Java Application Design

Threads and RTTI

Weng Kai
http://fm.zju.edu.cn

Synchronized Section

synchronized (object) { // }

- The key is in an object, not in the code.
  Case Study: Crunch2.java

1. There is a key in every object.
2. To execute synchronized() block, the thread need to get the key in the object. Once the key is got, the object does not have the key.
3. If the key were not in the object when the thread wants to exec. synchronized(), the thread is to be stall until the key returns to the object.
4. The key is to be returned to the object when the thread leave the synchorized() block.

How to protect data?

- synchronized() is not to protect the data, but to guarantee there is only one thread at a time.

Tips to protect data:

1. Private data
2. All access to the data is synchronized
3. The key is in the data itself

Nested synchronized

- Nested synchronized is safe in Java

```
synchronized(a) {
    synchronized(a) {
    }
}

synchronized(a) {
    f();
}
```

Synchronized method

```
void f() {
   synchronized(this) {
      ...
   }
}

synchronized void f() {
}
```

Avoiding Deadlocks

Step	Thread 1	Thread 2
1	`synchronized (object1) {`	`synchronized (object2) {`
2	`// do something here`	`// do something here`
3		
4		
5	`synchronized (object2) {`	`synchronized (object1) {`
6	`// do something here`	`// do something here`
	`}`	`}`
	`}`	`}`

Wait for Thread 2 to release the lock on `object2`

Wait for Thread 1 to release the lock on `object1`

Communication among threads

- PipedInputStream/PipedOutputStream
  Case Study: Pipe.java

Producer and consumer

- is a pattern that one thread produces data and the other one read it. There must be a shared variable for transportation and a flag to indicate the data is valid or has been read.

Case Study: FlagComm.java

Wait() & notify()

- wait() and notify() of Object
  Every object can have a thread pool. A thread can call wait() to join the pool and call notify() to leave the pool.
  Case Study: WaitComm.java

Class Class

Class Object

Class Object

Object is the base class for all java classes. Every class *implicitly* extends Object.

Class Object

Object is the base class for all java classes. Every class *implicitly* extends Object.

Among the methods contained in Object are the following:

Class Object

Object is the base class for all java classes. Every class *implicitly* extends Object.

Among the methods contained in Object are the following:

```
public boolean equals(Object x) {…}

public String toString( ) {…}

public Class getClass(Object x)  {…}

public int hashCode( )  {…}

protected Object clone(Object x)  {…}

public void wait( )  {…}                    //used in multithreading

public void notify( ) {…}                    //used in multithreading
```

Class Object

Object is the base class for all java classes. Every class *implicitly* extends Object.

Among the methods contained in Object are the following:

```
public boolean equals(Object x) {…}

public String toString( ) {…}

public Class getClass(Object x)  {…}

public int hashCode( )  {…}

protected Object clone(Object x)  {…}

public void wait( )  {…}                    //used in multithreading

public void notify( ) {…}                    //used in multithreading
```

We will examine the first five of these methods in more detail.

Class Object

Method equals( )

Class Object

Class Object

Method equals( )

Class Object

Method equals( )

The default behavior of equals( ) is to compare references. You must override this method in the class you are constructing (or understand how it is overridden in the library class you are using) if you are using this method to test whether two objects have the same value.

Class Object

Method equals( )

The default behavior of equals( ) is to compare references. You must override this method in the class you are constructing (or understand how it is overridden in the library class you are using) if you are using this method to test whether two objects have the same value.

In class String – equals( ) is overridden to test whether the two String objects have the same value.

Class Object

Method equals( )

The default behavior of equals( ) is to compare references. You must override this method in the class you are constructing (or understand how it is overridden in the library class you are using) if you are using this method to test whether two objects have the same value.

In class String – equals( ) is overridden to test whether the two String objects have the same value.

```
String s1 = “A string”;
String s2 = “A” + new String(“ string”);
System.out.println(“Strings s1 and s2 have same reference: ”+s1==s2);

System.out.println(“Strings s1 and s2 have same value: ”+s1.equals(s2));
```

Class Object

Method equals( )

The default behavior of equals( ) is to compare references. You must override this method in the class you are constructing (or understand how it is overridden in the library class you are using) if you are using this method to test whether two objects have the same value.

In class String – equals( ) is overridden to test whether the two String objects have the same value.

```
String s1 = “A string”;
String s2 = “A” + new String(“ string”);
System.out.println(“Strings s1 and s2 have same reference: ”+s1==s2);
```

false

```
System.out.println(“Strings s1 and s2 have same value: ”+s1.equals(s2));
```

Class Object

Method equals( )

The default behavior of equals( ) is to compare references. You must override this method in the class you are constructing (or understand how it is overridden in the library class you are using) if you are using this method to test whether two objects have the same value.

In class String – equals( ) is overridden to test whether the two String objects have the same value.

```
String s1 = “A string”;
String s2 = “A” + new String(“ string”);
System.out.println(“Strings s1 and s2 have same reference: ”+s1==s2);
```

false

```
System.out.println(“Strings s1 and s2 have same value: ”+s1.equals(s2));
```

true

Method toString( )

Class Object

Class Object

Method toString( )

Class Object

Method toString( )

The default method toString( ) contained in class Object returns a String giving

- the name of the Class and
- the address of the Object in memory.

Class Object

Method toString( )

The default method toString( ) contained in class Object returns a String giving

- the name of the Class and
- the address of the Object in memory.

Consider the following example

Class Object

Method toString( )

The default method toString( ) contained in class Object returns a String giving

- the name of the Class and
- the address of the Object in memory.

Consider the following example

```
class ShowObject {
    private int val1, val2;
    public ShowObject(int v1, int v2) {val1 = v1; val2 = v2;}
    public static void main(String [ ] args) {
        ShowObject theObj = new ShowObject(24, 12);
        System.out.println("the output is "+theObj);
    }
}
```

Class Object

Method toString( )

The default method toString( ) contained in class Object returns a String giving

- the name of the Class and
- the address of the Object in memory.

Consider the following example

```
class ShowObject {
    private int val1, val2;
    public ShowObject(int v1, int v2) {val1 = v1; val2 = v2;}
    public static void main(String [ ] args) {
        ShowObject theObj = new ShowObject(24, 12);
        System.out.println("the output is "+theObj);
    }
}
```

the output is ShowObject@107077e

Class Object

Class Object

Method toString( )

Class Object

Method toString( )

Note in the previous example that the call to

System.out.println(“some string ”+ theObj)

Results in a call to the method toString( ) in class ShowObject that is inherited from Object.

Class Object

Method toString( )

Note in the previous example that the call to

System.out.println("some string "+ theObj)

Results in a call to the method toString( ) in class ShowObject that is inherited from Object.

System.out.println(theObj); and

System.out.println(theObj.toString( )); are equivalent

Class Object

Method toString( )

Note in the previous example that the call to

System.out.println(“some string ”+ theObj)

Results in a call to the method toString( ) in class ShowObject that is inherited from Object.

System.out.println(theObj); and

System.out.println(theObj.toString( )); are equivalent

Whenever methods print( ) or println( ) are passed an Object as an argument, they will use the object’s toString( ) method to formulate the output string.

Class Object

Method hashCode( )

Class Object

Class Object

Method hashCode( )

Class Object

Method hashCode( )

Method hashCode( ) uses properties of an object to determine a (nearly) unique integer value.

Class Object

Method hashCode( )

Method hashCode( ) uses properties of an object to determine a (nearly) unique integer value.

Two objects with the same values will have different hash codes. Consider class ShowObject described on the previous slide. If we declare two objects of this class with identical values, we will obtain different hash codes.

Class Object

Method hashCode( )

Method hashCode( ) uses properties of an object to determine a (nearly) unique integer value.

Two objects with the same values will have different hash codes. Consider class ShowObject described on the previous slide. If we declare two objects of this class with identical values, we will obtain different hash codes.

```
pubic static void main (String [ ] args) {
    ShowObject obj1 = new ShowObject(24, 12);
    ShowObject obj2 = new ShowObject(24, 12);
    System.out.println("hash code for obj1: "+obj1.hashCode( ));
    System.out.println("hash code for obj2: "+obj2.hashCode( ));
}
```

Class Object

Method hashCode( )

Method hashCode( ) uses properties of an object to determine a (nearly) unique integer value.

Two objects with the same values will have different hash codes. Consider class ShowObject described on the previous slide. If we declare two objects of this class with identical values, we will obtain different hash codes.

```
pubic static void main (String [ ] args) {
    ShowObject obj1 = new ShowObject(24, 12);
    ShowObject obj2 = new ShowObject(24, 12);
    System.out.println("hash code for obj1: "+obj1.hashCode( ));
    System.out.println("hash code for obj2: "+obj2.hashCode( ));
}
```

8187137

Class Object

Method hashCode( )

Method hashCode( ) uses properties of an object to determine a (nearly) unique integer value.

Two objects with the same values will have different hash codes. Consider class ShowObject described on the previous slide. If we declare two objects of this class with identical values, we will obtain different hash codes.

```
pubic static void main (String [ ] args) {
    ShowObject obj1 = new ShowObject(24, 12);
    ShowObject obj2 = new ShowObject(24, 12);
    System.out.println("hash code for obj1: "+obj1.hashCode( ));        8187137
    System.out.println("hash code for obj2: "+obj2.hashCode( ));        28050664
}
```

Class Object

Class Object

Method hashCode( )

Class Object

Method hashCode( )

A good hash function is one that uniformly distributes the keys into the range of integer values that index the hash table. For most applications, the programmer will need to override hashCode( ) derived from the base class Object.

If the programmer chooses to override method hashCode( ) in a particular class, he or she must supply

1. **Supply a (private) hash function that maps attributes of the object into integer values.**
2. **Choose some attribute or attributes of the class that form the domain of the hash function.**

Class Object

Method clone( )

Class Object

Class Object

Method clone( )

Class Object

Method clone( )

Not all classes can be cloned! clone( ) is a protected method in Base class Object, and hence is not directly accessible to a client application.

Class Object

Method clone( )

Not all classes can be cloned! clone( ) is a protected method in Base class Object, and hence is not directly accessible to a client application.

Protected methods are accessible to derived classes and hence may be overridden and made public.

Class Object

Method clone( )

Not all classes can be cloned! clone( ) is a protected method in Base class Object, and hence is not directly accessible to a client application.

Protected methods are accessible to derived classes and hence may be overridden and made public.

- **An application can create a clone of some object only if it has access to the clone( ) method in the class to which that object belongs.**

Class Object

Method clone( )

Not all classes can be cloned! clone( ) is a protected method in Base class Object, and hence is not directly accessible to a client application.

Protected methods are accessible to derived classes and hence may be overridden and made public.

- **An application can create a clone of some object only if it has access to the clone( ) method in the class to which that object belongs.**
- **Unless method clone( ) is overriden, a derived class will inherit clone( ) from Object, but the method is protected and not accessible to a client – therefore no clones of objects of that class can be created.**

Class Object

Method clone( )

Not all classes can be cloned! clone( ) is a protected method in Base class Object, and hence is not directly accessible to a client application.

Protected methods are accessible to derived classes and hence may be overridden and made public.

- **An application can create a clone of some object only if it has access to the clone( ) method in the class to which that object belongs.**
- **Unless method clone( ) is overriden, a derived class will inherit clone( ) from Object, but the method is protected and not accessible to a client – therefore no clones of objects of that class can be created.**
- **To allow clients to create clones of objects of a particular class, method clone( ) will have to be overridden and made public.**

Class Object

Method clone( )

Not all classes can be cloned! clone( ) is a protected method in Base class Object, and hence is not directly accessible to a client application.

Protected methods are accessible to derived classes and hence may be overridden and made public.

- **An application can create a clone of some object only if it has access to the clone( ) method in the class to which that object belongs.**
- **Unless method clone( ) is overriden, a derived class will inherit clone( ) from Object, but the method is protected and not accessible to a client – therefore no clones of objects of that class can be created.**
- **To allow clients to create clones of objects of a particular class, method clone( ) will have to be overridden and made public.**
- **Once clone( ) is made public in a class, it is public also for any classes derived from that class.**

Class Object

Method clone( )

There are three important rules to follow if you are going to override clone( ) in a class you are creating.

1. (Virtually) Always call super.clone( ) -- the base class clone( ) method performs the bitwise duplication of the derived class object. (If the attributes are all primitive types it is safe to not call super.)

2. Make your clone( ) method public

3. Implement the Cloneable interface

   •To determine whether an instance of an object can be cloned

   if (myReference instanceof Cloneable) {..}

   •Cloneable is an "empty interface" with no methods to implement. It acts as a "flag" identifying that an object can be cloned.

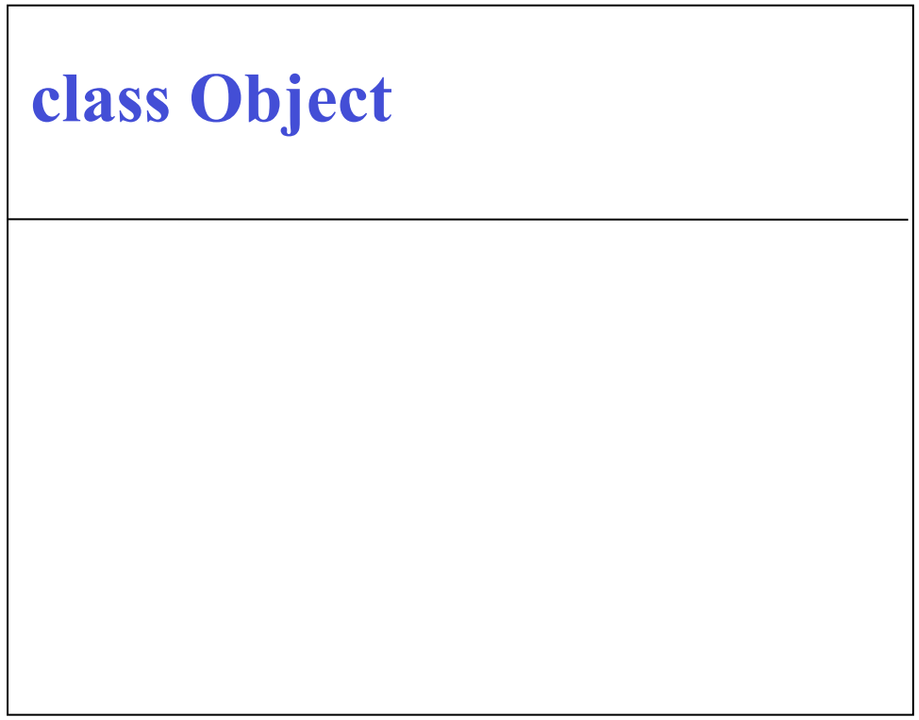
Class Object

Class Object

Method clone( )

Class Object

Method clone( )



Class Object

Method clone( )

class Object

class MyThing extends Object

implements Cloneable

Class Object

Method clone( )

class Object

class MyThing extends Object

implements Cloneable

//data attributes

private int myInt; //primitive type

private YourThing aThing; //object

Class Object

Method clone( )

```
class Object
```

```
protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

```
class MyThing extends Object
implements Cloneable
```

```
//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object
```

Class Object

Method clone( )

```
class Object
```

```
protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

Override method clone( ) and make it public

```
class MyThing extends Object
implements Cloneable
```

```
//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object
```

Class Object

Method clone( )

```
class Object
```

```
protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

Override method clone( ) and make it public

```
class MyThing extends Object
implements Cloneable
```

```
//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object
```

```
public Object clone ( ) {
```

Class Object

Method clone( )

```
class Object

protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

Declare a *MyThing* object that will become the clone you return

```
class MyThing extends Object
implements Cloneable

//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object

public Object clone ( ) {
```

Class Object

Method clone( )

```
class Object

protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

Declare a *MyThing* object that will become the clone you return

```
class MyThing extends Object
implements Cloneable

//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object

public Object clone ( ) {
    MyThing result = null;
```

Class Object

Method clone( )

```
class Object

protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

```
class MyThing extends Object
implements Cloneable

//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object

public Object clone ( ) {
    MyThing result = null;
    result = (MyThing)super.clone( );
```

Class Object

Method clone( )

```
class Object
protected Object clone( ) {
   //perform bitwise copy
}
//other methods
```

A call to clone( ) in the base class makes a bitwise copy of the attributes of the MyThing object. Method clone() in the base class is protected—hence accessible to derived classes.

```
class MyThing extends Object
implements Cloneable

//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object

public Object clone ( ) {
    MyThing result = null;
    result = (MyThing)super.clone( );
```

Class Object

Method clone( )

```
class Object

protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

A cast is performed to down cast the Object returned by the clone( ) method in Object to a MyThing object before assigning it to result.

```
class MyThing extends Object
implements Cloneable

//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object

public Object clone ( ) {
    MyThing result = null;
    result = (MyThing)super.clone( );
```

Class Object

Method clone( )

```
class Object

protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

```
class MyThing extends Object
implements Cloneable

//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object
```

```
public Object clone ( ) {
    MyThing result = null;
    result = (MyThing)super.clone( );
    result.aThing = (YourThing)
            result.aThing.clone( );
    return result;
}
```

Class Object

Method clone( )

```
class Object

protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

```
class MyThing extends Object
implements Cloneable
```

```
//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object
```

```
public Object clone ( ) {
    MyThing result = null;
    result = (MyThing)super.clone( );
    result.aThing = (YourThing)
            result.aThing.clone( );
    return result;
}
```

Here we assume the object attribute is Cloneable, and we convert the bitwise copy done in the call to super – that only copies a handle to an object – into a "deep copy" of the object itself by calling the object's own clone( ) method.

Class Object

Method clone( )

```
class Object

protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

```
class MyThing extends Object
implements Cloneable
```

```
//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object
```

```
public Object clone ( ) {
    MyThing result = null;
    result = (MyThing)super.clone( );
    result.aThing = (YourThing)
            result.aThing.clone( );
    return result;
}

//other methods
```

Here we assume the object attribute is Cloneable, and we convert the bitwise copy done in the call to super – that only copies a handle to an object – into a "deep copy" of the object itself by calling the object's own clone( ) method.

Class Object

Class Object

Method clone( )

Class Object

Method clone( )

Some frequently used Java library classes are not cloneable.

Class Object

Method clone( )

Some frequently used Java library classes are not cloneable.

- **The wrapper classes (Integer, Double, etc.) do not override method clone( ) and hence cannot be cloned.**

Class Object

Method clone( )

Some frequently used Java library classes are not cloneable.

- **The wrapper classes (Integer, Double, etc.) do not override method clone( ) and hence cannot be cloned.**
- **String and StringBuffer classes are not Cloneable.**

Class Object

Method clone( )

Some frequently used Java library classes are not cloneable.

- **The wrapper classes (Integer, Double, etc.) do not override method clone( ) and hence cannot be cloned.**

- **String and StringBuffer classes are not Cloneable.**

- **Standard Containers, such as *arrayList*, can be cloned, but a shallow copy of their contents is made because it cannot be certain that those contained objects can be cloned.**

Class Object

Class Object

Method clone( ) – an example

Class Object

Method clone( ) – an example

A Box is an unordered collection of objects of fixed size.

Class Object

Method clone( ) – an example

A Box is an unordered collection of objects of fixed size.

```
public class Box implements Cloneable {
    protected Object [ ] box;
    protected int size;
    public Box(int bsize) {
        box = new Object[bsize];
        size = 0;
    }
    public Object clone ( ) {
        Box  result = null;
        try {
            result = (Box)super.clone( );
        } catch (CloneNotSupportedException e) {System.out.println(e); }
        for (int i = 0; i < box.length; i++)  //override bitwise copy if possible
            if (box[i]  instanceof Cloneable)       { result.box[i] = box[i].clone( ); }
        return result;
    }
}
```

Class Object

Method clone( ) – an example

A Box is an unordered collection of objects of fixed size.

```
public class Box implements Cloneable {
    protected Object [ ] box;
    protected int size;
    public Box(int bsize) {
        box = new Object[bsize];
        size = 0;
    }
    public Object clone ( ) {
        Box  result = null;
        try {
            result = (Box)super.clone( );
        } catch (CloneNotSupportedException e) {System.out.println(e); }
        for (int i = 0; i < box.length; i++)  //override bitwise copy if possible
            if (box[i]  instanceof Cloneable)       { result.box[i] = box[i].clone( ); }
        return result;
    }
}
```

You need to enclose the call to super.clone inside a try block

Class Object

Method getClass( )

Class Object

Class Object

Method getClass( )

Class Object

Method getClass( )

This method is useful when you have a container holding objects of a base class (such as Shape) some or all of which are instances of a derived class (such as Circle or Rectangle). The method getClass( ) allows one to determine which kind of an object each is, and determine whether a particular down cast is appropriate.

This method is inherited from the base class Object and should never be overridden in any derived class.

Class Object

Class Object

Method getClass( ) -- Example

Class Object

Method getClass( ) -- Example

Let Shape be an abstract class and Circle and Rectangle concrete classes that extend Shape.

Class Object

Method getClass( ) -- Example

Let Shape be an abstract class and Circle and Rectangle concrete classes that extend Shape.

```
public static void main (String [ ] args) {
    Shape [ ] shapeList = new Shape [3];
    shapeList[0] = new Circle(5.0);
    shapeList[1] = new Rectangle(3.0, 4.0);
    shapeList[2] = new Circle(6.0);
    for (int i = 0; i < shapeList.length; i++) {
        Class c = shapeList[i].getClass( );
        if (c.equals(Circle.class))
            System.out.println("I can be cast as a Circle");
        else if (c.equals(Rectangle.class))
            System.out.println("I can be cast as a Rectangle");
    }
}
```

Class Object

Method getClass( ) -- Example

Let Shape be an abstract class and Circle and Rectangle concrete classes that extend Shape.

```
public static void main (String [ ] args) {
    Shape [ ] shapeList = new Shape [3];
    shapeList[0] = new Circle(5.0);
    shapeList[1] = new Rectangle(3.0, 4.0);
    shapeList[2] = new Circle(6.0);
    for (int i = 0; i < shapeList.length; i++) {
        Class c = shapeList[i].getClass( );
        if (c.equals(Circle.class))
            System.out.println("I can be cast as a Circle");
        else if (c.equals(Rectangle.class))
            System.out.println("I can be cast as a Rectangle");
    }
}
```

Find out to which class each object in shapeList belongs

Class Object

Method getClass( ) -- Example

Let Shape be an abstract class and Circle and Rectangle concrete classes that extend Shape.

```
public static void main (String [ ] args) {
    Shape [ ] shapeList = new Shape [3];
    shapeList[0] = new Circle(5.0);
    shapeList[1] = new Rectangle(3.0, 4.0);
    shapeList[2] = new Circle(6.0);
    for (int i = 0; i < shapeList.length; i++) {
        Class c = shapeList[i].getClass( );
        if (c.equals(Circle.class))
            System.out.println("I can be cast as a Circle");
        else if (c.equals(Rectangle.class))
            System.out.println("I can be cast as a Rectangle");
    }
}
```

Find out to which class each object in shapeList belongs

Knowing the class that an object belongs to allows the client programmer to down cast the object and send messages to methods particular to that derived class.

RTTI

The idea of run-time type identification (RTTI) seems fairly simple at first: it lets you find the exact type of an object when you have a handle to only the base type.

Class object

- The Class object is used to create all of the "regular" objects of your class. Each time you write a new class, a single Class object is also created (and stored, appropriately enough, in an identically named .class file). At run time, when you want to make an object of that class, the Java Virtual Machine (JVM) that's executing your program first checks to see if the Class object for that type is loaded. If not, the JVM loads it by finding the .class file with that name.
- Case Study: SweetShop.java

- The output of this program for one JVM is:
  - inside main Loading Candy
  - After creating Candy
  - Loading Gum
  - After Class.forName("Gum")
  - Loading Cookie
  - After creating Cookie

The Class Object

- Class literals also provide a reference to the Class object
  - *E.g.* `Gum.class`
- Each object of a primitive wrapper class has a standard field called TYPE that also provides a reference to the Class object
  - http://java.sun.com/j2se/1.3/docs/api/java/lang/Boolean.html

Instance of

- Java 1.1 has added the isInstance method to the class Class.
- Case Study: PetCount3.java

Class Object

Class Object

Method getClass( )

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Let x be an object of one of the derived classes in the previous example. Then we may determine the class of this object by:

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Let x be an object of one of the derived classes in the previous example. Then we may determine the class of this object by:

```
Class c = x.getClass( );    if (c.equals(Circle.class)){ }
```

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Let x be an object of one of the derived classes in the previous example. Then we may determine the class of this object by:

```
Class c = x.getClass( );    if (c.equals(Circle.class)){ }
```

⟵ Compares 2 classes

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Let x be an object of one of the derived classes in the previous example. Then we may determine the class of this object by:

```
Class c = x.getClass( );    if (c.equals(Circle.class)){ }
```

← Compares 2 classes

Tests whether references are the same -- c == Circle.class is equivalent

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Let x be an object of one of the derived classes in the previous example. Then we may determine the class of this object by:

```
Class c = x.getClass( );    if (c.equals(Circle.class)){ }
```

← Compares 2 classes

Tests whether references are the same -- c == Circle.class is equivalent

```
if (x instanceof Circle) { }
```

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Let x be an object of one of the derived classes in the previous example. Then we may determine the class of this object by:

```
Class c = x.getClass( );    if (c.equals(Circle.class)){ }
```

← Compares 2 classes

Tests whether references are the same -- c == Circle.class is equivalent

```
if (x instanceof Circle) { }
```

← Tests whether object x is an instance of a class

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Let x be an object of one of the derived classes in the previous example. Then we may determine the class of this object by:

```
Class c = x.getClass( );    if (c.equals(Circle.class)){ }
```

← Compares 2 classes

Tests whether references are the same -- c == Circle.class is equivalent

```
if (x instanceof Circle) { }
```

← Tests whether object x is an instance of a class

instanceof is a comparator (an operator)

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Let x be an object of one of the derived classes in the previous example. Then we may determine the class of this object by:

```
Class c = x.getClass( );    if (c.equals(Circle.class)){ }
```

← Compares 2 classes

Tests whether references are the same -- c == Circle.class is equivalent

```
if (x instanceof Circle) { }
```

← Tests whether object x is an instance of a class

instanceof is a comparator (an operator)

```
if (c.isInstance(x) ) { }
```

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Let x be an object of one of the derived classes in the previous example. Then we may determine the class of this object by:

```
Class c = x.getClass( );    if (c.equals(Circle.class)){ }
```

← Compares 2 classes

Tests whether references are the same -- c == Circle.class is equivalent

```
if (x instanceof Circle) { }
```

← Tests whether object x is an instance of a class

instanceof is a comparator (an operator)

```
if (c.isInstance(x) ) { }
```

← Alternatively (but equivalently) use the isInstance( ) method of an object with a class parameter

RTTI mothods

- **Case Study: ToyTest.java**

Reflection: run-time class information

- Where to use Reflection?
  - Component-based programming in which you build projects using Rapid Application Development (RAD) in an application builder tool.
    - To provide the ability to create and execute objects on remote platforms across a network. This is called Remote Method Invocation (RMI) and it allows a Java program (version 1.1 and higher) to have objects distributed across many machines.
- Case Study: ShowMethods.java

Get Filed Value

To print the names of the ViolinNote fields as well as their current values in the particular ViolinNote object referenced by note:

```
Field fields[] = c.getFields(); try {
 for(int i = 0; i < fields.length; i++) {
  System.out.print(fields[i].getName() + " =
  ");
  System.out.println(fields[i].getInt(note))
  ;
 }
} catch(Exception e) {
}
```

Invoke Method

```
note = new Note();
c = note.getClass();
Method meth = c.getMethod("play",
null);
meth.invoke(note, null);
```

串行化和RTTI

- 反串行化回来的对象为什么还是能用的?