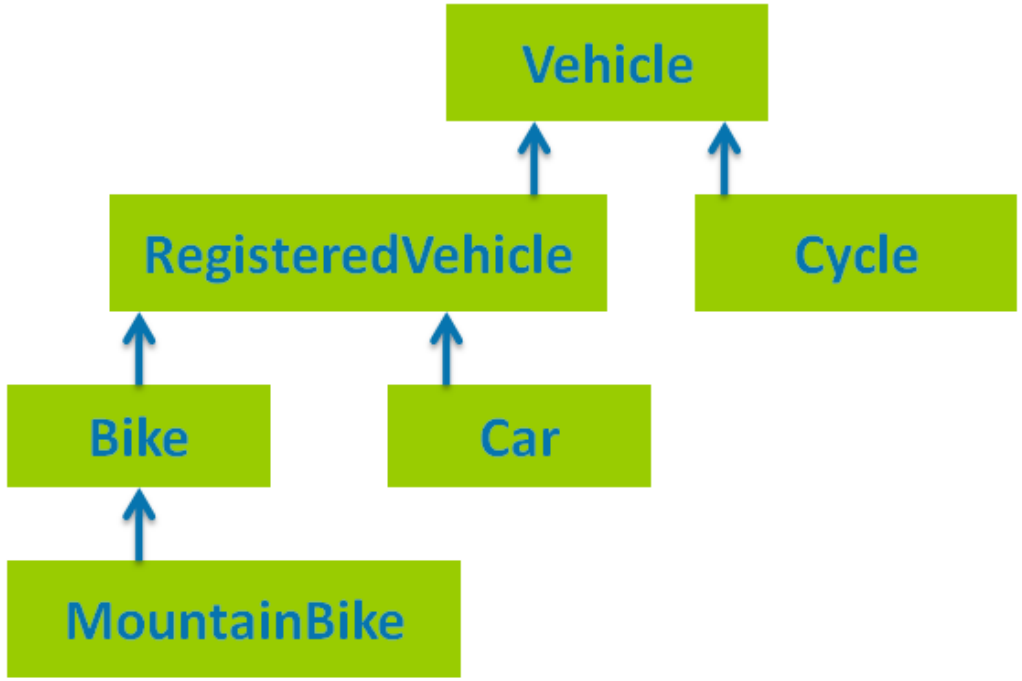
Abstract Class
(Advanced Inheritance Concepts)

Outlines

ADVANCED CONCEPTS

- Abstract class
- Dynamic binding
  - Create a single method that has one or more parameters that might be one of several types
  - Create a single array of superclass object references but store multiple subclass instances in it
- Using the "object" class



Abstract Classes

- Abstract (template) classes cannot be instantiated, meaning you cannot create new instances of an abstract class.
- The purpose of an abstract class is to function as a base for subclasses.

Code

```
public abstract class MyAbstractClass { … }

MyAbstractClass myClassInstance = new MyAbstractClass();
//not valid
```

Abstract Classes (cont.)

- An abstract class can have abstract (template) methods. You declare a method abstract by adding the abstract keyword in front of the method declaration.
  - An abstract method has no implementation. It just has a method signature.
  - Subclasses of an abstract class must implement (override) all abstract methods of its abstract superclass.

Code

```
public abstract class MyAbstractClass {
   public abstract void abstractMethod();
}
```

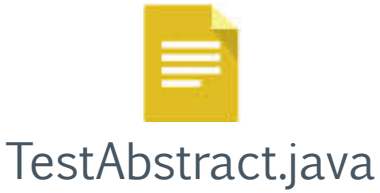
```
public class MySubClass extends MyAbstractClass {
    public void abstractMethod() {
        System.out.println("My method implementation");
    }
}
```

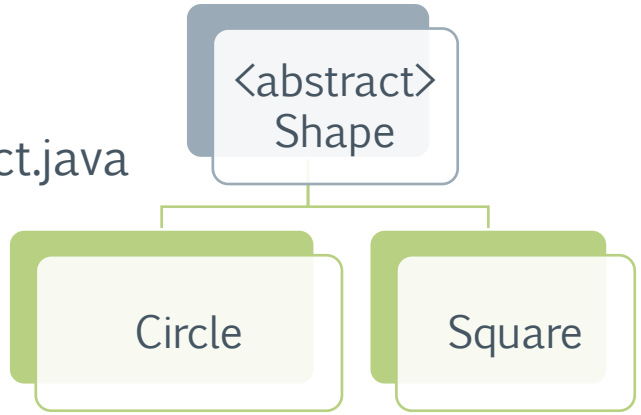
Abstract Classes (cont.)

- Abstract method does not have:
  - Body
  - Curly braces
  - Method statements
- To create abstract method
  - Keyword `abstract`
  - Header including method type, name, and arguments
  - Include semicolon at end of declaration

```
public abstract void speak();
```

- Subclass of abstract class
  - Inherits abstract method from parent
    - Must provide implementation for inherited method
    - Or be abstract itself
  - Code subclass method to override empty superclass method


TestAbstract.java



Using Dynamic Method Binding

- Static binding (Early binding) vs. Dynamic binding (Late binding)
  - In static binding, the method or variable version that is going to be called is resolved at compile time,
  - While in dynamic binding the compiler cannot resolve which version of a method or variable is going to bind.
- Every subclass object "is a" superclass member
  - Convert subclass objects to superclass objects
  - Can create reference to superclass object
    - Create variable name to hold memory address
    - Store concrete subclass object
    - Example:

```
Animal ref;
ref = new Cow();
```

- Dynamic method binding
  - Application's ability to select correct subclass method
  - Makes programs flexible
- When application executes
  - Correct method attached to application based on current one

Using Dynamic Method Binding (cont.)

StudentTest4.java

```
public class StudentTest4 {
  public static void main(String[] args) {
    Student s;
    GraduateStudent g = new GraduateStudent("Nat");
    UndergraduateStudent u = new UndergraduateStudent("Toey");
    // This is called Dynamic binding, as the compiler will never know
    // which version of printName() is going to called at runtime.
    s = g;
    s.printName();
    s = u;
    s.printName();
  }
}
```

Result

```
GraduateStudent [Nat]
UndergraduateStudent [Toey]
```

Using a Superclass as a Method Parameter Type (method argument)

1

```
public class TalkingAnimalDemo
{
   public static void main(String[] args)
   {
      Dog dog = new Dog();
      Cow cow = new Cow();
      dog.setName("Ginger");
      cow.setName("Molly");
      talkingAnimal(dog);
      talkingAnimal(cow);
   }
   public static void talkingAnimal(Animal animal)
   {
      System.out.println("Come one. Come all.");
      System.out.println
         ("See the amazing talking animal!");
      System.out.println(animal.getName() +
         " says");
      animal.speak();
      System.out.println("***************");
   }
}
```

Creating Arrays of Subclass Objects 2

- Create superclass reference
  - Treat subclass objects as superclass objects
    - Create array of different objects
    - Share same ancestry
- Creates array of three `Animal` references

```
Animal[] ref = new Animal[3];
```

  - Reserve memory for three `Animal` object references

Using the "object" class

- `Object` Class
  - Every Java class extension of `Object` class
  - Defined in `java.lang` package
  - Imported automatically
  - Includes methods to use or override

Method summary

```
clone()
equals(Object obj)
hashcode()
finalize()
toString()
getClass()
```

Methods

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

```
notify()
notifyAll()
wait(); wait(long timeout);
wait(long timeout, int nanos)
```

Cannot override

The equals() method

http://www.javaranch.com/journal/2002/10/equalhash.html

- Compares this object to the specified object.
- The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

- The equals method for class Object implements only the most discriminating possible equivalence relation on objects as follow:

Object.equals

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

The equals() method (cont.)

- The equals method implements an equivalence relation on non-null object references:
- It is reflexive: for any non-null reference value x, x.equals(x) should return true.
- It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

The equals() method (cont.)

```
public class BankAccount
{
   private int acctNum;
   private double balance;
   public BankAccount(int num, double bal)
   {
      acctNum = num;
      balance = bal;
   }
   public String toString()
   {
      String info = "BankAccount acctNum = " + acctNum +
        "    Balance = $" + balance;
      return info;
   }
   public boolean equals(BankAccount secondAcct)
   {
      boolean result;
      if(acctNum == secondAcct.acctNum &&
           balance == secondAcct.balance)
        result = true;
      else
        result = false;
      return result;
   }
}
```

The equals() method (cont.)

Employee.java

```
public class Employee {
  int employeeId;
  String firstName, lastName;
  public boolean equals(Object o){
    if(this == o)                          return true;
    if(o == null)                          return false;
    if(o.getClass() != this.getClass())    return false;

    Employee other = (Employee) o;
    if(this.employeeId != other.employeeId)       return false;
    if(! this.firstName.equals(other.firstName)) return false;
    if(! this.lastName.equals(other.lastName))    return false;
    return true;
  }
}
```

The hashCode() method

- The value returned by `hashCode` is an `int` that maps an object into a bucket in a hash table.
- An object must always produce the same hash code.
- If you override `equals`, you must override `hashCode`.
- `hashCode` must generate equal values for equal objects.

```
public int hashCode() { . . . }
```

Equal objects must produce the same hash code as long as they are equal
If obj1.equals(obj2) is true, obj1.hashCode() == obj2.hasCode() must be true

; however unequal objects need not produce distinct hash codes.

The hashCode() method (cont.)

HashCodeTest.java

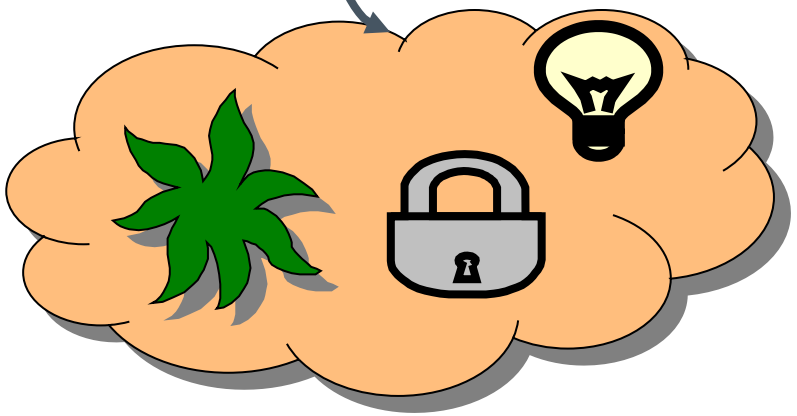
```
public class Employee {
  private int num;
  private String data;

  public boolean equals(Object obj) {}
  public int hashCode() {
    int hash = 7;
    hash = 31 * hash + num;
    hash = 31 * hash + (null == data ? 0 : data.hashCode());
    return hash;
  }
  // other methods
}
```

The clone() method

› To create an object from an existing object

```
aCloneableObject.clone()
```

- x.clone() != x
- x.clone().getClass() == x.getClass()
- x.clone().equals(x)



Clone example

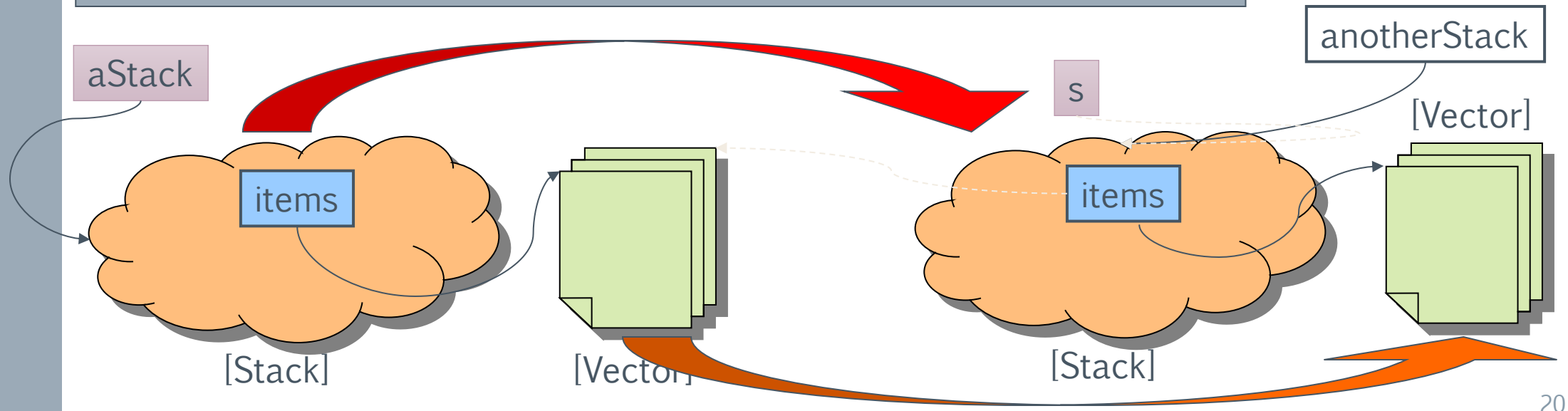
```
public class Stack implements Cloneable {
  private Vector items;

  // code for Stack's methods and constructor not shown

  protected Object clone() {
    try {
      Stack s = (Stack)super.clone();  // clone the stack
      s.items = (Vector)items.clone(); // clone the vector
      return s;                        // return the clone
    } catch (CloneNotSupportedException e) {
      // this shouldn't happen because Stack is Cloneable
      throw new InternalError();
    }
  }
}
```

To have **clone()**, one must implement **Cloneable**, otherwise **CloneNotSupportedException** will be thrown.

Clone example

```
Stack aStack = new Stack();
Stack anotherStack = aStack.clone();
```

```
    Stack s = (Stack)super.clone();
    s.items = (Vector)items.clone();
    return s;
```



More clone example

```
class A {
  private int x;
  public A(int i) {
    x = i;
  }
}

public class CloneDemo1 {
  public static void main(String args[])
      throws CloneNotSupportedException {
    A obj1 = new A(37);
    A obj2 = (A)obj1.clone();
  }
}
```

compile error:
because **`Object.clone()`**
is a protected method.

More clone example

```
class A {
  private int x;
  public A(int i) {
    x = i;
  }
  public Object clone() {
    try {
      return super.clone();
    }
    catch (CloneNotSupportedException e) {
      throw new InternalError(e.toString());
    }
  }
}
```

```
public class CloneDemo2 {
  public static void main(String args[])
      throws CloneNotSupportedException {
    A obj1 = new A(37);
    A obj2 = (A)obj1.clone();
    }
}
```

CloneNotSupportedException is thrown at runtime.

More clone example

```
class A implements Cloneable {
  private int x;
  public A(int i) {
    x = i;
  }
  public Object clone() {
    try {
      return super.clone();
    }
    catch (CloneNotSupportedException e) {
      throw new InternalError(e.toString());
    }
  }
  public int getx() {
    return x;
  }
}
```

```
public class CloneDemo3 {
  public static void main(String args[])
      throws CloneNotSupportedException {
    A obj1 = new A(37);
    A obj2 = (A)obj1.clone();
    System.out.println(obj2.getx());
  }
}
```

success!

There are other ways better than clone()

- 1) Copy constructor
- 2) Static factory

```
class A4 implements Cloneable {
    private int x;

    public A4(int i) {
        this.x = i;
    }

    // 1. copy constructor
    public A4(A4 other) {
        this.x = other.getx();
    }

    // 2. static factory
    public static A4 newInstace(A4 other) {
        A4 obj = new A4(other.getx());
        return obj;
    }

    public int getx() {
        return x;
    }
}
```

CloneDemo4.java

```
public class CloneDemo4 {
    public static void main(String args[]) {
        A4 obj1 = new A4(37);
        A4 obj2 = new A4(obj1);
        System.out.println(obj2.getx());

        A4 obj3 = A4.newInstace(obj1);
        System.out.println(obj3.getx());
    }
}
```

The finalize() method

http://howtodoinjava.com/2012/10/31/why-not-to-use-finalize-method-in-java/

- Before an object is garbage collected, the runtime system calls its finalize() method.
- The intent is for finalize() to release system resources such as open files or open sockets before getting collected.
- Guide for correct usage:
  - Always call super.finalize() in your finalize() method.
  - Do not put time critical application logic in finalize(), seeing its unpredictability. (finalize() add heavy penalty in performance)
  - Do not use Runtime.runFinalizersOnExit(true); as it can put your system in danger.

The finalize() method (cont.)

› Try to follow below template for finalize method

```
@Override
protected void finalize() throws Throwable
{
    try{
        //release resources here
    }catch(Throwable t){
        throw t;
    }finally{
        super.finalize();
    }
}
```

The finalize() method (cont.)

OpenAFile.java

```
import java.io.FileInputStream;
class OpenAFile {
  FileInputStream aFile = null;
  OpenAFile(String filename) {
    try { aFile = new FileInputStream(filename);  }
    catch (java.io.FileNotFoundException e)
    { System.err.println("Could not open file " + filename);}
  }
  protected void finalize() throws Throwable {
    super.finalize();
    if (aFile != null) {
      aFile.close();
      aFile = null;
    }
  }
}
```