Inheritance and Polymorphism

Motivation

- Consider a transportation computer game
  - Different types of vehicles:
    - Planes
      - Jets, helicopters, space shuttle
    - Automobiles
      - Cars, trucks, motorcycles
    - Trains
      - Diesel, electric, monorail
    - Ships
      - …

- Let's assume a class is written for each type of vehicle

Motivation

- Sample code for the types of planes:
  - fly()
  - takeOff()
  - land()
  - setAltitude()
  - setPitch()
- Note that a lot of this code is common to all types of planes
  - They have a lot in common!
  - It would be a waste to have to write separate fly() methods for each plane type
    - What if you then have to change one – you would then have to change dozens of methods

Motivation

- Indeed, all vehicles will have similar methods:
  - move()
  - getLocation()
  - setSpeed()
  - isBroken()

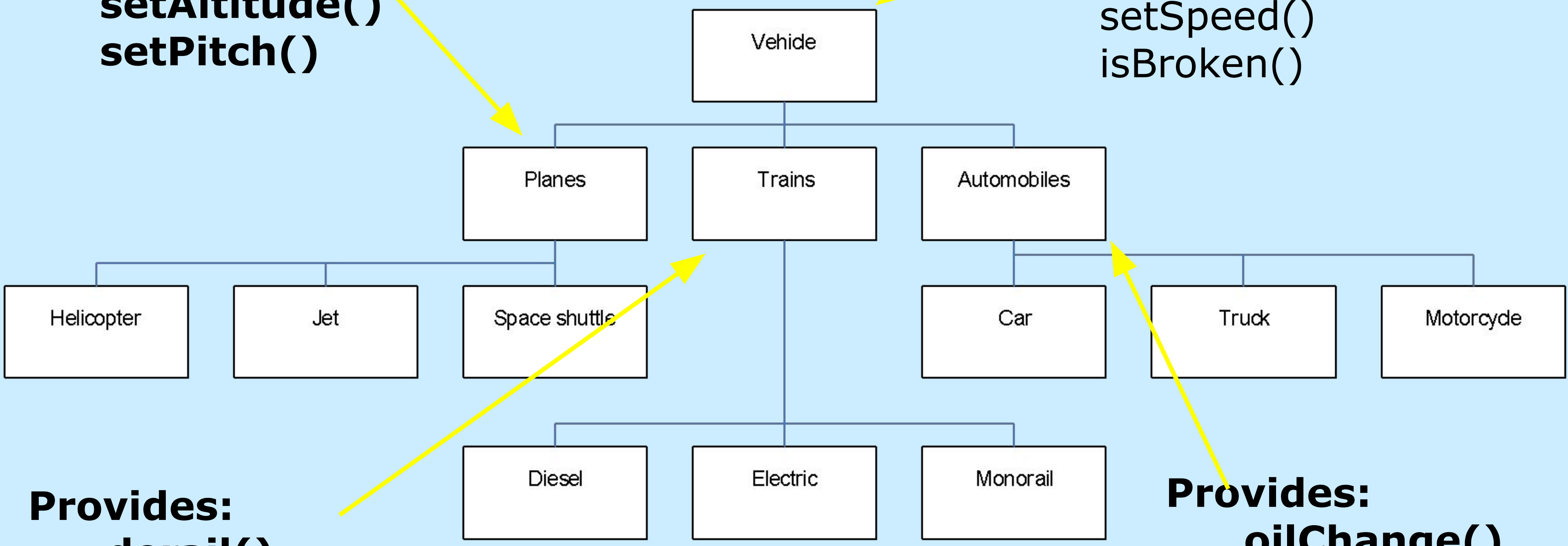
- Again, a lot of this code is common to all types of vehicles
  - It would be a waste to have to write separate move() methods for each vehicle type
    - What if you then have to change one – you would then have to change dozens of methods

- What we want is a means to specify one move() method, and have each vehicle type ***inherit*** that code
  - Then, if we have to change it, we only have to change one copy

Motivation

Provides:
- **fly()**
- **takeOff()**
- **land()**
- **setAltitude()**
- **setPitch()**

Provides:
- move()
- getLocatio
n()
- setSpeed()
- isBroken()



Provides:
- **derail()**
- **getStation()**

Provides:
- **oilChange()**
- **isInTraffic()**

Motivation

- What we will do is create a “parent” class and a “child” class

- The “child” class (or subclass) will inherit the methods (etc.) from the “parent” class (or superclass)

- Note that some classes (such as Train) are both subclasses and superclasses

Inheritance code

```
class Vehicles {
   ...
}

class Train extends Vehicles {
 ...
}

class Monorail extends Train {
 ...
}
```

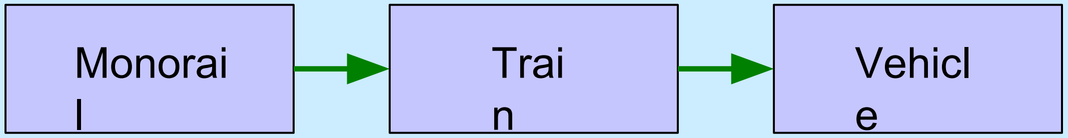
About extends

- If class A extends class B
  - Then class A is the subclass of B
  - Class B is the superclass of class A
  - A “is a” B
  - A has (almost) all the methods and variables that B has

- If class Train extends class Vehicle
  - Then class Train is the subclass of Vehicle
  - Class Vehicle is the superclass of class Train
  - Train “is a” Vehicle
  - Train has (almost) all the methods and variables that Vehicle has

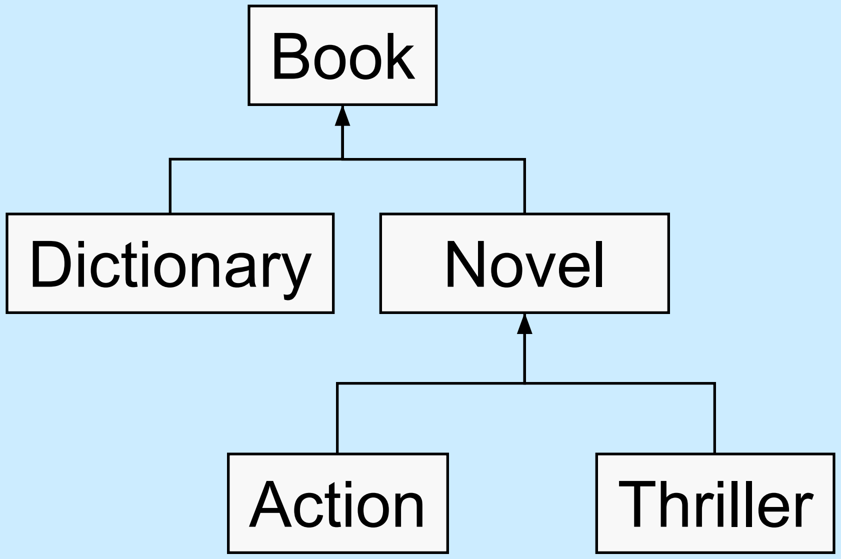
Object-oriented terminology

- In object-oriented programming languages, a class created by extending another class is called a *subclass*
- The class used for the basis is called the *superclass*
- Alternative terminology
  - The superclass is also referred to as the *base* class
  - The subclass is also referred to as the *derived* class



Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class* or *superclass*
- The derived class is called the *child class* or *subclass*.
- The subclass is a more specific version of the Original



Inheritance

- The child class inherits the methods and data defined for the parent class
- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Software reuse* is at the heart of inheritance

Inheritance: a Basis for Code Reusability

- **Fast implementation** - we need not write the implementation of `all classes` from scratch, we just implement the additional functionality.
- **Ease of use** - If someone is already familiar with the base class, then the derived class will be easy to understand.
- **Less debugging** - debugging is restricted to the additional functionality.
- **Ease of maintenance** - if we need to correct/improve the implementation of `base class`, `derive class` is automatically corrected as well.
- **Compactness** - our code is more compact and is easier to understand.

Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Dictionary extends Book {

      // class contents
}
```

```
Dictionary webster = new Dictionary();

webster.message();

webster.defMessage();
```

```
Number of pages: 1500

Number of definitions: 52500

Definitions per page: 35
```

```
public class Book {

     protected int pages = 1500;

     public String message() {
          System.out.println("Number of pages: " + pages);
     }
}

public class Dictionary extends Book {

     private int definitions = 52500;

     public void defMessage() {
          System.out.println("Number of definitions" +
                              definitions);
          System.out.println("Definitions per page: " +
                             (definitions/pages));
     }
}
```

Some Inheritance Details

- An instance of a child class does not rely on an instance of a parent class
    - Hence we could create a Dictionary object without having to create a Book object first

- Inheritance is a one-way street
    - The Book class cannot use variables or methods declared explicitly in the Dictionary class

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a member of a base class to be inherited into a child
- Protected visibility provides
  - more encapsulation than public visibility does
  - the best possible encapsulation that permits inheritance

Accessing Class Members within a package and from another package

Any class has direct access to any other class name But members are not necessarily accessible, it is controlled by access attributes

No access attribute	from methods in any class of the same package
public	From methods in any class anywhere
private	Only accessible from methods inside the class. No access from outside the class at all
protected	From methods in the same package. From any subclass anywhere

protected Members

- A superclass’s protected members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package.

- For non-subclasses, they act as private members of the class.

- Public members of the superclass become public members of the subclass, and protected members of the superclass become protected members of the subclass.

protected Members

- Subclass methods can refer to public and protected members inherited from the superclass simply by using the member names.

- When a subclass method overrides an inherited superclass method, the superclass method can be accessed from the subclass by preceding the superclass method name with keyword **super** and a dot (.) separator.

The super Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The **`super`** reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the **`super`** reference to call the parent's constructor
- The **`super`** reference can also be used to reference other variables and methods defined in the parent's class

The keyword "super"

- It is possible to access overriding members by using the **super** keyword
- **Super** much like **this** keyword except that super doesn't refer in the current object but rather to its superclass.

Constructors of Subclasses

- Can invoke a constructor of the direct superclass.
  - super(…) must be the first statement.
  - If the super constructor call is missing, by default the no-arg super() is invoked implicitly.
- Can also invoke another constructor of the same class.
  - this(…) must be the first statement.

```
public class Book {

    protected int pages;

    Book(int numPages) {
        pages = numPages;
    }
}


public class Dictionary extends Book {

    private int definitions;

    Dictionary(int numPages, int numDefinitions) {
        super(numPages);
        definitions = numDefinitions;
    }
}
```

Example of “this” Calls

```
public class Point {
  private int x, y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public Point() { // default constructor
    this(0,0);
  }
}
```

Example of “super” Calls

```
public class ColoredPoint extends Point {
  private Color color;

  public ColoredPoint(int x, int y, Color color) {

      super(x,y);

    this.color = color;
  }

  public ColoredPoint(int x, int y) {
    this(x, y, Color.BLACK); // point with default value
  }

  public ColoredPoint() {
    color = Color.BLACK;    // what will be the values of x and y?
  }
}
```

Default Constructor

- If no constructor is defined, the following form of no-arg default constructor is automatically generated by the compiler.

```
public ClassName() {
  super();
}
```

Execution Order of Constructors

Rule: Super class’s field initializes first

Example: S x = new S();

```
public class S extends T {
  int y = 30;  // third

  public S() {
    super();
    y = 40;   // fourth
  }
  // ...
}
```

```
public class T {
  int x = 10;  // first

  public T() {
    x = 20;    // second
  }
  // ...
}
```

More Examples

```
class BaseClass
{
   public void doSomething()
   {
      System.out.println("BaseClass doSomething");
   }
}
class SubClass extends BaseClass
{
}
public class InheritanceExample1
{
   public static void main(String args[])
   {
      SubClass sc = new SubClass();
      sc.doSomething();
      BaseClass bc = new SubClass();
      bc.doSomething();
   }
}
```

More Examples (cont.)

```
class BaseClass{
    public void doSomething()       {
                  System.out.println("BaseClass doSomething");
    }
}
class SubClass extends BaseClass{
    public void doSomething()       {
        System.out.println("SubClass doSomething");
    }
}
public class InheritanceExample2      {
    public static void main(String args[])    {
        SubClass sc = new SubClass();
        sc.doSomething();
        BaseClass bc = new SubClass();
        bc.doSomething();
    }
}
```

More Examples (cont.)

```
class BaseClass{
    public void doSomething(){
            System.out.println("BaseClass doSomething");
    }
}
class SubClass extends BaseClass{
    public void doSomething()    {
        System.out.print("Super: ");
        super.doSomething();
        System.out.println("SubClass doSomething");
    }
}
```

Overriding Methods

- When a child class defines a method with the same name and signature as a method in the parent class, we say that the child's version **overrides** the parent's version in favor of its own.
  - **Signature:** method's name along with number, type, and order of its parameters
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked

Overriding

- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overriding Methods (Cont.)

```
public class T {
  public void m() { … }
}

public class S extends T {
  public void m() { … }
}

T t = new T();
S s = new S();
t.m(); // invoke m of class T
s.m(); // invoke m of class S
```

Overriding Methods (Cont.)

- Dynamic dispatch (binding): The method to be invoked is determined at runtime by the runtime type of the object, not by the declared type (static type).

```
class Student {
  public int maxCredits() { return 15; }
  …
}
class GraduateStudent extends Student {
  public int maxCredits() { return 12; }
  …
}

Student s;
// …
s.getMaxCredits(); // which maxCredits method?
```

```
public class Book {

    protected int pages;

    Book(int numPages) {
        pages = numPages;
    }

    public void message() {
        System.out.println("Number of pages: " + pages);
    }
}
public class Dictionary extends Book{

    protected int definitions;

    Dictionary(int numPages, int numDefinitions) {
        super(numPages);
        definitions = numDefinitions;
    }

    public void message() {
        System.out.println("Number of definitions" +
                                 definitions);
        System.out.println("Definitions per page: " +
                                 (definitions/pages));
        super.message();
    }
}
```

Overloading vs. Overriding

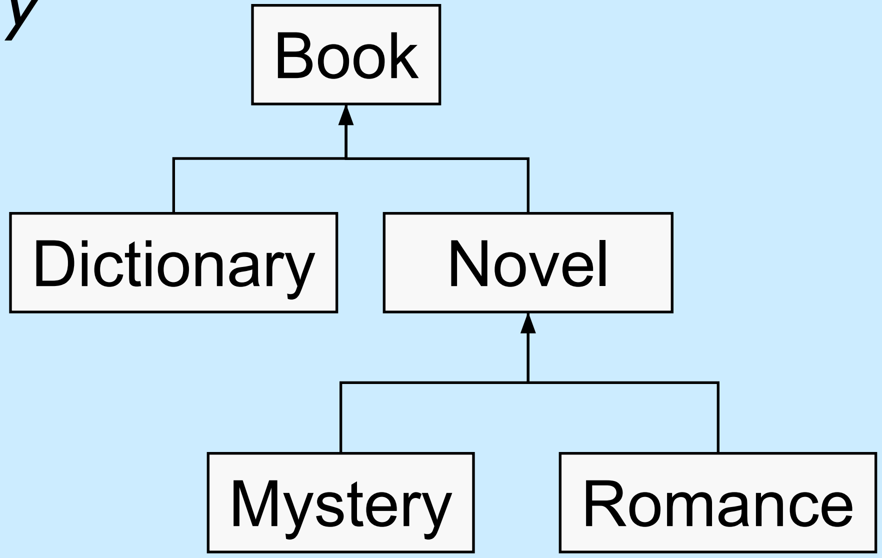
- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead (will discuss later)

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Class Hierarchies

- Two children of the same parent are called *siblings*
  - *However they are not related by inheritance because one is not used to derive another.*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes

Objects Life Cycle

- Stages
  - creation
  - use
  - cleanup

Creation of an object

- Class Declaration (providing name and definition for the object)
- instantiation (setting aside memory for the object)
- optional initialization (providing initial values for the object via constructors)

Use of an object

- We activate the behavior of an object by *invoking one of its methods* (sending it a message).
- When an object receives a message (has one of its methods invoked), it either *performs an action*, or *modifies its state*, or both.

Cleanup

- What happens to all the objects that are instantiated?
  - When an object is no longer needed, we simply forget it. Eventually, the garbage collector may (or may not) come by and pick it up for recycling

Cleanup Approach

- **The good news**
  - if things work as planned, the Java programmer never needs to worry about returning memory to the operating system. This is taken care of automatically by a feature of Java known as the *garbage* collector.
- **The bad news**
  - Java does not support anything like a *destructor* that is guaranteed to be called whenever the object goes out of scope or is no longer needed. Therefore, other than returning allocated memory, it is the responsibility of the programmer to explicitly perform <u>any other required cleanup</u> at the appropriate point in time.
  - Other kinds of cleanup could involve closing files, disconnecting from open telephone lines, etc.

Garbage Collection

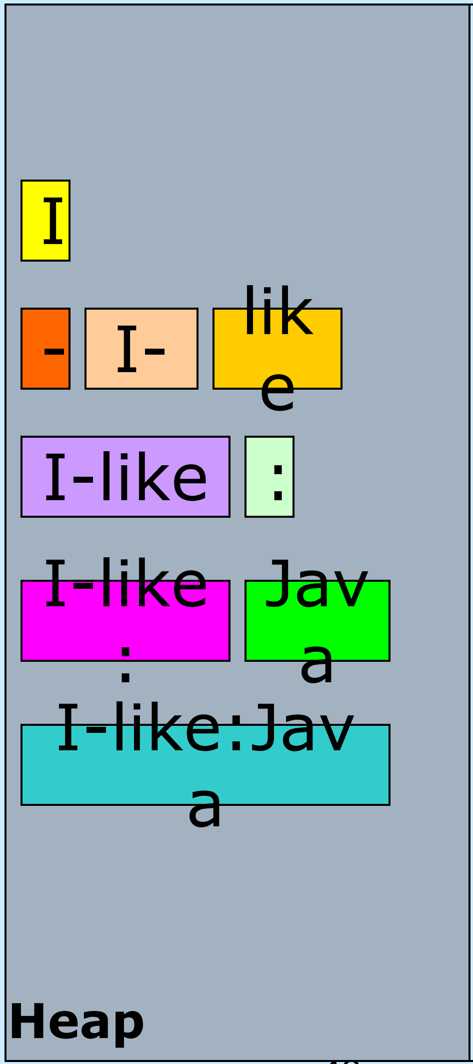
- The sole purpose of *garbage collection* is to reclaim memory occupied by objects that are no longer needed.
- Eligibility for garbage collection
  - An object becomes *eligible* for *garbage collection* when there are no more references to that object. You can make an object *eligible* for garbage collection by setting all references to that object to **null**, or allowing them to go out of scope.

No Guarantees

- However, just because an object is *eligible* for *garbage collection* doesn't mean that it will be reclaimed.
- The *garbage collector* runs in a low-priority thread, and presumably is designed to create minimal interference with the other threads of the program. Therefore, the garbage collector may not run unless a memory shortage is detected. And when it does run, it runs asynchronously relative to the other threads in the program.

Garbage Collection (contd.)

```
1. String str1 = "I";
2. str1 = str1 + "-" +
      "like";
3. String str2 = str1
      + ":" + "Java";
4. str1 = null;
```



1. "I" is created. str1 points to "I".
2.1. "-" is created (used internally).
2.2. "I-" is created (used internally).
2.3. "like" is created (used internally).
2.4. "I-like" is created. str1 now points to "I-like". "I", "-", "I-", "like" are marked for garbage collection.
3.1. ":" is created (used internally).
3.2. "I-like:" is created (used internally).
3.3. "Java" is created (used internally).
3.4. "I-like:Java" is created. str2 points to "I-like:Java". ":", "I-like:", "Java" are marked for garbage collection.
4. str1 is set to null. So "I-like" is marked for garbage collection.

Finalize Method

- Before the *garbage collector* reclaims the memory occupied by an object, it calls the object's **finalize()** method.
- The **finalize()** method is a member of the **Object** class. Since all classes inherit from the **Object** class, your classes also contain the default **finalize()** method. This gives you an opportunity to execute your special cleanup code on each object before the memory is reclaimed
- In order to make use of the **finalize()** method, you must *override* it, providing the code that you want to have executed before the memory is reclaimed.

When do I use the finalize() method

- Is the cleanup timing critical?
  - If you simply need to do cleanup work on an object sometime before the program terminates, (and you have specified finalization on exit) you can ALMOST depend on your overridden **finalize()** method being executed sometime before the program terminates.
  - If you need cleanup work to be performed earlier (such as disconnecting an open long-distance telephone call), you must explicitly call methods to do cleanup at the appropriate point in time and not depend on finalization to get the job done.
- If you use the finalize() method, make sure that you call the super.finalize() method at the end of it.

The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies

The Object Class

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we have defined `toString`, we have actually been overriding an existing definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class together along with some other information
  - All objects are guaranteed to have a `toString` method via inheritance, thus the `println` method can call `toString` for any object that is passed to it

The Object Class

- The **`equals`** method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- The `String` class (as we've seen) defines the `equals` method to return true if two `String` objects contain the same characters
- Therefore the `String` class has overridden the `equals` method inherited from `Object` in favor of its own version

Using “==” on References

- If two references are compared using “==”, then Java just examines whether they point to the same object or not
  - By default, the “equals” methods also performs the same check
- If we want to assume that two different objects are equal when their members have identical values, we have to override the “equals” method and use it instead of “==”