles.
    public static Circle bigger(Circle a, Circle b){
        if(a.r > b.r) return a; else return b;
    }
    // other methods omitted here.
}
```
A class method for circles, cont.

- You would invoke the instance method like:
  ```java
  Circle a = new Circle(2.0);
  Circle b = new Circle(3.0);
  Circle c = a.bigger(b);
  ```
- And you would invoke the class method like:
  ```java
  Circle a = new Circle(2.0);
  Circle b = new Circle(3.0);
  Circle c = Circle.bigger(a, b);
  ```

Inheritance

- The inheritance hierarchy is like a family tree where the classes of the top (the base classes or superclasses) represent general data abstractions and the classes of the bottom (the derived classes or subclasses) represent specializations of the base classes.
- The derived classes inherit the attributes or behaviors of their parent classes, and they can override what they inherit with their own attributes and behaviors.
Example

- We want to be able to manipulate circles and draw them on the screen.

```java
public class GraphicCircle extends Circle {
    Color outline, fill;
    public void draw(DrawWindow dw){
        dw.drawCircle(x, y, r, outline, fill);
    }
}
```

- The `extends` keyword tells Java that GraphicCircle is a subclass of Circle.

Inheritance

- 1 of the fundamental principles of OOP
  - allows code reuse
- Models the **IS-A** relationship
  - a student is-a person
  - an undergraduate is-a student
  - a rectangle is-a shape
- Contrast with the **Has-A** relationship
  - a student has-a name
- Is-a relationships indicate inheritance, has-a relationships indicate composition (fields)
Inheriting from a Class

- The `extends` keyword is used to specify which preexisting class a new class is inheriting from.
  ```java
  public class Student extends Person
  ```
- Person is said to be
  - the parent class of Student
  - the super class of Student
  - the base class of Student
  - an ancestor of Student
- Student is said to be
  - a child class of Person
  - a sub class of Person
  - a derived class of Person
  - a descendant of Person

Inheriting from a Class

- If a class header does not include the `extends` clause the class extends the `Object` class by default.
  - `Object` is an ancestor to all classes
  - it is the only class that does not extend some other class
- A class extends exactly one other class
  - extending two or more classes is multiple inheritance. Java does not support this directly, rather it uses `Interfaces`. 
Implications of Inheritance

- The sub class gains all of the behavior (methods) and data regarding state (instance variables) of the super class and all ancestor classes.
- Sub classes can:
  - add new fields
  - add new methods
  - override existing methods (change behavior)
- Sub classes may not:
  - remove fields
  - remove methods
- Note, even though an object may have instance variables from its parent they may not be accessible by the code of the child class if the fields are private.

The Object Class

- All classes inherit the Object class either directly or indirectly (via a chain of inheritance).
- Minimal class, but it does contain the toString, equals and getClass methods.
  - implication is every class has a toString and equals method (Even if the writer of a class did not provide one.)
  - normally classes will override these methods to give them a more meaningful behavior than the ones Object provides.
Access Modifiers and Inheritance

- **public**
  - accessible to all classes
- **private**
  - accessible only within that class. Hidden from all sub classes.
- **protected**
  - accessible by classes within the same package and all descendant classes
- Instance variables *should* be private
- protected methods are used to allow descendant classes to modify instance variables in ways other classes can't

Shape Classes

- Declare a class called `ClosedShape`
  - assume all shapes have x and y coordinates
  - override Object's version of `toString`
- Possible sub classes of `ClosedShape`
  - Rectangle
  - Circle
  - Ellipse
  - Square
- Possible hierarchy
  - `ClosedShape -> Rectangle -> Square`
A ClosedShape class

```java
public class ClosedShape {
    private double dMyX;
    private double dMyY;

    public ClosedShape() {
        this(0, 0);
    }

    public ClosedShape (double x, double y) {
        dMyX = x;
        dMyY = y;
    }

    public String toString() {
        return "x: " + dMyX + " y: " + dMyY;
    }
}
// Other methods not shown
```

Overriding methods

- any method that is not final may be overridden in a descendant class
- same signature as method in ancestor
- may not reduce visibility
- may use the original method if simply want to add more behavior to existing
- The Rectangle class
  - adds data, overrides `toString`
A Rectangle Class

```java
public class Rectangle extends ClosedShape {
    private double dMyWidth;
    private double dMyHeight;

    public Rectangle() {
        this(0, 0);
    }

    public Rectangle(double width, double height) {
        dMyWidth = width;
        dMyHeight = height;
    }

    public String toString() {
        return super.toString() + " width " + dMyWidth
                      + " height " + dMyHeight;
    }
}
// other methods not shown
```

Constructors

- Constructors handle initialization of objects
- When creating an object with one or more ancestors (every type except Object) a chain of constructor calls takes place
- The reserved word `super` may be used in a constructor to call one of the parent’s constructors
  - must be first line of constructor
- If no parent constructor is explicitly called the default, 0 parameter constructor of the parent is called
  - if no default constructor exists a syntax error results
- If a parent constructor is called another constructor in the same class may no be called
  - no `super();this();` allowed. One or the other, not both
  - good place for an initialization method
Constructors chaining

- If the first line of a constructor is a call to another constructor in the same class using `this()` syntax, Java does not implicitly call the superclass constructor.
- But note that if the constructor invoked with the `this()` syntax does not invoke `super()` explicitly, Java does invoke `super()` implicitly. So while constructor methods within a class may invoke each other, eventually one of them must invoke the superclass constructor method.
- What this means is that constructor calls are chained -- any time an object is created, a sequence of constructor methods are invoked, from subclass to superclass on up to `Object` at the root of the class hierarchy.

Another Rectangle Constructor

```java
public class Rectangle extends ClosedShape {
    private double dMyWidth;
    private double dMyHeight;

    public Rectangle(double width, double height, double x, double y)
    { super(x,y);
        // calls the 2 double constructor in
        // closed shape
        dMyWidth = width;
        dMyHeight = height;
    }
    // other methods not shown
}
```
Initialization method

```java
public class Rectangle {
    private double dMyWidth;
    private double dMyHeight;

    public Rectangle() {
        init(0, 0);
    }

    public Rectangle(double width, double height) {
        init(width, height);
    }

    public Rectangle(double width, double height, double x, double y) {
        super(x, y);
        init(width, height);
    }

    private void init(double width, double height) {
        dMyWidth = width;
        dMyHeight = height;
    }
}
```

The Keyword `super`

- **super** is used to access something (any protected or public field or method) from the super class that has been overridden
- Rectangle’s `toString` makes use of the `toString` in ClosedShape by calling `super.toString()`
- without the super calling `toString` would result in infinite recursive calls
- Java does not allow nested supers `super.super.toString()` results in a syntax error even though technically this refers to a valid method, Object’s `toString`