

Objects: Data Abstraction

- In Object-Oriented programming languages like Java, *objects* are used to represent data
- A **class** defines a **type of object**, including
 - its data
 - its permissible operations
- Once a type is defined, objects of that type can be declared and used

Example: Planet

- Suppose that we want to represent planets in Java
- We can define a class called Planet
 - Data: diameter, mass, orbit, orbital period, location at a given time, ...
 - Methods: `setDiameter()`, `setMass()`, etc., `getDiameter()`, `getMass()`, etc.

String: Using a Standard Class

- Different objects have different methods for manipulating their data
- The specific methods are determined based on what makes sense for that type of object
- For Strings: length, concatenation, comparison, substring, substring comparison, ...

Example: Palindromes

- Two String methods:
 - length() – returns the length of the string
 - charAt() – returns the character at a given position in the string
- Palindrome – a word that reads the same forward or backward
 - Examples: eye, madam, radar, ...

Algorithm

1. Get a word from the user
2. Compare the first and last characters
 1. If they are different, return false
 2. Otherwise, repeat with the second and second to the last, etc.
3. If the characters all match, return true

Algorithm 2

- Set *left* to the index of the first (leftmost) character
- Set *right* to index the last (rightmost) character
- While *left* is less than *right*
 - Compare the *left* character with the *right* character
 - If they are not equal return false
 - Increment *left*
 - Decrement *right*
- Return true

```
public class Palindrome {
    public static void main(String[] args) {
        String str = Console.in.readWord();
        System.out.println(str + " " +isPalindrome(str));
    }
    static boolean isPalindrome(String s) {
        int left =0, right =s.length()-1;
        while (left < right) {
            if ( s.charAt(left) != s.charAt(right) )
                return false;
            left++;
            right--;
        }
        return true;
    }
}
```

Methods

- Each **type** of object supports a specified set of methods
- The methods are called for a specific object and have direct access to that object's data without having to pass the object as a parameter

```
String s;
```

```
s.length();
```

String Methods

- `boolean equals(Object anObject)`
 - Compares this string with another object
- `int length()`
 - Number of characters in this string
- `char charAt(int index)`
 - Returns the character at the position index withi
this string

String Methods II

- `int compareTo(String str)`
 - Returns an integer value, based on lexicographic order
- `int indexOf(int ch)`
 - Index of where the `ch` occurs in this string or -1 if not present
- `int indexOf(String str)`
 - Index of the first character of a matching substring `str`
- `String concat(String str)`
 - Concatenates this string instance with `str` and returns the result

String Methods III

- String toLowerCase()
 - Returns a copy of this string but in all lowercase
- String toUpperCase()
 - Returns a copy of this string but in all uppercase
- static String valueOf(*type* prim)
 - Returns the String representation of primitive value *prim*, where *type* can be any primitive

```
public class StringTest {  
    public static void main(String [ ] args){  
        String str1 ="aBcD",str2 ="abcd",str3;  
        System.out.println(str1.equals(str2));  
        System.out.println(str1.length());  
        System.out.println(str1.charAt(1));  
        System.out.println(str1.compareTo("aBcE"));  
        System.out.println(str1.compareTo("aBcC"));  
        System.out.println(str1.compareTo("aBcD"));  
        System.out.println(str1.indexOf('D'));  
        System.out.println(str1.indexOf("Bc"));  
        System.out.println(str1.indexOf("zz"));  
        System.out.println(str1.concat("efg"));  
    }  
}
```

```
public class StringTest {  
    public static void main(String [ ] args){  
        String str1 ="aBcD",str2 ="abcd",str3;  
        str3 =str1.toLowerCase();  
        System.out.println(str3);  
        str3 =str1.toUpperCase();  
        System.out.println(str3);  
        System.out.println(str1);  
        str3 =String.valueOf(123);  
        System.out.println(str3.equals("123"));  
    }  
}
```

StringBuffer

- Strings are *immutable*
 - Once you create one, you can't change it
 - You can only return a new string that is a changed version of the old one
- StringBuffers are *mutable*
 - You can change them: `insert()`, `reverse()`, `replace()`, `setCharAt()`, `setLength()`, `deleteCharAt()`, `append()`, ...

Elements of a Simple Class

- Data
 - called *instance variables, data members, fields*
- Methods
 - called *instance methods, procedure members, member functions*
- Together these implement a level of abstraction for some particular type of data

Defining a new type

- First describe the data that will be stored in objects of this type
- Then describe the operations that will be supported on objects of that type

Example: Counter

- We often want to count things, why not create an abstraction for doing it?
 - Advantage: you can reuse it in different places in the program, or even in other programs
- **Data:**
 - Current value of the counter (initially zero)
- **Operations:**
 - Reset, Increment, Decrement, Get the current value

Counter.java

```
class Counter {  
    int value;  
    void reset() { value = 0; }  
    int readValue() { return value; }  
    void increment() { value = value + 1; }  
    void decrement() { value = value - 1; }  
}
```

Using the Counter

```
Counter c1 = new Counter();  
Counter c2 = new Counter();  
c1.reset();  
c2.reset();  
c1.increment();  
c1.increment();  
System.out.println(c1.readValue());  
System.out.println(c2.readValue());
```

Abstract Data Types

- Classes allow us to implement Abstract Data Types (ADTs) – an abstraction representing a particular kind of data
 - The data and methods combine to implement the functionality we desire or expect for this type of data
 - The implementation details are hidden from the user
 - The implementation is all in one place
 - The type can be used in many different places in the program or in many programs

Important Details

- Each Counter object has its own copy of the member variables
 - In this case, the integer variable called *value*
- When the methods are called, the call is of the form
<objectname>.<methodname>()
- The object itself is an implicit parameter to the method, so that any references to the data access that object's copy of the member variables

More Examples

- **Complex numbers**, vectors, matrices, time/date information, address information, shapes (circle, square, rectangle, oval, triangle), **a file**, a keyboard, a game board (checkers, chess, **tictactoe**), a game piece, **a character string**, a die, **a deck of cards**
- Think of some more
- Let's implement some of them

Data Hiding: public vs. private

- In general, each class is in a separate file
 - The name of the file matches the name of the class (with .java at the end)
- All classes in the same directory (or file) are part of the same *package*
- Whether or not a method is in the same class or package as the data or method it is accessing affects what it can see and do

Public and Private

- Public data and methods are preceded by the keyword **public**
- Private data and methods are preceded by the keyword **private**
- By default, everything is semi-private (my term)
 - If you don't specify public or private, you get this default behavior

Public and Private in Action

- Private data and methods are accessible only by methods in the same class
- Semi-private data and methods are accessible by any method in the same class or the same package
- Public data and methods are accessible by any method, regardless of whether or not it is in the same class or package
- Protected data is available in the class, subclass or package

Example

```
//Counter.java -a simple counter
public class Counter {
    //instance variables - hidden
    private int value;

    //methods - exposed
    public void reset() {value = 0; }
    public int get() { return value; }
    public void click() {value = value +1; }
}
```

What If The Data Wasn't Private

```
Counter foo = new Counter();
```

```
foo.reset();
```

```
foo.click();
```

```
foo.click();
```

```
foo.value = 17;
```

```
int a = foo.get(); // returns the wrong value
```

Constructors

- A *constructor* is a special method in a class
- It has the same name as the class
- It is automatically called when an object of that type is created
- Constructors are usually used to set data in the object to an initial value
- Constructors can take parameters

Example

```
//Counter.java -a simple counter
public class Counter {
    //instance variables - hidden
    private int value;

    //methods – exposed
    public void Counter() { value = 0; }
    public void reset() { value = 0; }
    public int get() { return value; }
    public void click() { value = value +1; }
}
```

Example 2

```
public class Complex {  
    private double real;  
    private double imaginary  
    public void Complex() { real = 0; imaginary = 0;}  
    public void Complex(double r, double i) {  
        real = r; imaginary = i;  
    }  
    public double getReal() { return real; }  
    public double getImaginary() { return imaginary; }  
}
```

Using Constructors

```
Complex a = new Complex();
```

```
Complex b = new Complex(1, 5.7);
```

```
Complex c = new Complex(1,0);
```

Static Fields and Methods

- Static fields and methods are preceded by the keyword **static**
- Unlike other methods, static methods are not associated with a specific object
- Static methods are called by using the class name and the method name
 - `main();` and `Math.random();`

Static Variables

- Static data members are associated with the class rather than a particular object of that type
- Static data members are accessed like static methods: class name followed by field name
 - Example: Math.PI
- Sometimes called *class variables*

Example

```
public class Counter {  
    //instance variables - hidden  
    private int value;  
    private static int howMany = 0;  
  
    //methods - exposed  
    public Counter() { howMany++; }  
    public void reset() { value =0; }  
    public int get() { return value; }  
    public void click() { value = value + 1; }  
    public static int howMany() { return howMany; }  
}
```

```
class CounterTest2 {
    public static void main(String[] args){
        System.out.println(Counter.howMany());
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        c1.click();
        c2.click();
        c2.click();
        System.out.println("Counter1 value is " +
c1.get());
        System.out.println("Counter2 value is "+
c2.get());
        System.out.println(Counter.howMany());
    }
}
```

Recap: Calling Methods

- There are three ways to call a method depending on
 - whether the method is in the same class or not
 - whether the method is an instance method or a class method

The Three Ways to Call a Method

- In the same class: you just use the method name followed by any parameters in parenthesis
 - `int a = foo();` // foo is a method in this class
- An instance method: you have to call it for a particular object
 - `String s = "abc"; int a = s.length();`
- A class method: call it with the class name
 - `int a = Math.random();`

```
class Change {  
    private int dollars,quarters,dimes,pennies;  
    private double total;  
  
    Change(int dl,int q,int dm,int p) {  
        dollars = dl;  
        quarters = q;  
        dimes = dm;  
        pennies = p;  
        total =dl + 0.25*q + 0.1*dm + 0.01*p;  
    }  
}
```

```
static Change makeChange(double paid,  
                          double owed) {  
    double diff = paid - owed;  
    int dollars, quarters, dimes, pennies;  
  
    dollars = (int)diff;  
    pennies = (int)((diff - dollars)*100);  
    quarters = pennies /25;  
    pennies -= 25 *quarters;  
    dimes = pennies /10;  
    pennies -= 10 *dimes;  
    return new Change(dollars, quarters,  
                      dimes,pennies);  
}
```

```
public String toString() {  
    return (" $" + total + "\n"  
        + dollars + " dollars \n"  
        + quarters + " quarters \n"  
        + dimes + " dimes \n"  
        + pennies + " pennies \n");  
}  
}
```

Using the Class

```
//ChangeTest.java
public class ChangeTest {
    public static void main(String [ ] args){
        double owed = 12.37;
        double paid = 15.0;
        System.out.println("You owe " + owed);
        System.out.println("You gave me " + paid);
        System.out.println("Your change is " +
            Change.makeChange(15.0, 12.37));
    }
}
```

Accessing Another Objects Private Fields

```
//ChangeTest2.java
public class ChangeTest2 {
    public static void main(String[] args){
        Change c1 = new Change(10,3,4,3);
        Change c2 = new Change(7,2,2,1);
        Change sum = c1.add(c2);
        System.out.println(sum);
    }
}
```

The add() method

```
public Change add(Change addend) {  
    Change result = new Change(  
        dollars + addend.dollars,  
        quarters + addend.quarters,  
        dimes + addend.dimes,  
        pennies + addend.pennies);  
    return result;  
}
```

A static add() method

```
public static Change add(Change augend,  
                        Change addend) {  
    Change result = new Change(  
        augend.dollars + addend.dollars,  
        augend.quarters + addend.quarters,  
        augend.dimes + addend.dimes,  
        augend.pennies + addend.pennies);  
  
    return result;  
}
```

How it is called

```
public class ChangeTest3 {  
    public static void main(String[] args){  
        Change c1 = new Change(10,3,4,3);  
        Change c2 = new Change(7,2,2,1);  
        Change sum = Change.add(c1, c2);  
        System.out.println(sum);  
    }  
}
```

Passing Objects: Reference Types

- Passing an object to a method is different than passing a primitive type
- Primitive types are call-by-value
 - The called method gets a copy of the passed object
- Objects are call-by-reference
 - The called method gets a copy of the reference to the object, which refers to the same object!

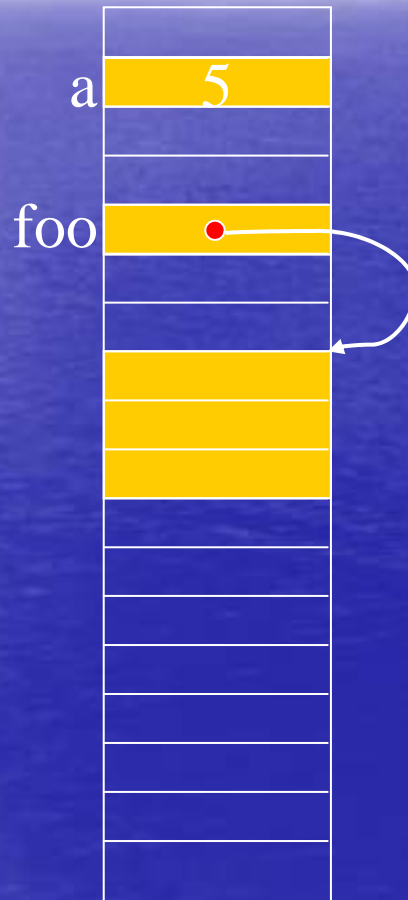
What's Really Going On Here

```
int a;
```

```
a = 5;
```

```
Complex foo;
```

```
foo = new Complex();
```



Call-by-reference

- The called method gets a copy of the reference!
- Because the called method has a reference to the same object, any changes to the object in a method will change the actual object!

Example

```
// Object parameters can be modified
class PassingReferences {
    public static void main(String [ ] args){
        StringBuffer sbuf =new StringBuffer("testing");
        System.out.println("sbuf is now "+ sbuf);
        modify(sbuf);
        System.out.println("sbuf is now "+ sbuf);
    }
    static void modify(StringBuffer sb) {
        sb.append(",1 2 3");
    }
}
```

Example

```
// You can't modify the actual arg
class ModifyParameters {
    public static void main(String[] args) {
        StringBuffer sbuf = new StringBuffer("testing");
        System.out.println("sbuf is now "+ sbuf);
        modify(sbuf);
        System.out.println("sbuf is now "+ sbuf);
    }
    static void modify(StringBuffer sb) {
        sb = new StringBuffer("doesn't work");
    }
}
```

Scope

- Recall: a local variable (defined in a method) is only accessible within that method
 - And, it is only accessible after the point at which it is defined in the method
- Instance and class variables are accessible from within any method in the class
 - Before or after the point at which they are defined in the class

Eclipsing an instance or class variable

- When a local variable (in a method) has the same name as an instance or class variable, the local variable eclipses the class variable
 - References to that name in the method will refer to the local variable
- An eclipsed class variable can be accessed using the class name
- An eclipsed instance variable can be accessed using *this*

```
// Scope2.java:instance vs. class vs. local scope
```

```
class Scope2 {  
    static int x =1;  
    int y = 2;  
    public static void main(String[] args){  
        int x = 3, y = 4;  
        System.out.println("local x = " + x);  
        System.out.println("class x =" + Scope2.x);  
        System.out.println("local y = " + y);  
        System.out.println("instance y =" + this.y);  
    }  
}
```

```
Change(int dollars,int quarters,int dimes,int pennies) {  
    this.dollars =dollars;  
    this.quarters =quarters;  
    this.dimes =dimes;  
    this.pennies =pennies;  
    total = dollars + 0.25 *quarters + 0.1 *dimes + pennies;  
}
```

Keyword **final** and Class Constants

- It is usually a bad idea to make instance and class variables public
 - It is better to provide accessor methods
 - This allows us to guarantee certain conditions about the data
- However, there is one type of class variable that is commonly made public: constants
 - Immutable variables with special values

Examples

- Math.PI
- Integer.MAXINT
- Note: generally written all uppercase
- Defined with the keyword **final**
public static final double PI = 3.14159265;
- Any attempt to modify a constant will result in an error

Why use these

- Constants are only defined once
 - Some numbers, such as pi, are used very often
- Constants allow us to name a value
 - Which is clearer: 60 or SECONDSPERMINUTE?

Arrays of Objects

- Just as we can have arrays of primitive types, we can also have arrays of objects
- Recall that
 - When we declare an array we have to use **new** to create the storage for the array
 - When we create an object we have to use **new** to create the storage for the object
- So, when we create an array of objects we have to use **new** twice; once for the array and once for the objects.

Example

```
int[] foo;  
foo = new int[15];
```

```
Complex[] bar;  
bar = new Complex[15];  
for(int i = 0; i < bar.length; i++)  
    bar[i] = new Complex();
```

- And, the book has a better Card class than mine

```
class Suit {  
    public static final int CLUBS = 1;  
    public static final int DIAMONDS = 2;  
    public static final int HEARTS = 3;  
    public static final int SPADES = 4;  
    int suitValue;  
    Suit(int i){ suitValue = i; }  
    public String toString() {  
        switch (suitValue) {  
            case CLUBS: return "clubs";  
            case DIAMONDS: return "diamonds";  
            case HEARTS: return "hearts";  
            case SPADES: return "spades";  
            default: return "error";  
        }  
    }  
}
```

```
class Pips {  
    int p;  
  
    Pips(int i) { p = i; }  
  
    public String toString() {  
        if (p > 1 && p <11)  
            return String.valueOf(p);  
        else switch(p) {  
            case 1: return "Ace";  
            case 11: return "Jack";  
            case 12: return "Queen";  
            case 13: return "King";  
            default: return "error";  
        }  
    }  
}
```

```
class Card {  
    Suit suit;  
    Pips pip;  
  
    Card(Suit s,Pips p) { suit =s; pip =p; }  
  
    Card(Card c) { suit = c.suit; pip = c.pip;}  
  
    public String toString() {  
        return pip.toString()+" of "+suit.toString();  
    }  
}
```



```
public void shuffle() {  
    for (int i = 0; i < deck.length; i++) {  
        int k =(int)(Math.random()*52);  
        Card t = deck[i];  
        deck[i] = deck[k];  
        deck[k] = t;  
    }  
}
```

```
public String toString(){
    String t = "";

    for (int i = 0; i < 52; i++)
        if ((i + 1)%5 == 0)
            t = t + "\n" + deck [i];
        else
            t = t +deck [i];

    return t;
}
}
```

```
public class CardTest {  
    public static void main(String args[]) {  
        Deck deck = new Deck();  
  
        System.out.println("\nNew Shuffle \n"+deck);  
  
        deck.shuffle();  
  
        System.out.println("\nNew Shuffle \n"+deck);  
  
        deck.shuffle();  
  
        System.out.println("\nNew Shuffle \n"+deck);  
    }  
}
```