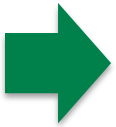
CS 10:
Problem solving via Object Oriented Programming

Animated Blobs

Agenda



1. Objects vs. primitives
2. Inheritance
3. Quick review of Blobs
4. Using Inheritance to create different types of Blobs
5. Graphical User Interface (GUI) programming

Java defines a number of primitive types, each of fixed memory size

Common primitive types

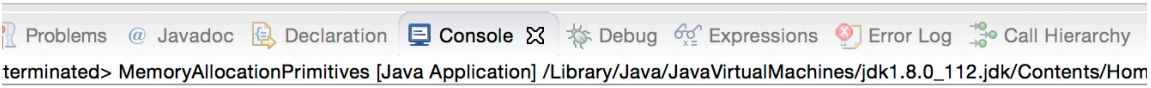
Type	Description	Size	Examples
int	Integer values (no decimal component)	32 bits (4 bytes)	-104,…1,2,3…107,… 5032..
double	Double precision floating point (has decimal component)	64 bits (8 bytes)	-123.45, 1.6
boolean	true or false	1 bit	true, false
char	Characters	16 bits (2 bytes for Unicode)	‘a’,’b’,…’z’

NOTE: Java provides an Object wrapper for each primitive (called autoboxing). Reference them with an initial capital letter (e.g., Integer, Double, Boolean, Character)

Declaring a primitive variable allocates stack space that holds variable's value

Stack **Heap**

```
 6 public class MemoryAllocationPrimitives {
 7
 8     public static void main(String[] args) {
 9         //declare local variables
10         int i; double d; boolean b; char c;
11
12         //assign values to local variables
13         i=7; d=1.6; b=true; c='a';
14
15         //print new values
16         System.out.println("Local variables: "+
17                 "i="+i+" d="+d+" b="+b+" c="+c);
18     }
19 }
```

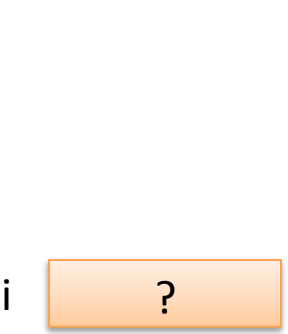


Declaring a primitive variable allocates stack space that holds variable's value

```
 6 public class MemoryAllocationPrimitives {
 7
 8     public static void main(String[] args) {
 9         //declare local variables
10         int i; double d; boolean b; char c;
11
12         //assign values to local variables
13         i=7; d=1.6; b=true; c='a';
14
15         //print new values
16         System.out.println("Local variables: "+
17                 "i="+i+" d="+d+" b="+b+" c="+c);
18     }
19 }
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom



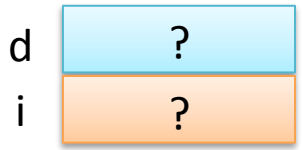
- While executing line 10, space is allocated on the stack for the primitive local variables
- Java doesn't initialize local variables (like it does instance variables)
- Exception (error) raised if try to use a local variable before value assigned

Declaring a primitive variable allocates stack space that holds variable's value

```
 6 public class MemoryAllocationPrimitives {
 7
 8     public static void main(String[] args) {
 9         //declare local variables
10         int i; double d; boolean b; char c;
11
12         //assign values to local variables
13         i=7; d=1.6; b=true; c='a';
14
15         //print new values
16         System.out.println("Local variables: "+
17                 "i="+i+" d="+d+" b="+b+" c="+c);
18     }
19 }
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom



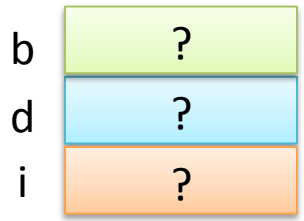
- While executing line 10, space is allocated on the stack for the primitive local variables
- Java doesn't initialize local variables (like it does instance variables)
- Exception (error) raised if try to use a local variable before value assigned
- NOTE: showing primitive types as same size for convenience

Declaring a primitive variable allocates stack space that holds variable's value

```
 6 public class MemoryAllocationPrimitives {
 7
 8     public static void main(String[] args) {
 9         //declare local variables
10         int i; double d; boolean b; char c;
11
12         //assign values to local variables
13         i=7; d=1.6; b=true; c='a';
14
15         //print new values
16         System.out.println("Local variables: "+
17                 "i="+i+" d="+d+" b="+b+" c="+c);
18     }
19 }
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom



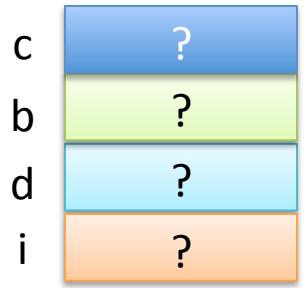
- While executing line 10, space is allocated on the stack for the primitive local variables
- Java doesn't initialize local variables (like it does instance variables)
- Exception (error) raised if try to use a local variable before value assigned
- NOTE: showing primitive types as same size for convenience

Declaring a primitive variable allocates stack space that holds variable's value

```
 6 public class MemoryAllocationPrimitives {
 7
 8     public static void main(String[] args) {
 9         //declare local variables
10         int i; double d; boolean b; char c;
11
12         //assign values to local variables
13         i=7; d=1.6; b=true; c='a';
14
15         //print new values
16         System.out.println("Local variables: "+
17                 "i="+i+" d="+d+" b="+b+" c="+c);
18     }
19 }
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom



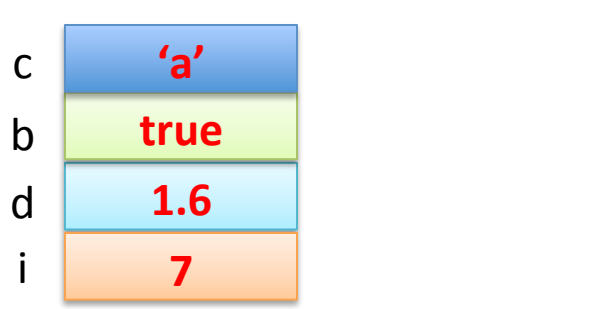
- While executing line 10, space is allocated on the stack for the primitive local variables
- Java doesn't initialize local variables (like it does instance variables)
- Exception (error) raised if try to use a local variable before value assigned
- NOTE: showing primitive types as same size for convenience

Declaring a primitive variable allocates stack space that holds variable's value

```
 6 public class MemoryAllocationPrimitives {
 7
 8     public static void main(String[] args) {
 9         //declare local variables
10         int i; double d; boolean b; char c;
11
12         //assign values to local variables
13         i=7; d=1.6; b=true; c='a';
14
15         //print new values
16         System.out.println("Local variables: "+
17                 "i="+i+" d="+d+" b="+b+" c="+c);
18     }
19 }
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom



- **After executing line 13, values assigned to primitive local variables**

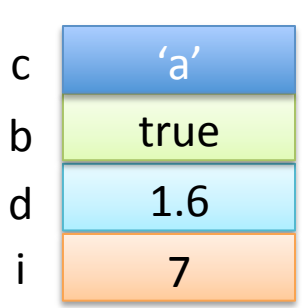
Declaring a primitive variable allocates stack space that holds variable's value

```
 6 public class MemoryAllocationPrimitives {
 7
 8     public static void main(String[] args) {
 9         //declare local variables
10         int i; double d; boolean b; char c;
11
12         //assign values to local variables
13         i=7; d=1.6; b=true; c='a';
14
15         //print new values
16         System.out.println("Local variables: "+
17                 "i="+i+" d="+d+" b="+b+" c="+c);
18     }
19 }
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom

Local variables: i=7 d=1.6 b=true c=a



- **Stack holds the *values* of the primitive data types**
- **Printing a primitive type prints its value**

Declaring objects makes pointer on the stack, but object itself is elsewhere

```
 8    public static void main(String[] args) {
 9        //declare Blob objects
10        Blob alice = new Blob();
11        Blob bob; //notice no new keyword
12        bob = alice; //bob equals alice
13        Blob charlie = new Blob();
14        System.out.println("alice.x="+alice.x+
15                " bob.x="+bob.x);
16
17        //update alice's x
18        alice.setX(3);
19        System.out.println("alice.x="+alice.x+
20                " bob.x="+bob.x);
21
22        //printing objects implicitly calls toString()
23        System.out.println("alice="+alice+
24                " bob="+bob+" charlie="+charlie);
25    }
26 }
```

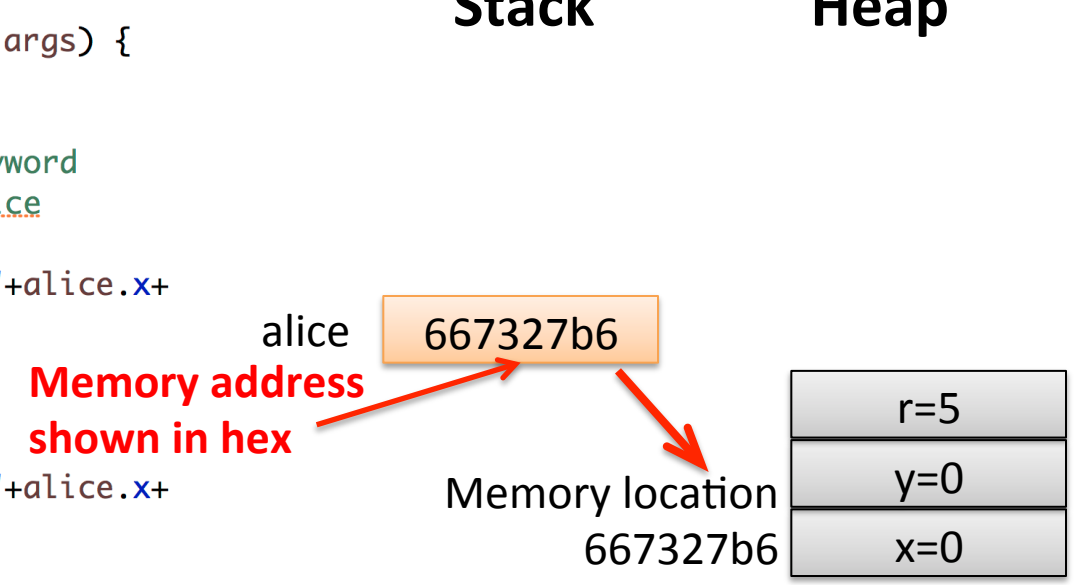
Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

Stack **Heap**

Declaring objects makes pointer on the stack, but object itself is elsewhere

```
 8    public static void main(String[] args) {
 9        //declare Blob objects
10        Blob alice = new Blob();
11        Blob bob; //notice no new keyword
12        bob = alice; //bob equals alice
13        Blob charlie = new Blob();
14        System.out.println("alice.x="+alice.x+
15                " bob.x="+bob.x);
16
17        //update alice's x
18        alice.setX(3);
19        System.out.println("alice.x="+alice.x+
20                " bob.x="+bob.x);
21
22        //printing objects implicitly calls toString()
23        System.out.println("alice="+alice+
24                " bob="+bob+" charlie="+charlie);
25    }
26 }
```

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

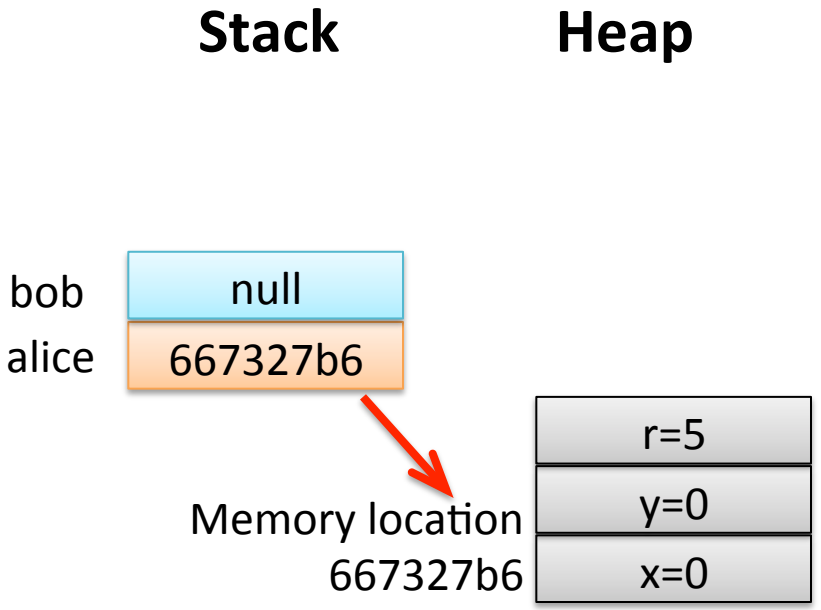


- After line 10, stack holds ***memory address*** of object (with primitives, stack holds variable's *value*)
- Memory address tells Java where to find the "alice" object in memory
- Object itself allocated elsewhere in memory (in heap, not on stack)
- OS chooses where to allocate

Declaring objects makes pointer on the stack, but object itself is elsewhere

```
 8    public static void main(String[] args) {
 9        //declare Blob objects
10        Blob alice = new Blob();
11        Blob bob; //notice no new keyword
12        bob = alice; //bob equals alice
13        Blob charlie = new Blob();
14        System.out.println("alice.x="+alice.x+
15                " bob.x="+bob.x);
16
17        //update alice's x
18        alice.setX(3);
19        System.out.println("alice.x="+alice.x+
20                " bob.x="+bob.x);
21
22        //printing objects implicitly calls toString()
23        System.out.println("alice="+alice+
24                " bob="+bob+" charlie="+charlie);
25    }
26 }
```

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

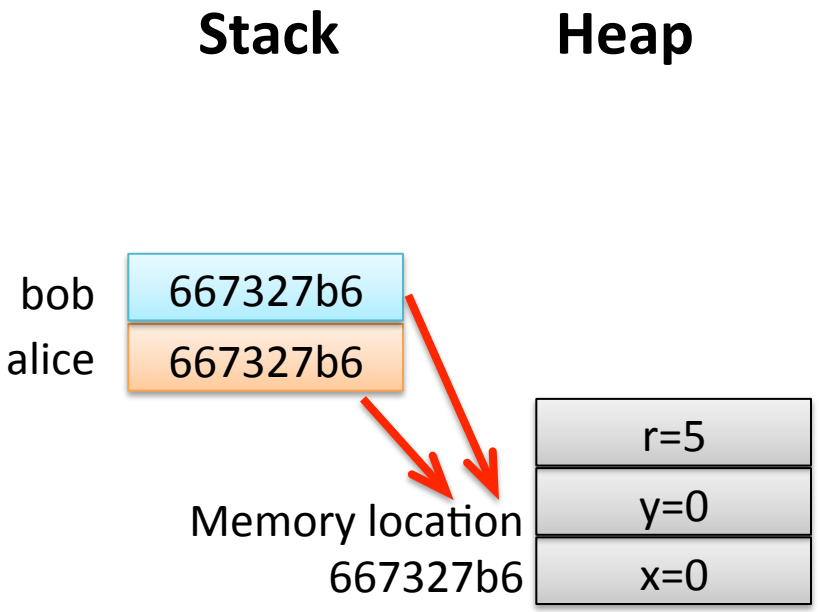


- After line 11, "bob" is allocated on the stack, but is null (points nowhere)
- This is because bob did not use the "new" keyword
- Null pointer exception if try to use bob now

Declaring objects makes pointer on the stack, but object itself is elsewhere

```
 8    public static void main(String[] args) {
 9        //declare Blob objects
10        Blob alice = new Blob();
11        Blob bob; //notice no new keyword
12        bob = alice; //bob equals alice
13        Blob charlie = new Blob();
14        System.out.println("alice.x="+alice.x+
15                " bob.x="+bob.x);
16
17        //update alice's x
18        alice.setX(3);
19        System.out.println("alice.x="+alice.x+
20                " bob.x="+bob.x);
21
22        //printing objects implicitly calls toString()
23        System.out.println("alice="+alice+
24                " bob="+bob+" charlie="+charlie);
25    }
26 }
```

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

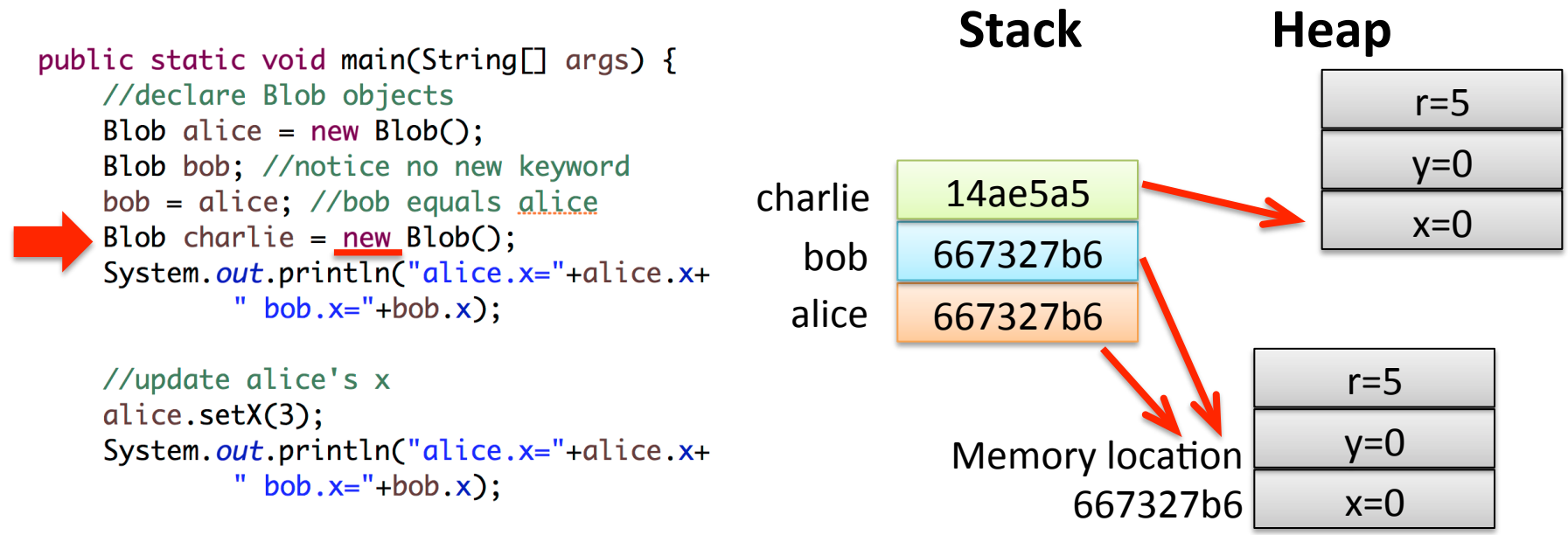


- **Line 12, bob set equal to alice**
- **bob gets same value on stack that alice holds**
- **bob now points to the exact same memory location as alice**
- **bob and alice are "aliases" of each other**

Declaring objects makes pointer on the stack, but object itself is elsewhere

```
 8    public static void main(String[] args) {
 9        //declare Blob objects
10        Blob alice = new Blob();
11        Blob bob; //notice no new keyword
12        bob = alice; //bob equals alice
13        Blob charlie = new Blob();
14        System.out.println("alice.x="+alice.x+
15                " bob.x="+bob.x);
16
17        //update alice's x
18        alice.setX(3);
19        System.out.println("alice.x="+alice.x+
20                " bob.x="+bob.x);
21
22        //printing objects implicitly calls toString()
23        System.out.println("alice="+alice+
24                " bob="+bob+" charlie="+charlie);
25    }
26 }
```

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy



Charlie object gets new allocation elsewhere in memory because "new" keyword used

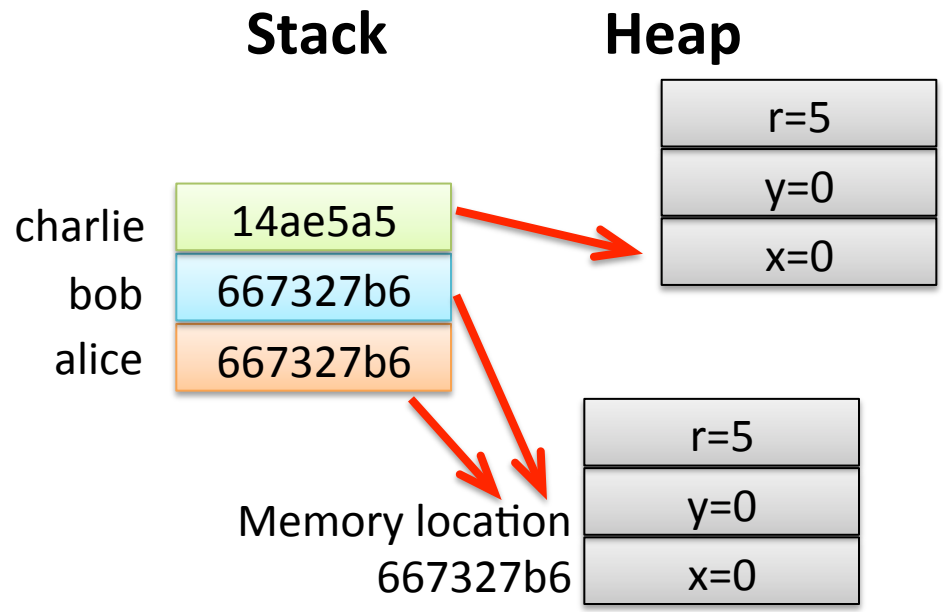
Declaring objects makes pointer on the stack, but object itself is elsewhere

```
 8    public static void main(String[] args) {
 9        //declare Blob objects
10        Blob alice = new Blob();
11        Blob bob; //notice no new keyword
12        bob = alice; //bob equals alice
13        Blob charlie = new Blob();
14        System.out.println("alice.x="+alice.x+
15                " bob.x="+bob.x);
16
17        //update alice's x
18        alice.setX(3);
19        System.out.println("alice.x="+alice.x+
20                " bob.x="+bob.x);
21
22        //printing objects implicitly calls toString()
23        System.out.println("alice="+alice+
24                " bob="+bob+" charlie="+charlie);
25    }
26 }
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> MemoryAllocationObjects [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec

```
alice.x=0.0 bob.x=0.0
```



x value for alice and bob is the same because stored at the exact same memory address

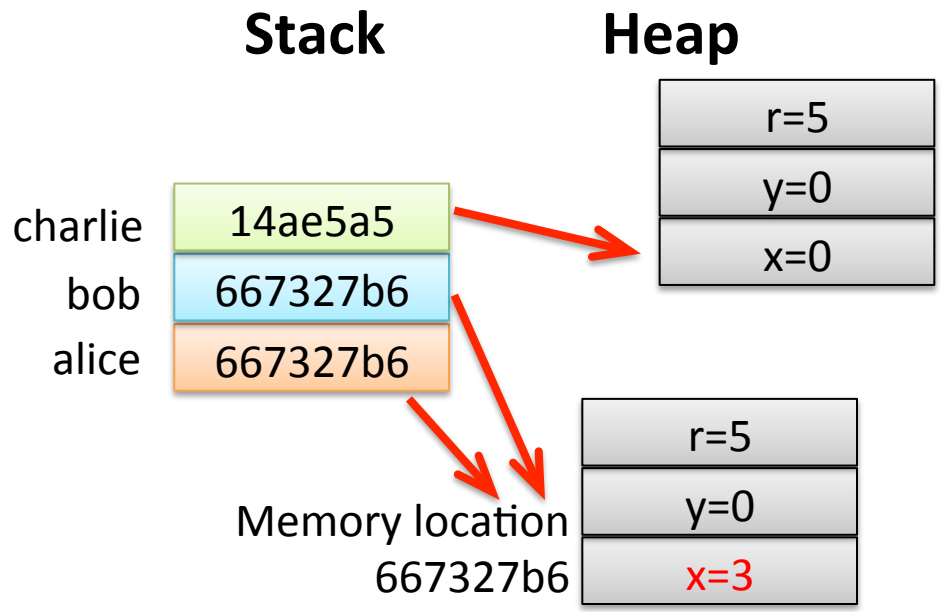
Declaring objects makes pointer on the stack, but object itself is elsewhere

```
 8    public static void main(String[] args) {
 9        //declare Blob objects
10        Blob alice = new Blob();
11        Blob bob; //notice no new keyword
12        bob = alice; //bob equals alice
13        Blob charlie = new Blob();
14        System.out.println("alice.x="+alice.x+
15                " bob.x="+bob.x);
16
17        //update alice's x
18        alice.setX(3);
19        System.out.println("alice.x="+alice.x+
20                " bob.x="+bob.x);
21
22        //printing objects implicitly calls toString()
23        System.out.println("alice="+alice+
24                " bob="+bob+" charlie="+charlie);
25    }
26 }
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

<terminated> MemoryAllocationObjects [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec

```
alice.x=0.0 bob.x=0.0
```



- **alice.x set to 3**
- **What is bob.x?**

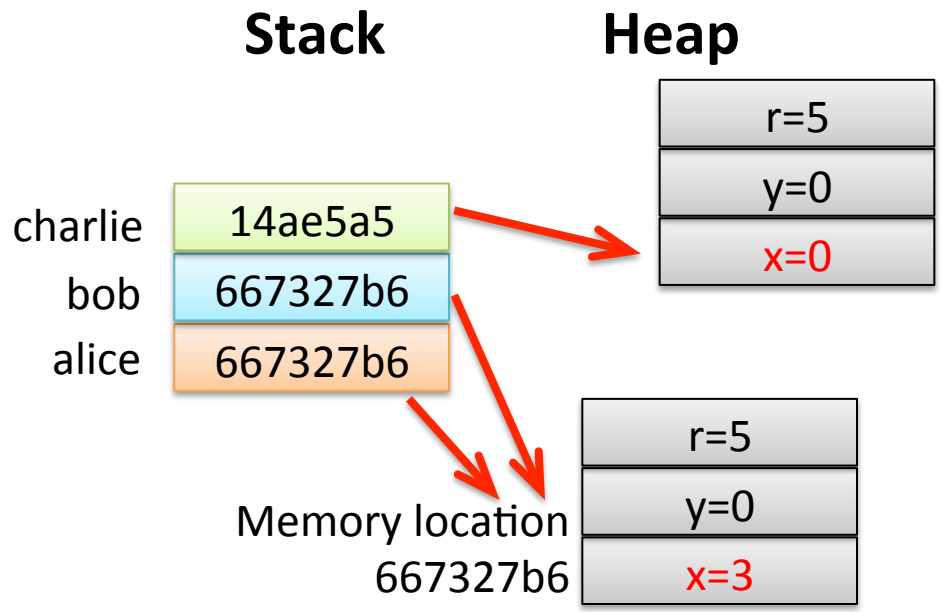
Declaring objects makes pointer on the stack, but object itself is elsewhere

```
 8    public static void main(String[] args) {
 9        //declare Blob objects
10        Blob alice = new Blob();
11        Blob bob; //notice no new keyword
12        bob = alice; //bob equals alice
13        Blob charlie = new Blob();
14        System.out.println("alice.x="+alice.x+
15                " bob.x="+bob.x);
16
17        //update alice's x
18        alice.setX(3);
19        System.out.println("alice.x="+alice.x+
20                " bob.x="+bob.x);
21
22        //printing objects implicitly calls toString()
23        System.out.println("alice="+alice+
24                " bob="+bob+" charlie="+charlie);
25    }
26 }
```

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

<terminated> MemoryAllocationObjects [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec

```
alice.x=0.0 bob.x=0.0
alice.x=3.0 bob.x=3.0
```



- **x is the same for both alice and bob objects because they point to the same memory address (called alias)**
- **Like Python setting two lists equal to each other, change one list, change the other also**
- **Charlie's x is still 0**

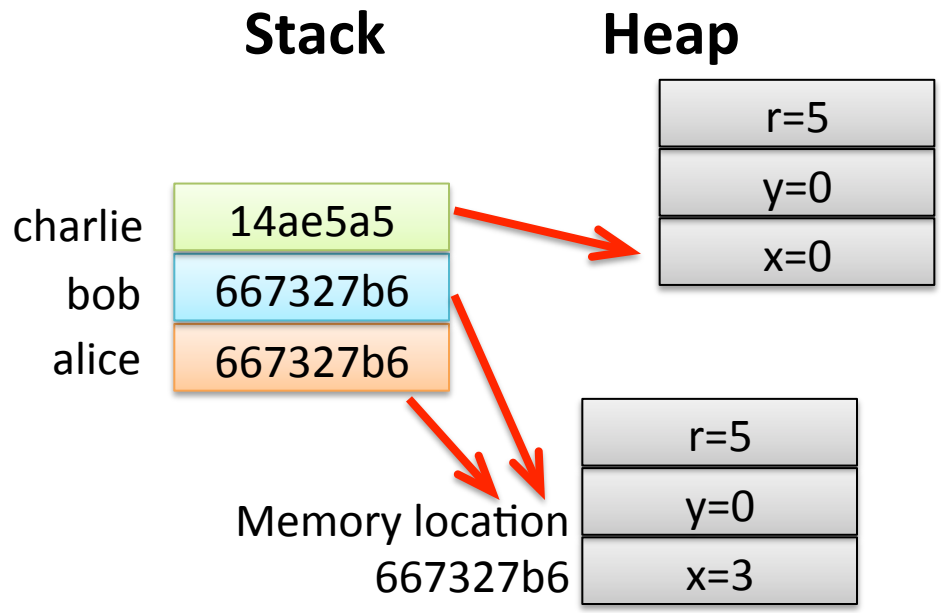
Declaring objects makes pointer on the stack, but object itself is elsewhere

```
 8    public static void main(String[] args) {
 9        //declare Blob objects
10        Blob alice = new Blob();
11        Blob bob; //notice no new keyword
12        bob = alice; //bob equals alice
13        Blob charlie = new Blob();
14        System.out.println("alice.x="+alice.x+
15                " bob.x="+bob.x);
16
17        //update alice's x
18        alice.setX(3);
19        System.out.println("alice.x="+alice.x+
20                " bob.x="+bob.x);
21
22        //printing objects implicitly calls toString()
23        System.out.println("alice="+alice+
24                " bob="+bob+" charlie="+charlie);
25    }
26 }
```

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

<terminated> MemoryAllocationObjects [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Dec

```
alice.x=0.0 bob.x=0.0
alice.x=3.0 bob.x=3.0
alice=Blob@677327b6 bob=Blob@677327b6 charlie=Blob@14ae5a5
```



- **Printing an object causes an implicit call to *"toString()"* function**
- **This can be overriden (see course webpage)**
- **By default *toString()* prints memory address of object (for primitives, value is printed)**

Agenda

1. Objects vs. primitives
2. Inheritance
3. Quick review of Blobs
4. Using Inheritance to create different types of Blobs
5. Graphical User Interface (GUI) programming

Inheritance allows us to "build upon" work we have already done

Instance variables
Engines, fuel capacity, top speed, stall speed

Base class
Airplane

Methods
Take off, land, climb, descend, turn

May have already created a class called "Airplane"

- **Defined instance variables common to all (or at least most) airplanes, such as "top speed" or "fuel capacity"**
- **Written methods for common actions such as "climb"**

Inheritance allows us to use that work, plus extend it to define "specialty" versions of the base class (base class is also called super class; I'll use both terms)

Subclasses get all the instance variables and methods of the base class (unless declared private in base class, then only the base class has access, not subclass)

Subclasses can also define new variables and methods, and can *override* the base class variables and methods with their own versions

Inheritance allows us to "build upon" work we have already done

Instance variables
Engines, fuel capacity, top speed, stall speed

Base class
Airplane

Methods
Take off, land, climb, descend, turn

Sub class
Fighter Jet

Sub class
Passenger Jet

Instance variables
Weapons layout, …

Methods
Take off, Fire weapon,…

Subclasses *inherit* variables and methods from base class (no need to rewrite)

Instance variables
Passenger capacity, …

Methods
Take off, Serve food

Inheritance allows us to "build upon" work we have already done

Instance variables
Engines, fuel capacity, top speed, stall speed

Methods
Take off, land, climb, descend, turn

Base class
Airplane

Sub class
Fighter Jet

Sub class
Passenger Jet

Instance variables
Weapons layout, …

Methods
Take off, Fire weapon,…

Subclasses *inherit* variables and methods from base class (no need to rewrite)

Can override base class with their own version

Instance variables
Passenger capacity, …

Methods
Take off, Serve food

Inheritance allows us to "build upon" work we have already done

Instance variables
Engines, fuel capacity, top speed, stall speed

Base class
Airplane

Methods
Take off, land, climb, descend, turn

Sub class
Fighter Jet

Sub class
Passenger Jet

Instance variables
Weapons layout, …

Methods
Take off, Fire weapon,…

Subclasses *inherit* variables and methods from base class (no need to rewrite)

Can override base class with their own version

Instance variables
Passenger capacity, …

Methods
Take off, Serve food

If not overriden, base class method will be used

Inheritance allows us to "build upon" work we have already done

Instance variables
Engines, fuel capacity, top speed, stall speed

Methods
Take off, land, climb, descend, turn

Base class
Airplane

Sub class
Fighter Jet

Sub class
Passenger Jet

Instance variables
Weapons layout, …

Methods
Take off, Fire weapon,…

Subclasses *inherit* variables and methods from base class (no need to rewrite)

Can override base class with their own version

Instance variables
Passenger capacity, …

Methods
Take off, Serve food

If overridden, most specific method will be used

Inheritance models "is a" relationships

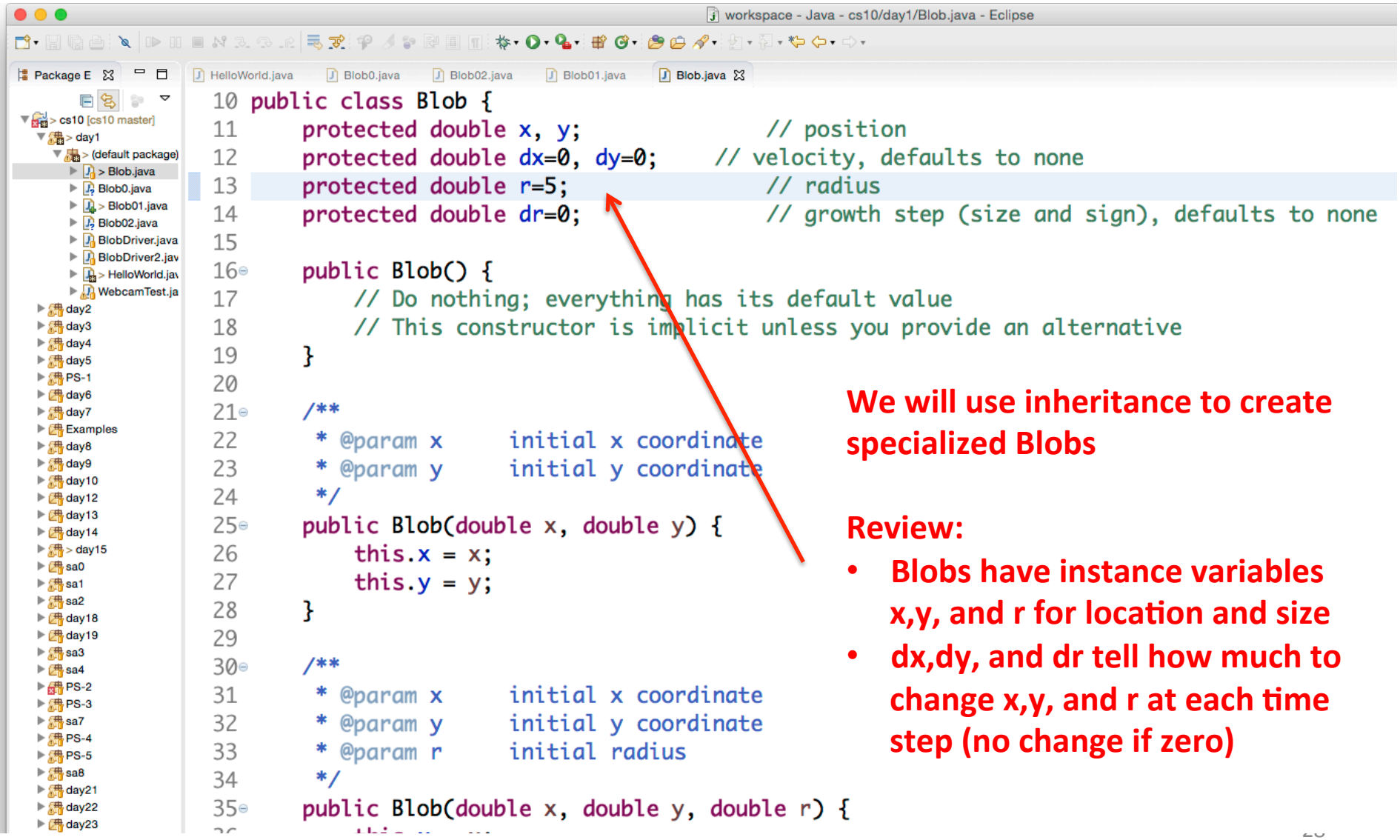
Inheritance rules

1. Every object of subclass "is a" object of base class
   - Every FighterJet "is a" Airplane
2. The set of objects of the subclass is a subset of the objects of base class
   - The *set* of FighterJets "is a" *subset* of Airplanes (there could be other types of airplanes)
   - Mathematically: $F \subseteq A$ where F = {FighterJets}, A = {Airplanes}
3. Every method that can be called on an object of base class, can also be called on an object of subclass
   - Results of the call may be very different if subclass overrides base class method, then subclass method is used
4. Objects of subclass are like objects of base class, but with additional specialization

Agenda

1. Objects vs. primitives
2. Inheritance
3. Quick review of Blobs
4. Using Inheritance to create different types of Blobs
5. Graphical User Interface (GUI) programming

Blob.java

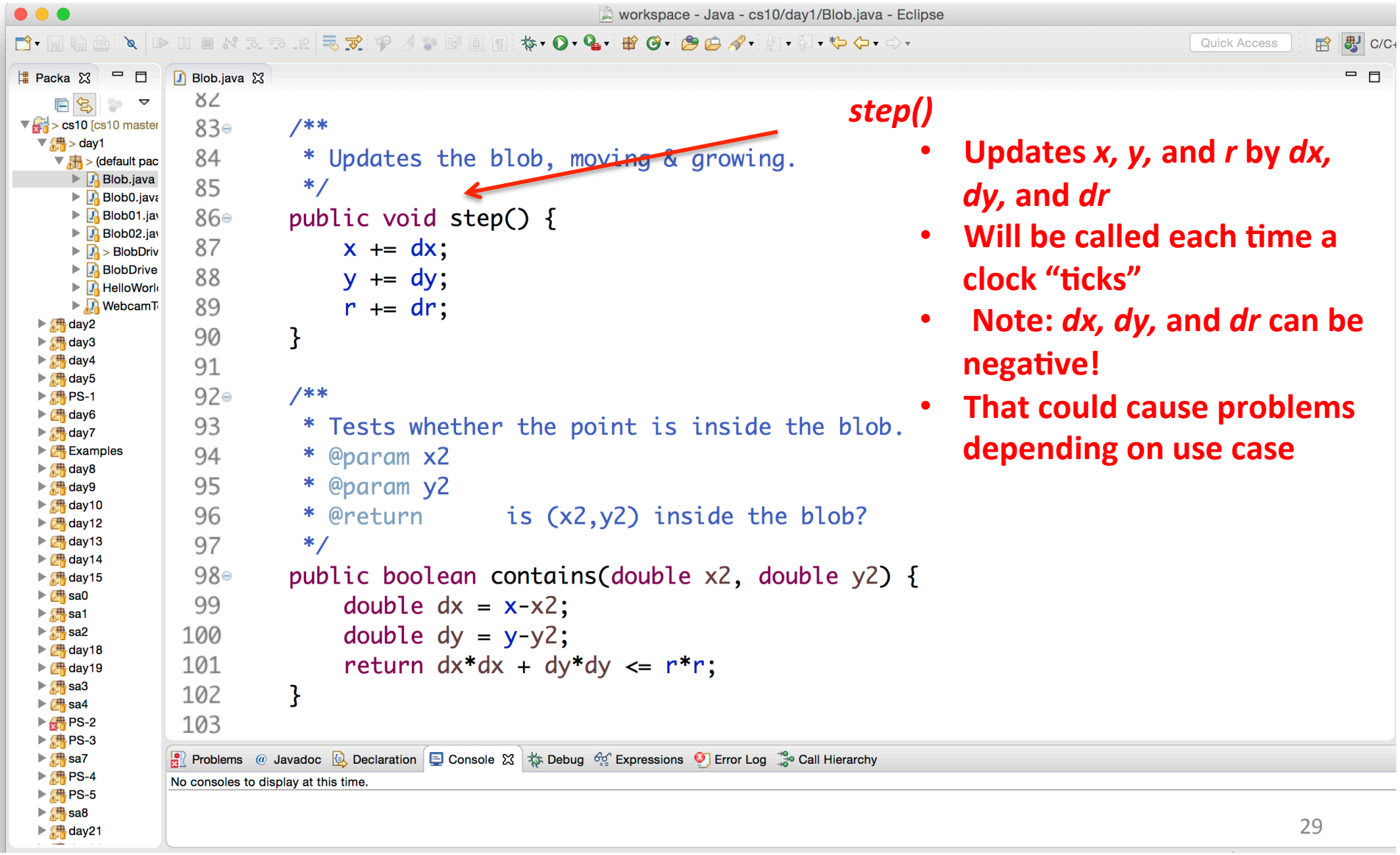


We will use inheritance to create specialized Blobs

Review:

- **Blobs have instance variables x,y, and r for location and size**
- **dx,dy, and dr tell how much to change x,y, and r at each time step (no change if zero)**

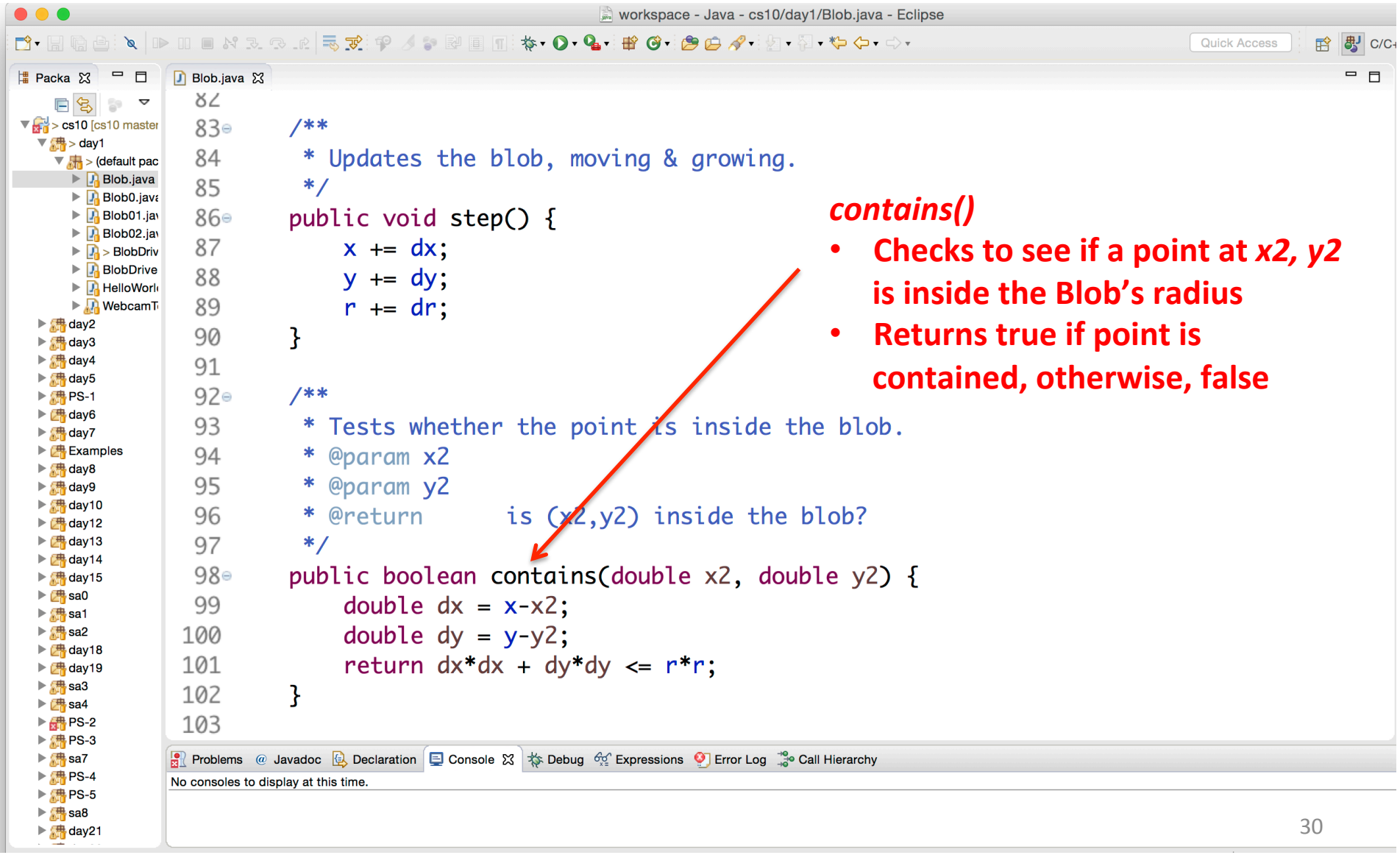
Blob: New methods to control how the blob behaves



step()

- **Updates *x*, *y*, and *r* by *dx*, *dy*, and *dr***
- **Will be called each time a clock "ticks"**
- **Note: *dx*, *dy*, and *dr* can be negative!**
- **That could cause problems depending on use case**

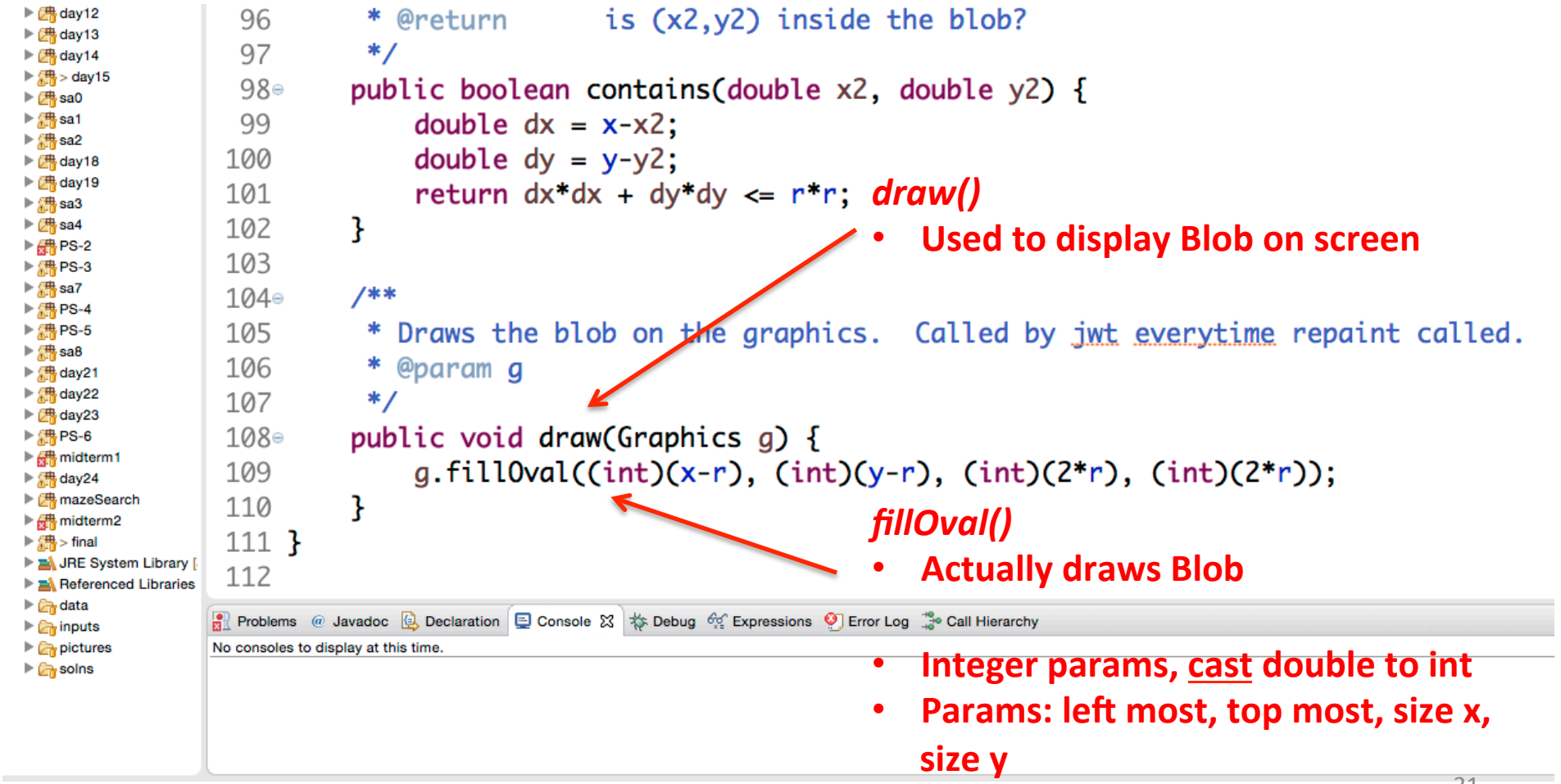
Blob: New methods to control how the blob behaves



contains()

- **Checks to see if a point at *x2, y2* is inside the Blob's radius**
- **Returns true if point is contained, otherwise, false**

Blob: We did not discuss the *draw()* function yesterday



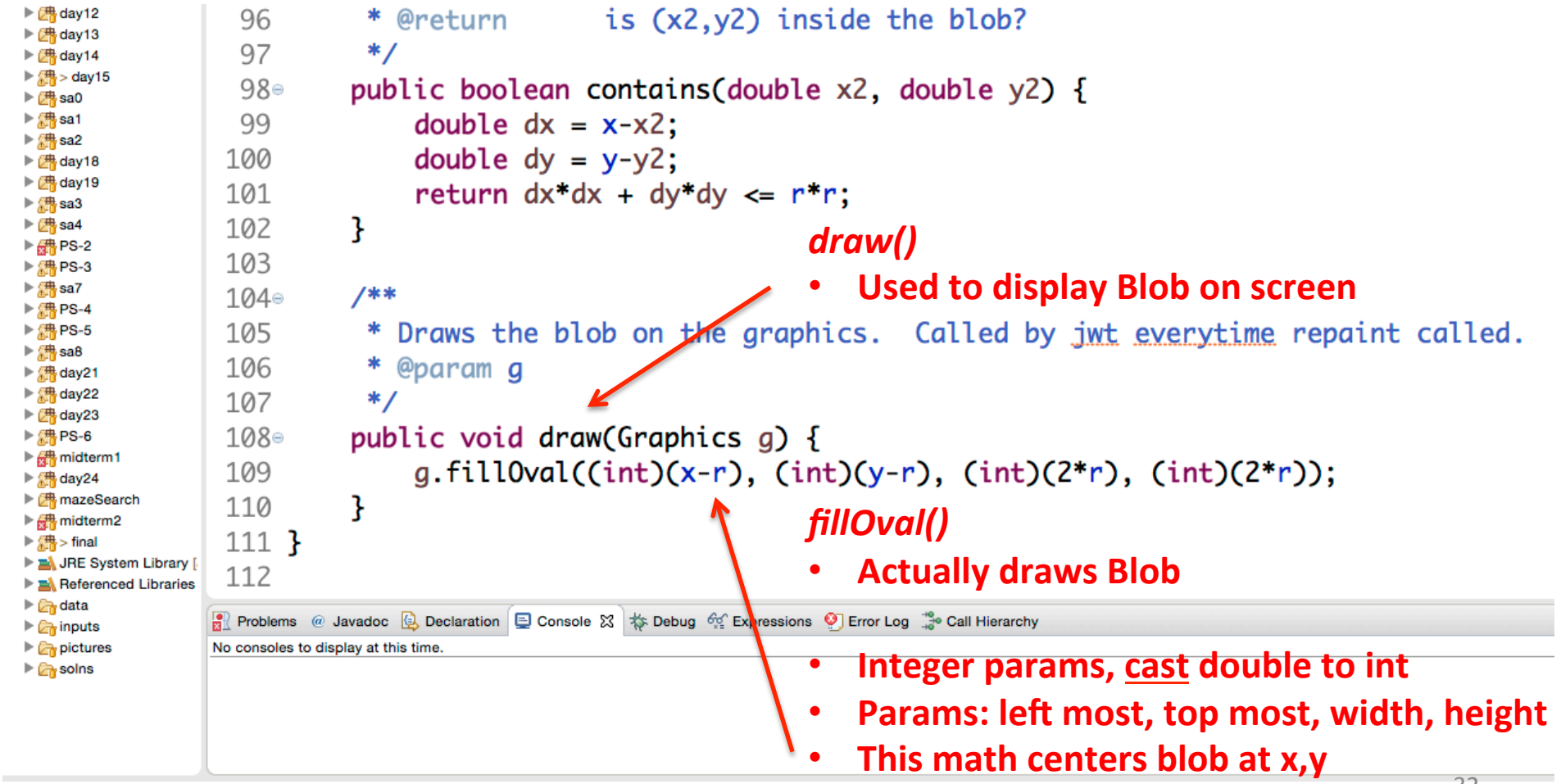
draw()

- **Used to display Blob on screen**

fillOval()

- **Actually draws Blob**
- **Integer params, cast double to int**
- **Params: left most, top most, size x, size y**

Blob: We did not discuss the draw() function last class



Agenda

1. Objects vs. primitives
2. Inheritance
3. Quick review of Blobs
4. Using Inheritance to create different types of Blobs
5. Graphical User Interface (GUI) programming

Use inheritance to create specialized Blobs that have different behavior

Instance variables

- ***x, y, r***
- ***dx, dy, dr***

Blob

Methods

- ***getters/setters***
- ***step()***
- ***contains()***
- ***draw()***

When *step()* called, blob moves location by *dx, dy* and changes size by *dr*

We can create "specialty" blobs that behave differently from the base blob when *step()* called

Specialty blobs (called subclasses) inherit the base blob's instance variables and methods (so no need to re-write them!)

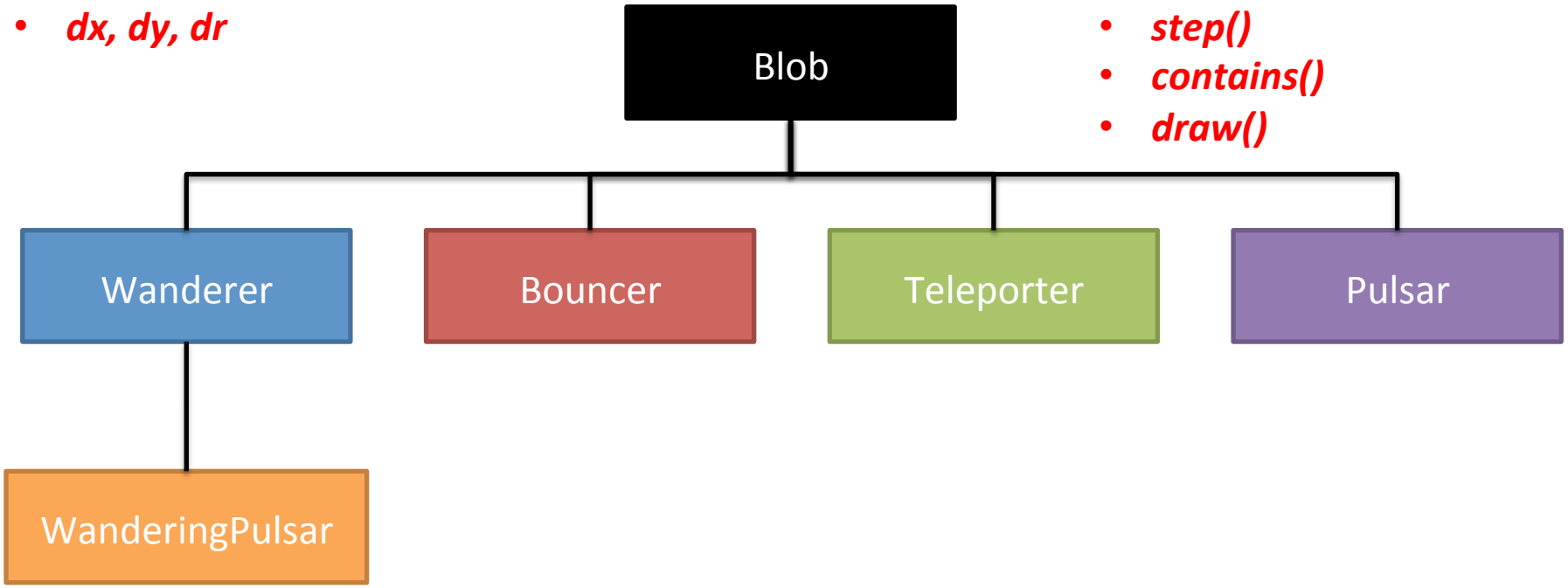
Use inheritance to create specialized Blobs that have different behavior

Instance variables

- *x, y, r*
- *dx, dy, dr*

Methods

- *getters/setters*
- *step()*
- *contains()*
- *draw()*



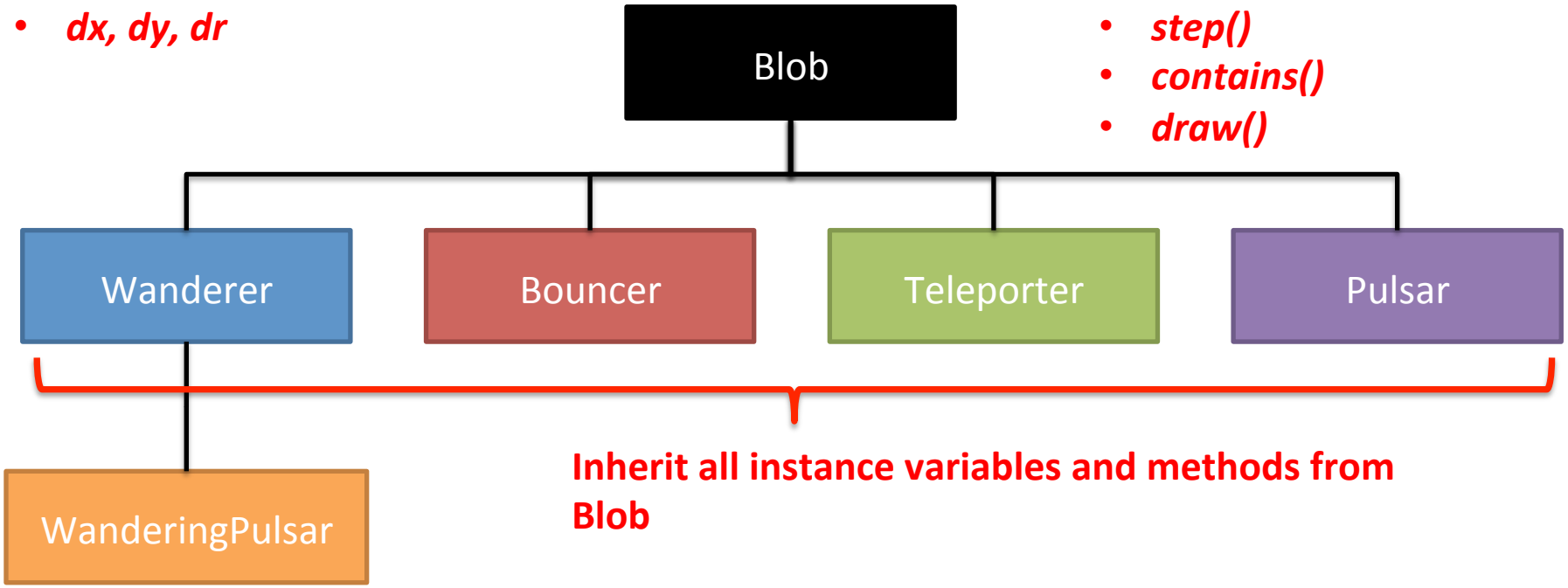
Use inheritance to create specialized Blobs that have different behavior

Instance variables

- ***x, y, r***
- ***dx, dy, dr***

Methods

- ***getters/setters***
- ***step()***
- ***contains()***
- ***draw()***



Inherit all instance variables and methods from Blob

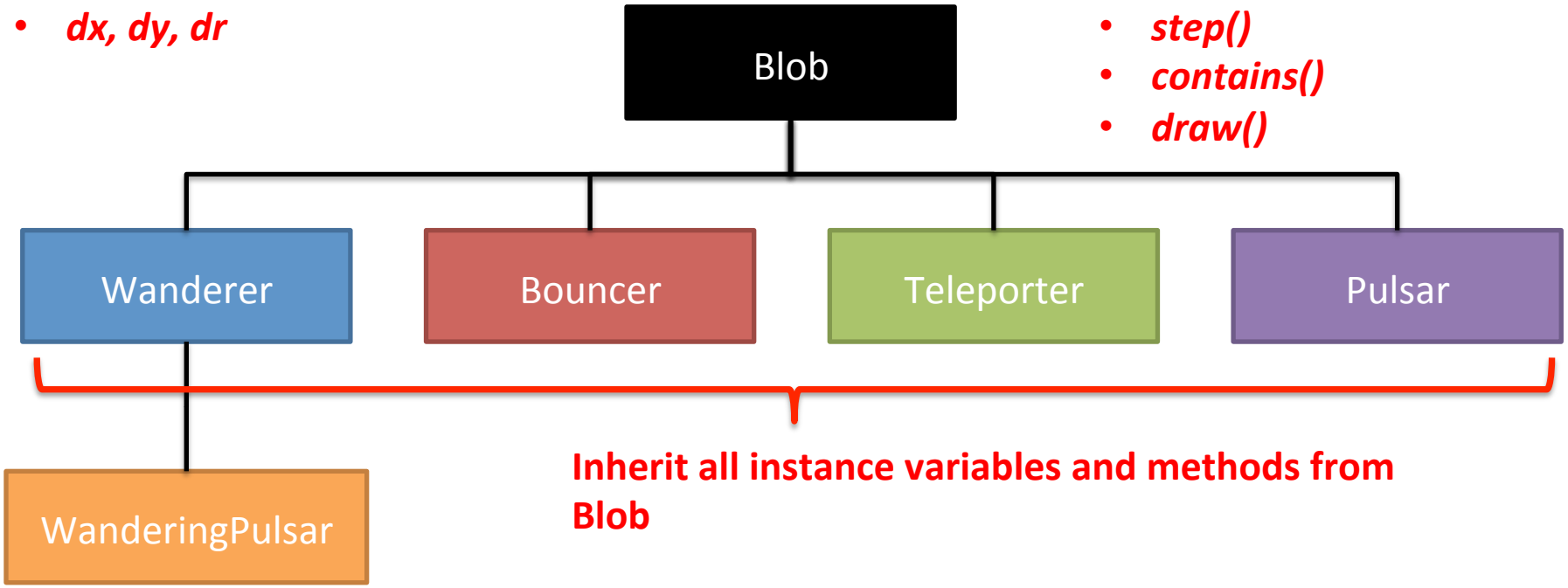
Can override base class methods with "specialty" versions in subclass (this is Polymorphism)

If subclass defines method, subclass's method is used, otherwise base class method used (dynamic dispatch)

Use inheritance to create specialized Blobs that have different behavior

Instance variables
- *x, y, r*
- *dx, dy, dr*

Methods
- *getters/setters*
- *step()*
- *contains()*
- *draw()*



Inherit all instance variables and methods from Blob

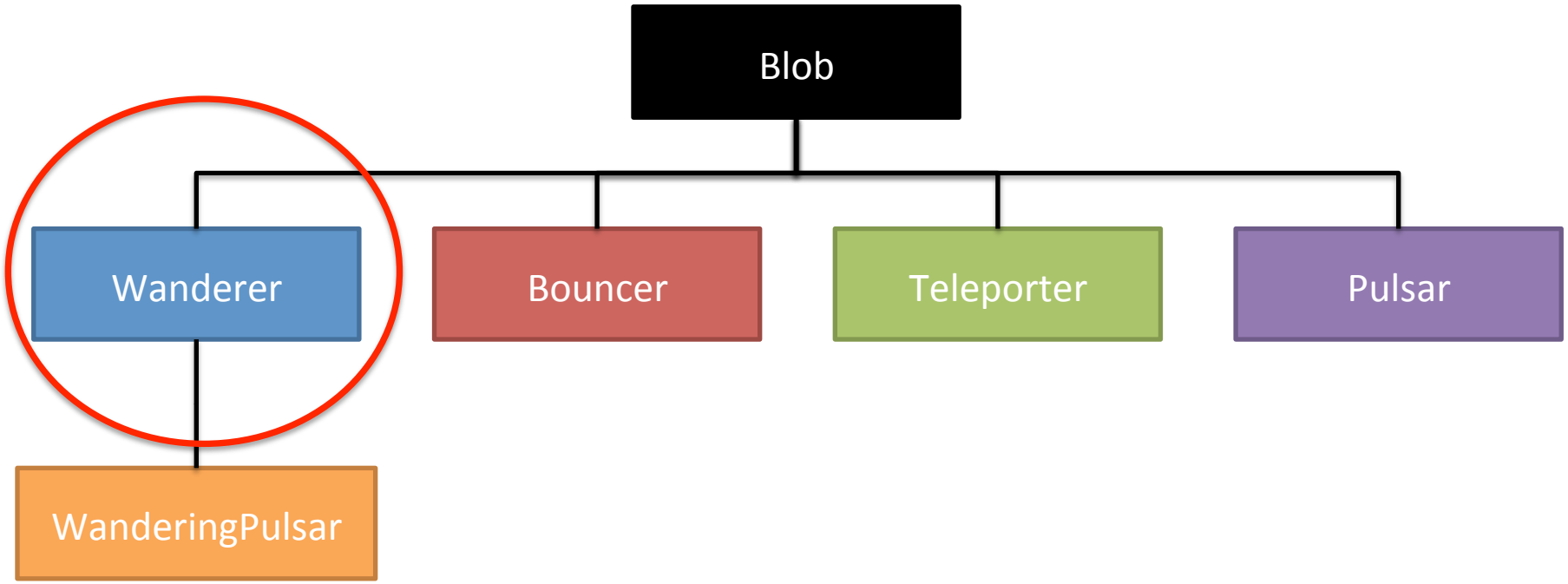
Can override base class methods with "specialty" versions in subclass (this is Polymorphism)

If subclass defines method, subclass's method is used, otherwise base class method used (dynamic dispatch)

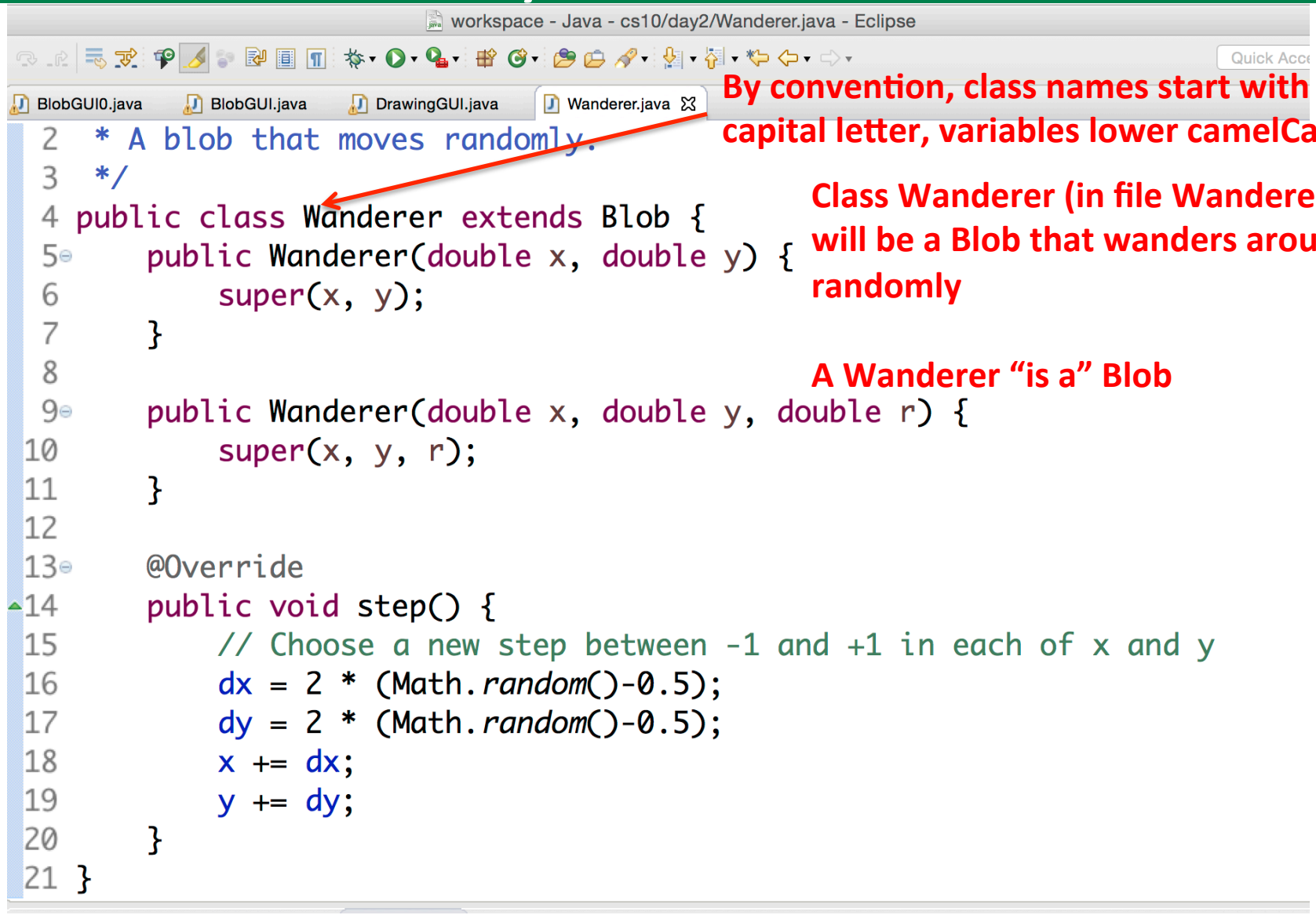
Inherits instance variables and methods from Wanderer

In Java, cannot inherit from multiple base classes

A Wanderer will move around randomly

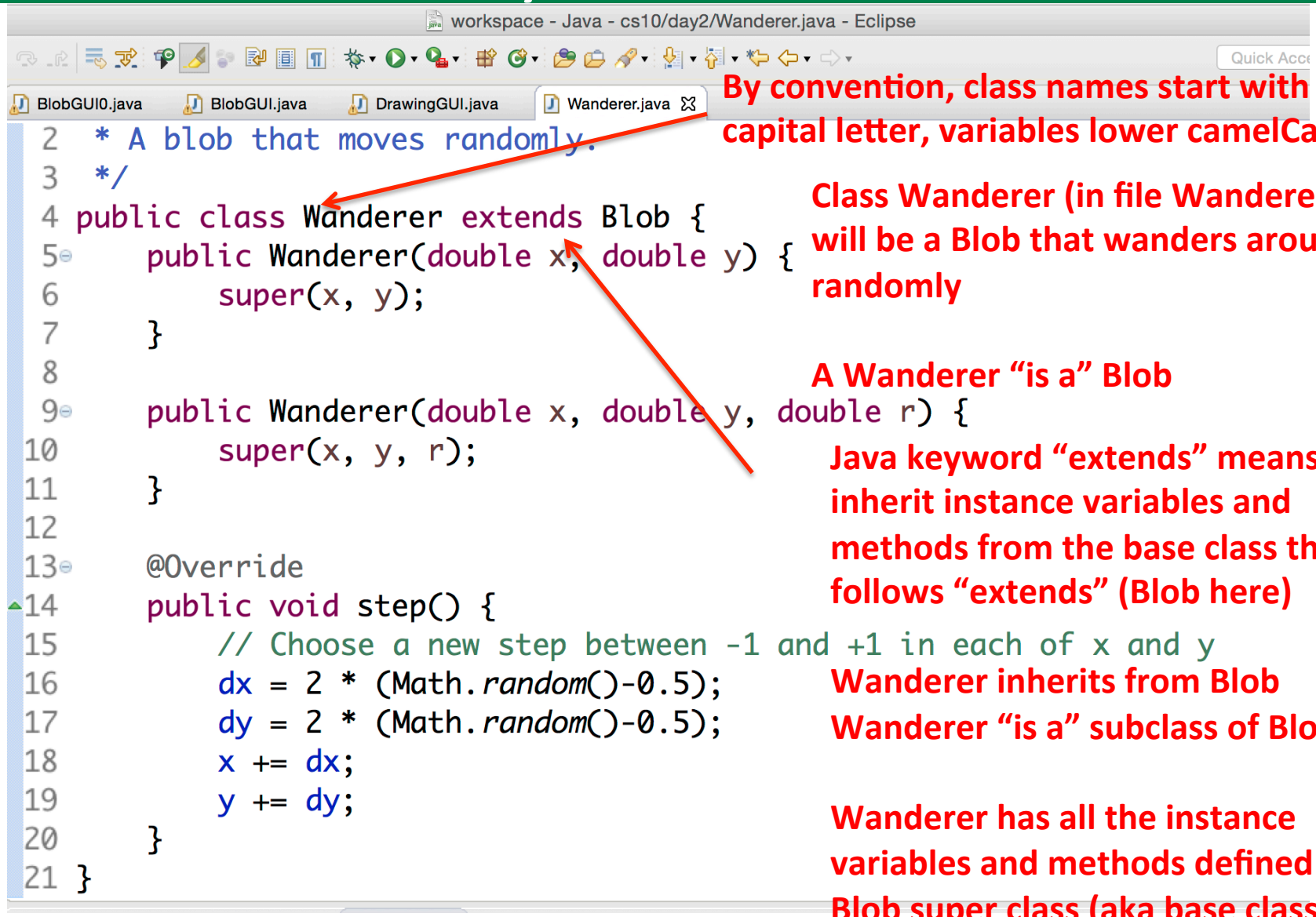


A Wanderer “is a” type of Blob that moves around randomly




```
 2  * A blob that moves randomly.
 3  */
 4 public class Wanderer extends Blob {
 5     public Wanderer(double x, double y) {
 6         super(x, y);
 7     }
 8
 9     public Wanderer(double x, double y, double r) {
10         super(x, y, r);
11     }
12
13     @Override
14     public void step() {
15         // Choose a new step between -1 and +1 in each of x and y
16         dx = 2 * (Math.random()-0.5);
17         dy = 2 * (Math.random()-0.5);
18         x += dx;
19         y += dy;
20     }
21 }
```

A Wanderer “is a” type of Blob that moves around randomly



By convention, class names start with capital letter, variables lower camelCase

Class Wanderer (in file Wanderer.java) will be a Blob that wanders around randomly

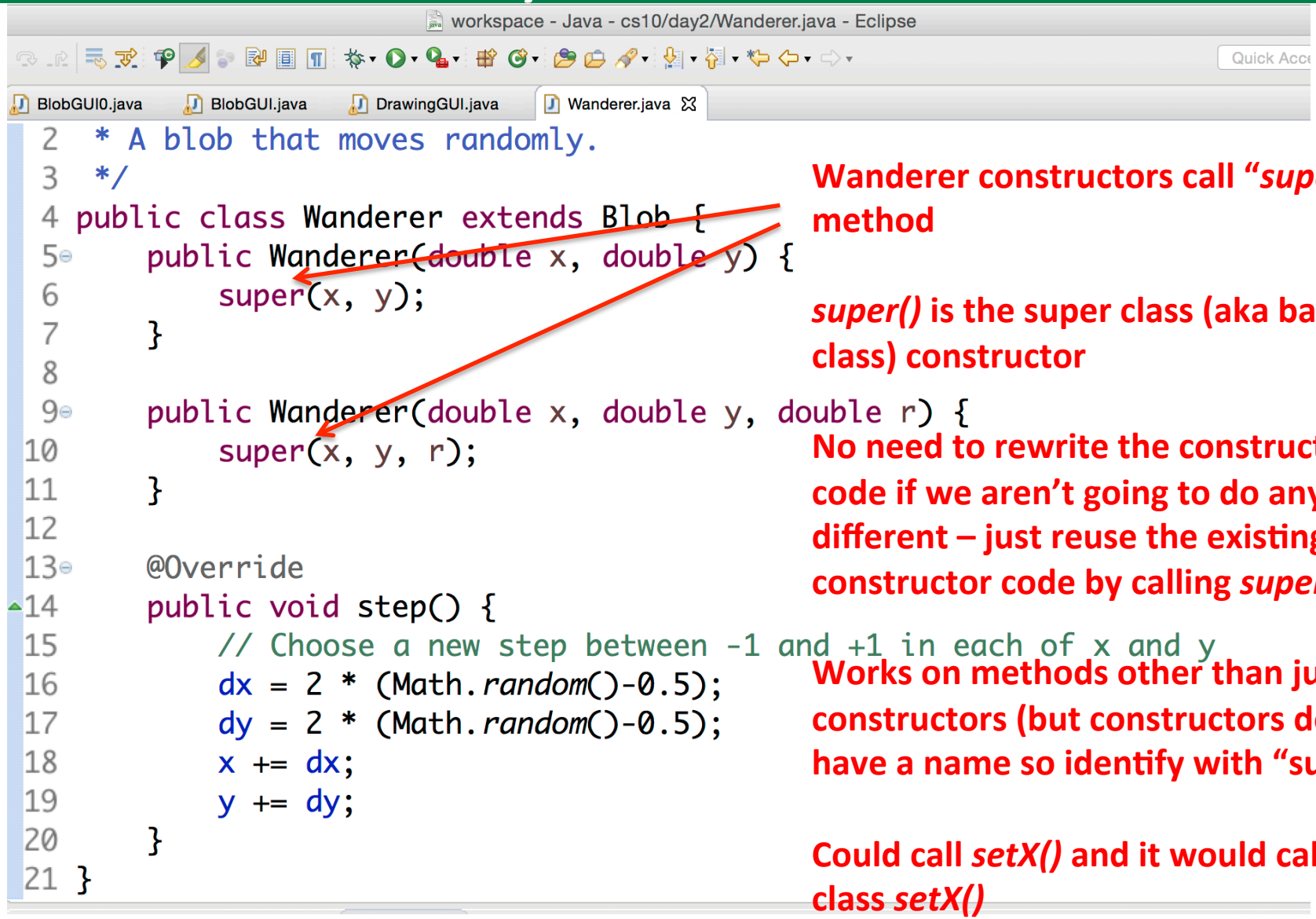
A Wanderer “is a” Blob

Java keyword “extends” means inherit instance variables and methods from the base class that follows “extends” (Blob here)

Wanderer inherits from Blob
Wanderer “is a” subclass of Blob

Wanderer has all the instance variables and methods defined in Blob super class (aka base class)

A Wanderer “is a” type of Blob that moves around randomly



Wanderer constructors call “*super()*” method

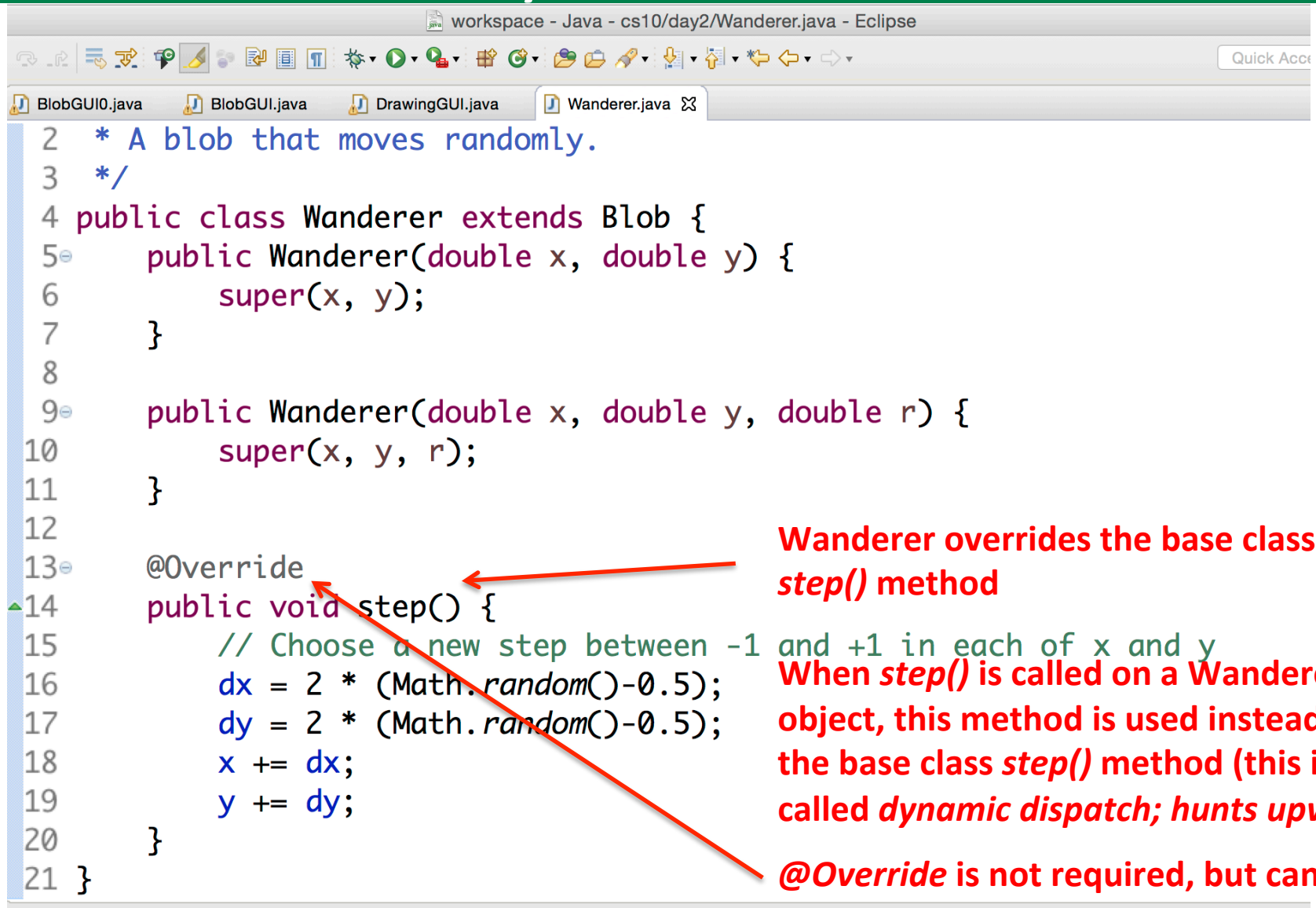
***super()* is the super class (aka base class) constructor**

No need to rewrite the constructor code if we aren’t going to do anything different – just reuse the existing constructor code by calling *super()*

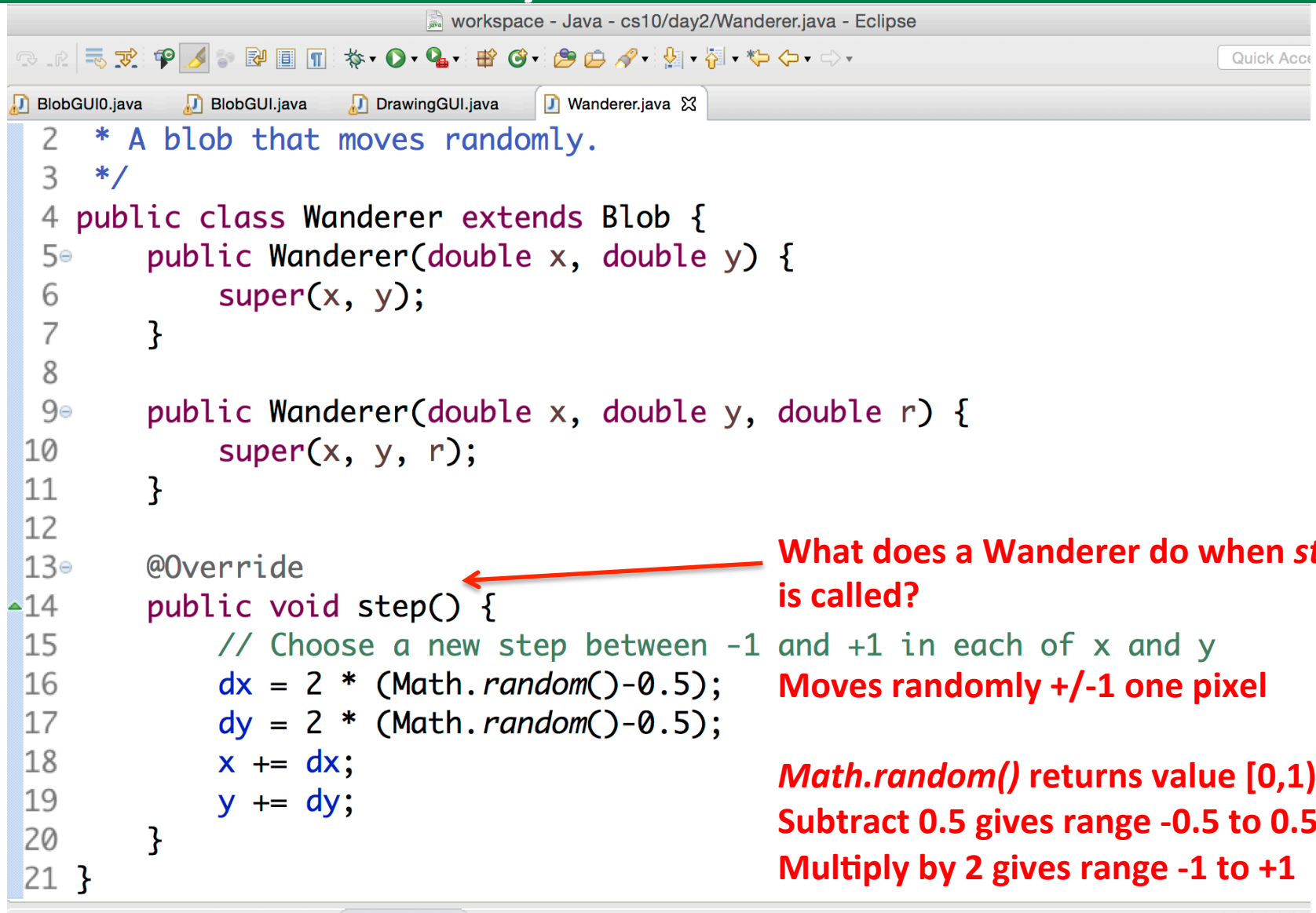
Works on methods other than just constructors (but constructors don’t have a name so identify with “super”)

Could call *setX()* and it would call base class *setX()*

A Wanderer “is a” type of Blob that moves around randomly



A Wanderer “is a” type of Blob that moves around randomly

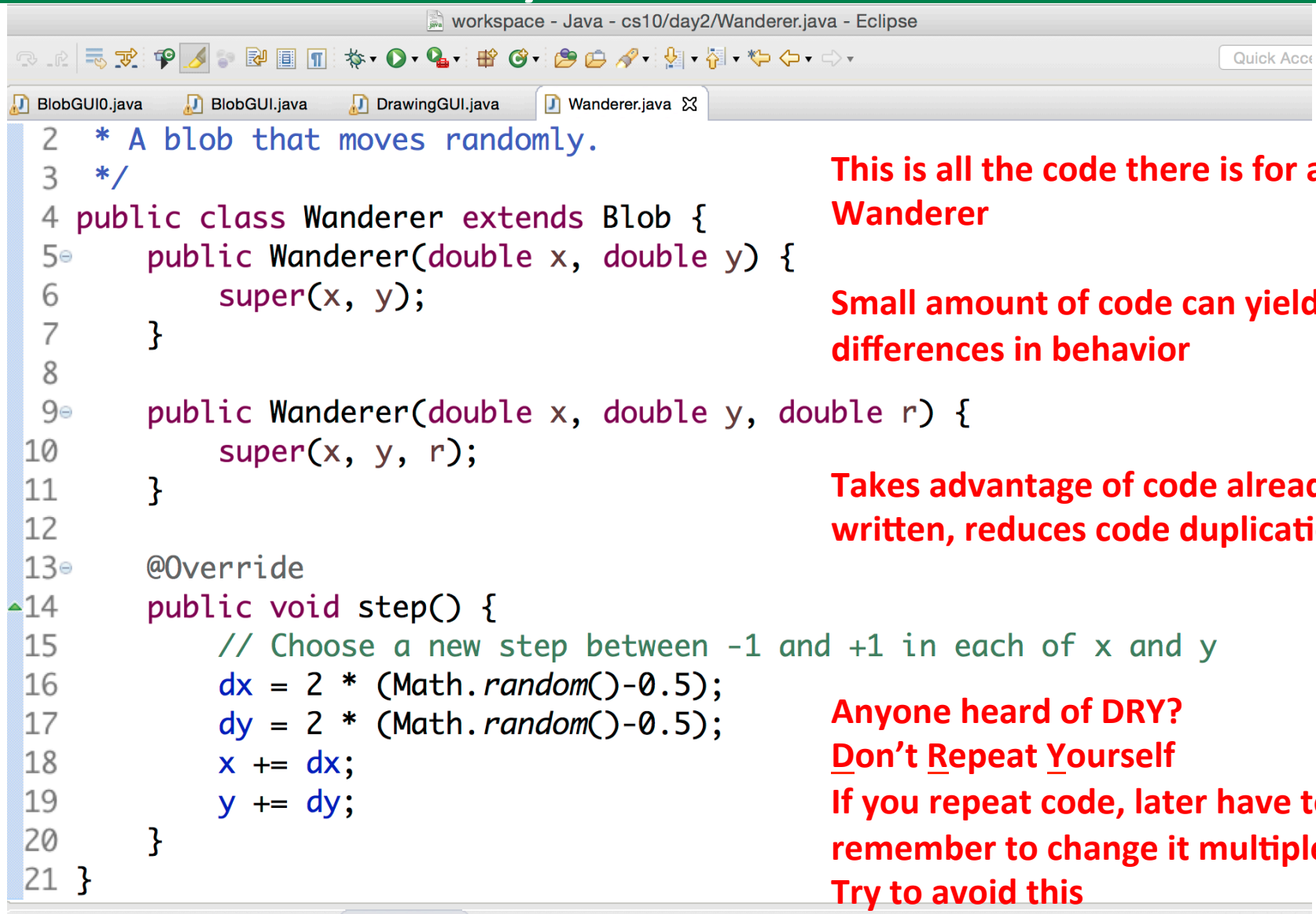


What does a Wanderer do when *step()* is called?

Moves randomly +/-1 one pixel

***Math.random()* returns value [0,1)**
Subtract 0.5 gives range -0.5 to 0.5
Multiply by 2 gives range -1 to +1

A Wanderer “is a” type of Blob that moves around randomly




```
 2  * A blob that moves randomly.
 3  */
 4 public class Wanderer extends Blob {
 5      public Wanderer(double x, double y) {
 6          super(x, y);
 7      }
 8
 9      public Wanderer(double x, double y, double r) {
10          super(x, y, r);
11      }
12
13      @Override
14      public void step() {
15          // Choose a new step between -1 and +1 in each of x and y
16          dx = 2 * (Math.random()-0.5);
17          dy = 2 * (Math.random()-0.5);
18          x += dx;
19          y += dy;
20      }
21 }
```

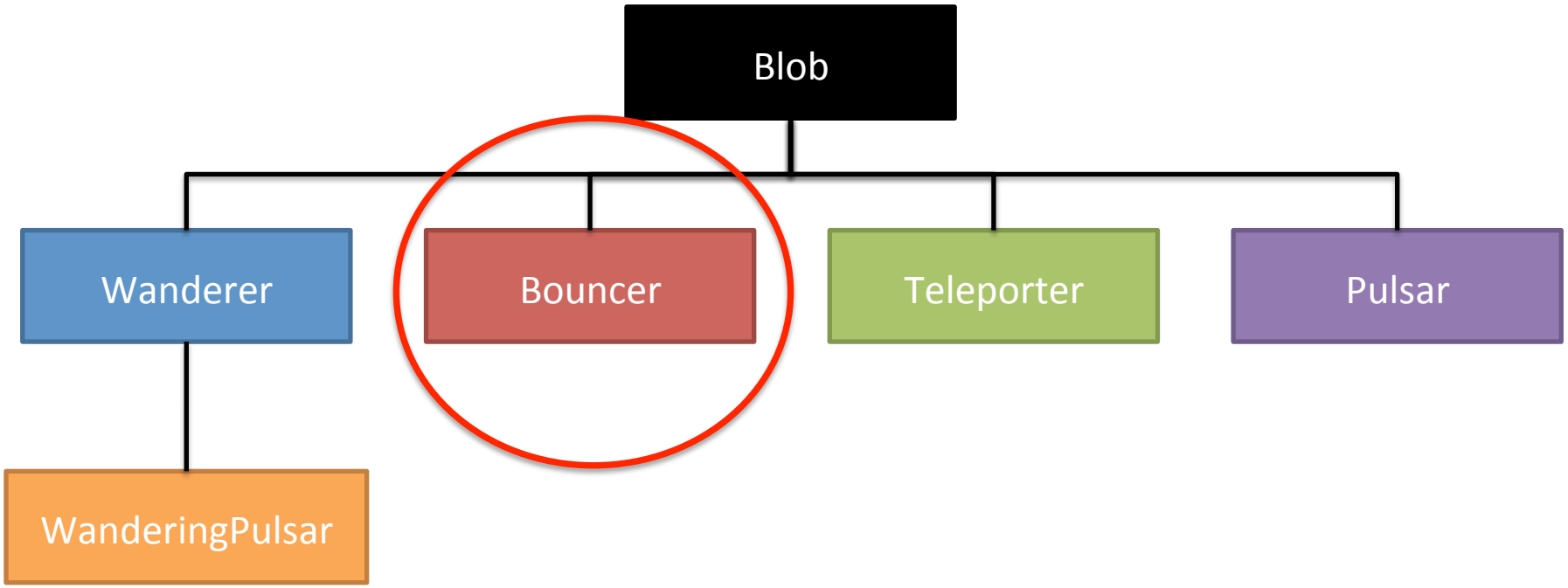
This is all the code there is for a Wanderer

Small amount of code can yield big differences in behavior

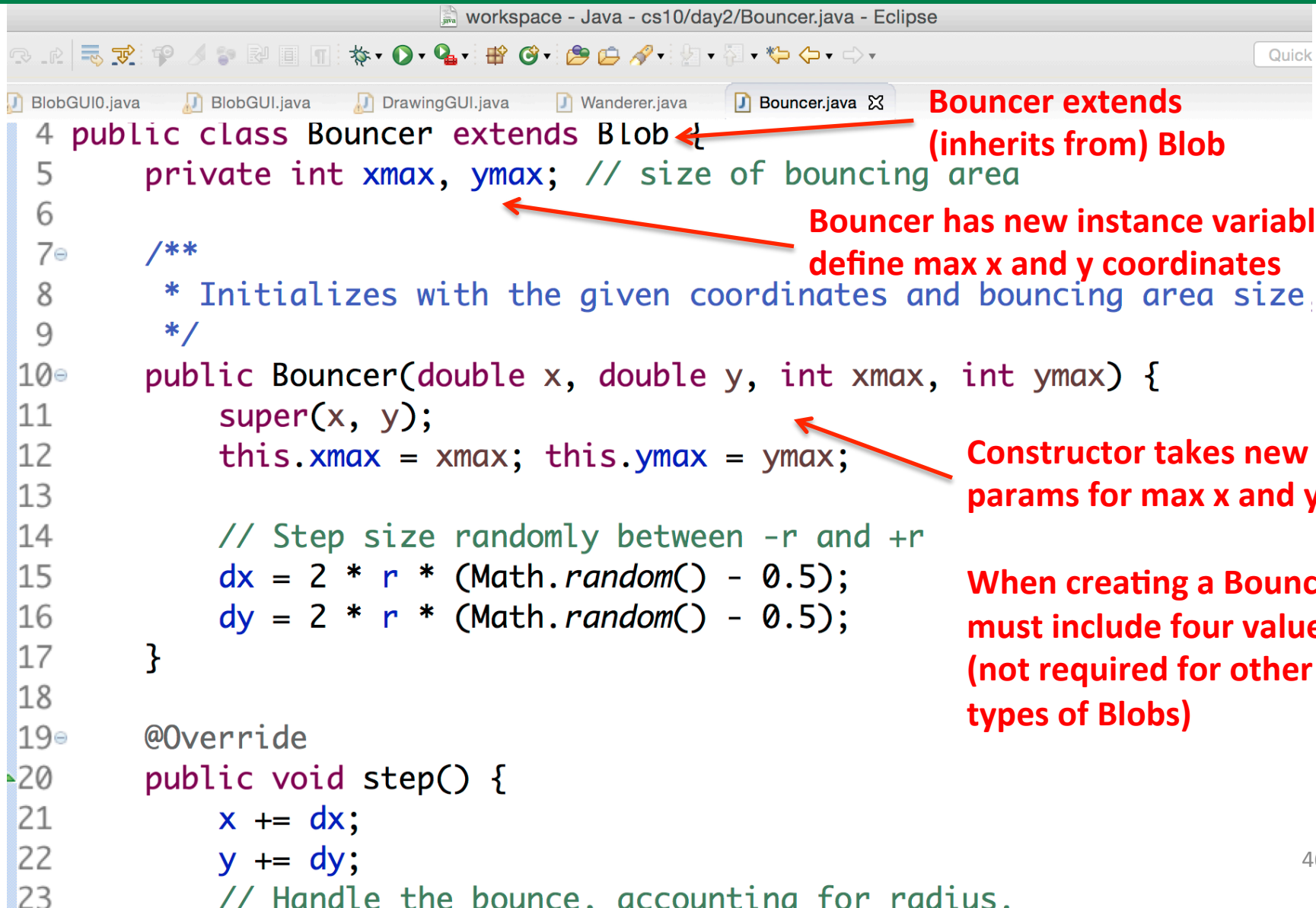
Takes advantage of code already written, reduces code duplication

Anyone heard of DRY?
Don’t Repeat Yourself
If you repeat code, later have to remember to change it multiple places
Try to avoid this

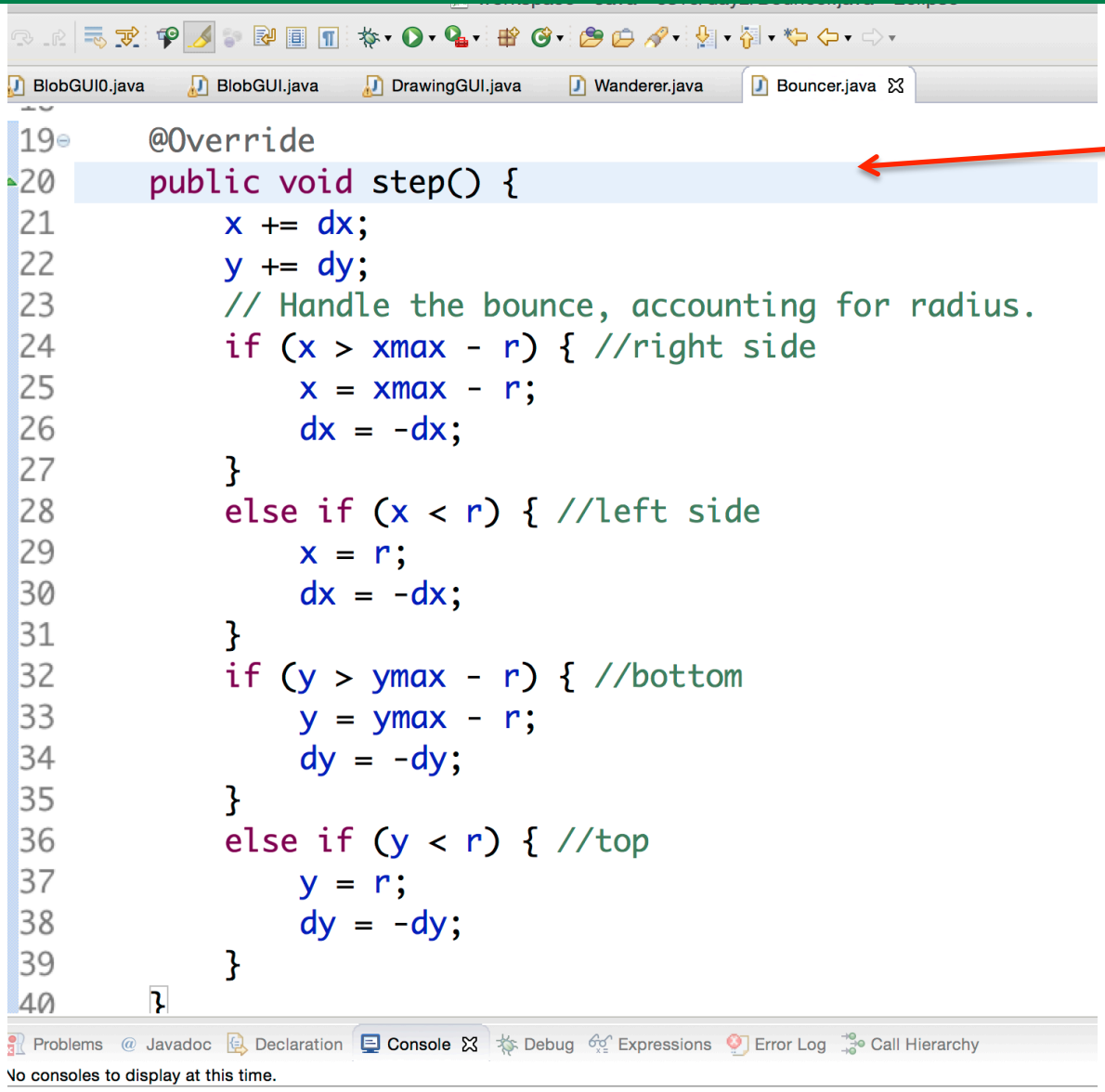
A Bouncer will move in a straight line until it hits a wall, then it will rebound



A Bouncer "is a" type of Blob that rebounds off sides



A Bouncer “is a” type of Blob that rebounds off sides

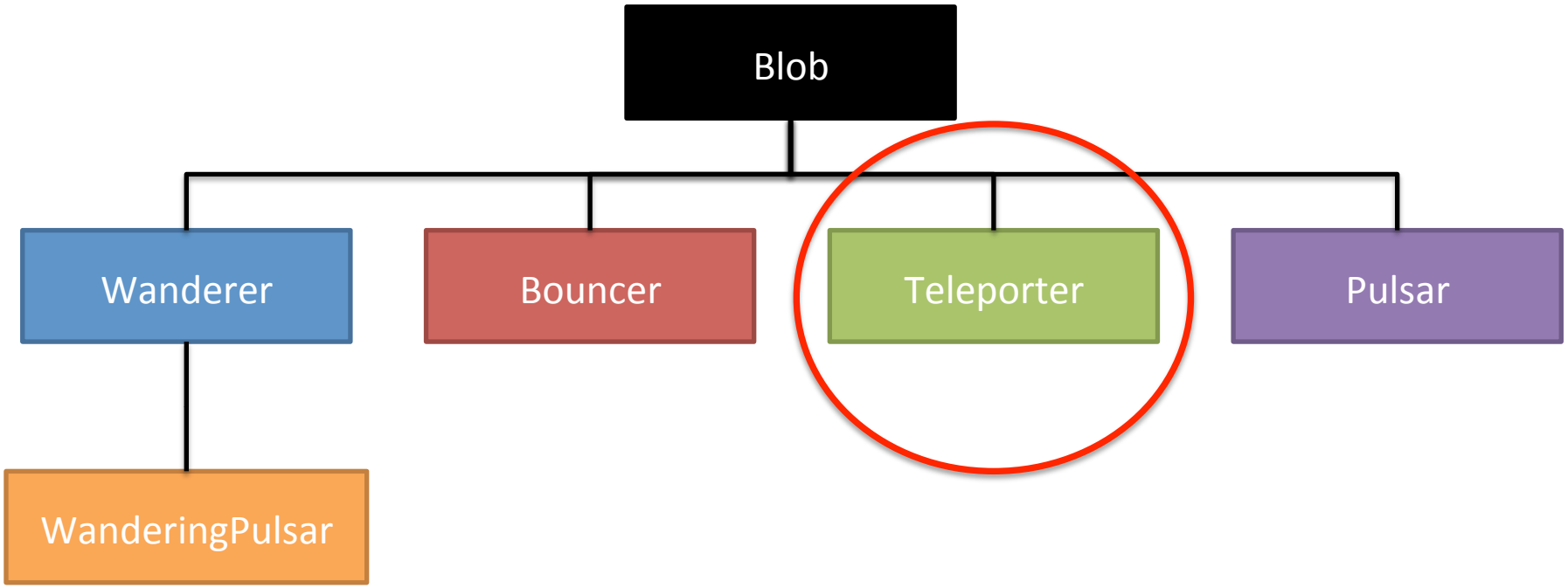


Overrides *step()* method

Blob moves in straight line, but ensures coordinates in range:

- ***r ≤ x ≤ xmax***
- ***r ≤ y ≤ ymax***
- **Reverse sign on *dx* or *dy* if hit wall to rebound in the opposite direction**

A Teleporter can jump to a new location



Teleporter adds a new method other classes do not have

BlobGUI0.java | BlobGUI.java | DrawingGUI.java | Wanderer.java | Bouncer.java | Teleporter.java | TeleportTest.java

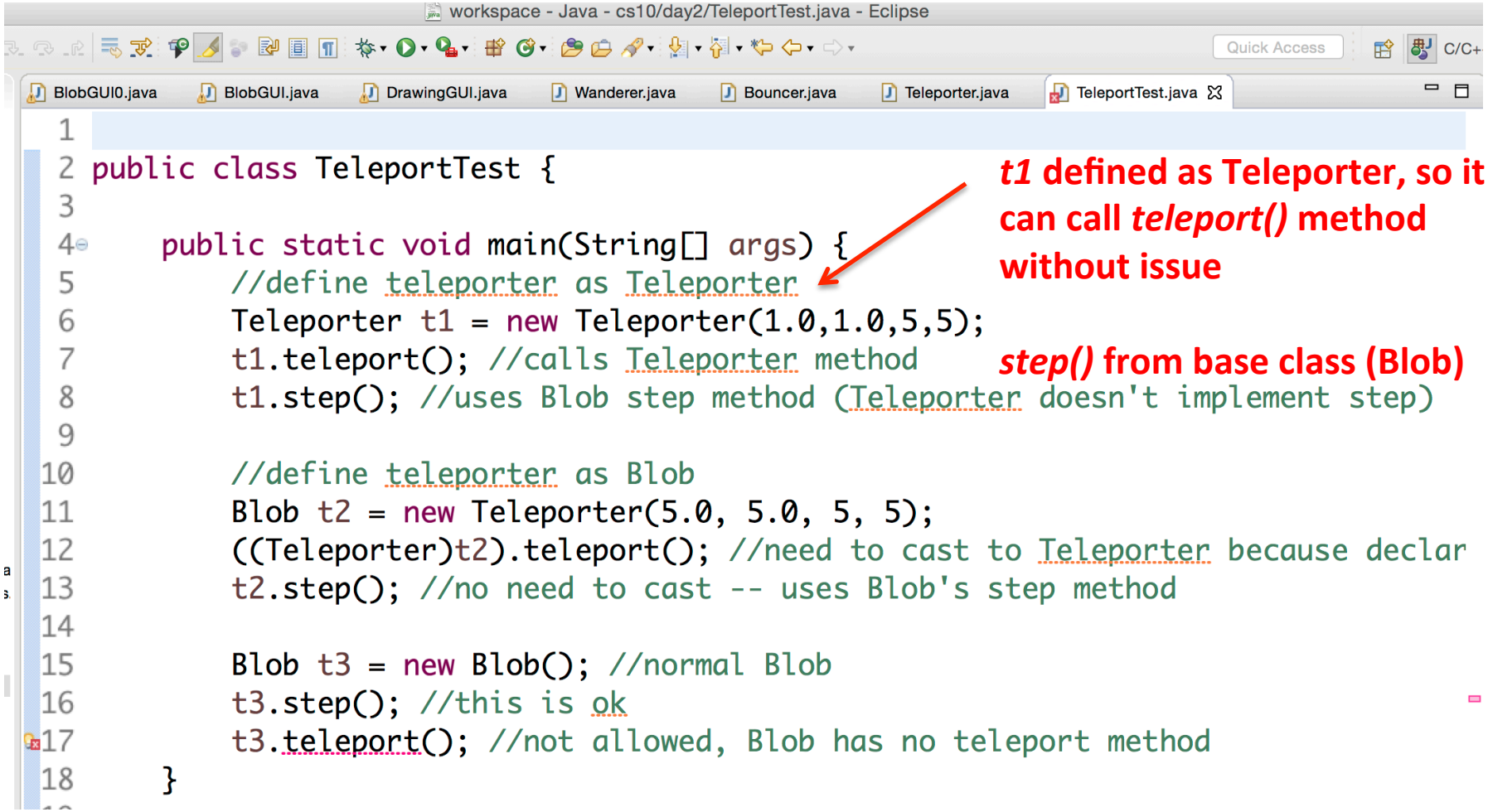
```
1  /**
2   * A blob with a new method to jump to a random position.
3   */
4  public class Teleporter extends Blob {
5      private int xmax, ymax; // size of teleporting area
6
7      public Teleporter(double x, double y, int xmax, int ymax) {
8          super(x, y);
9          this.xmax = xmax; this.ymax = ymax;
10     }
11
12     /**
13      * Moves the blob to a random new position
14      */
15     public void teleport() {
16         x = Math.random()*xmax;
17         y = Math.random()*ymax;
18     }
19 }
```

Size of screen stored in xmax and ymax

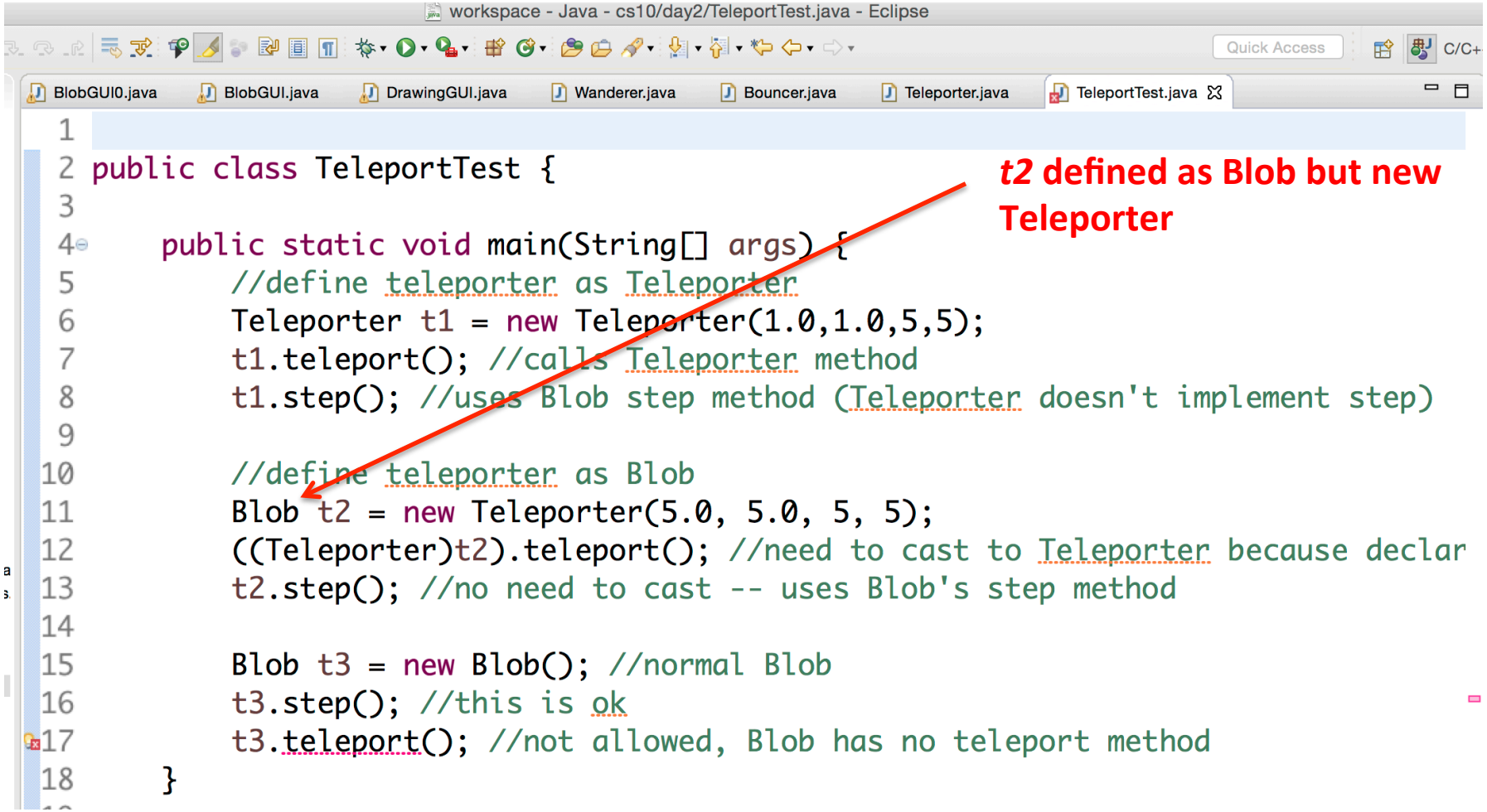
Only Blobs of type Teleporter can call *teleport()* method (others do not have this method)

What happens when *teleport()* called?
Jump to random x,y location

Calling teleport() can be tricky

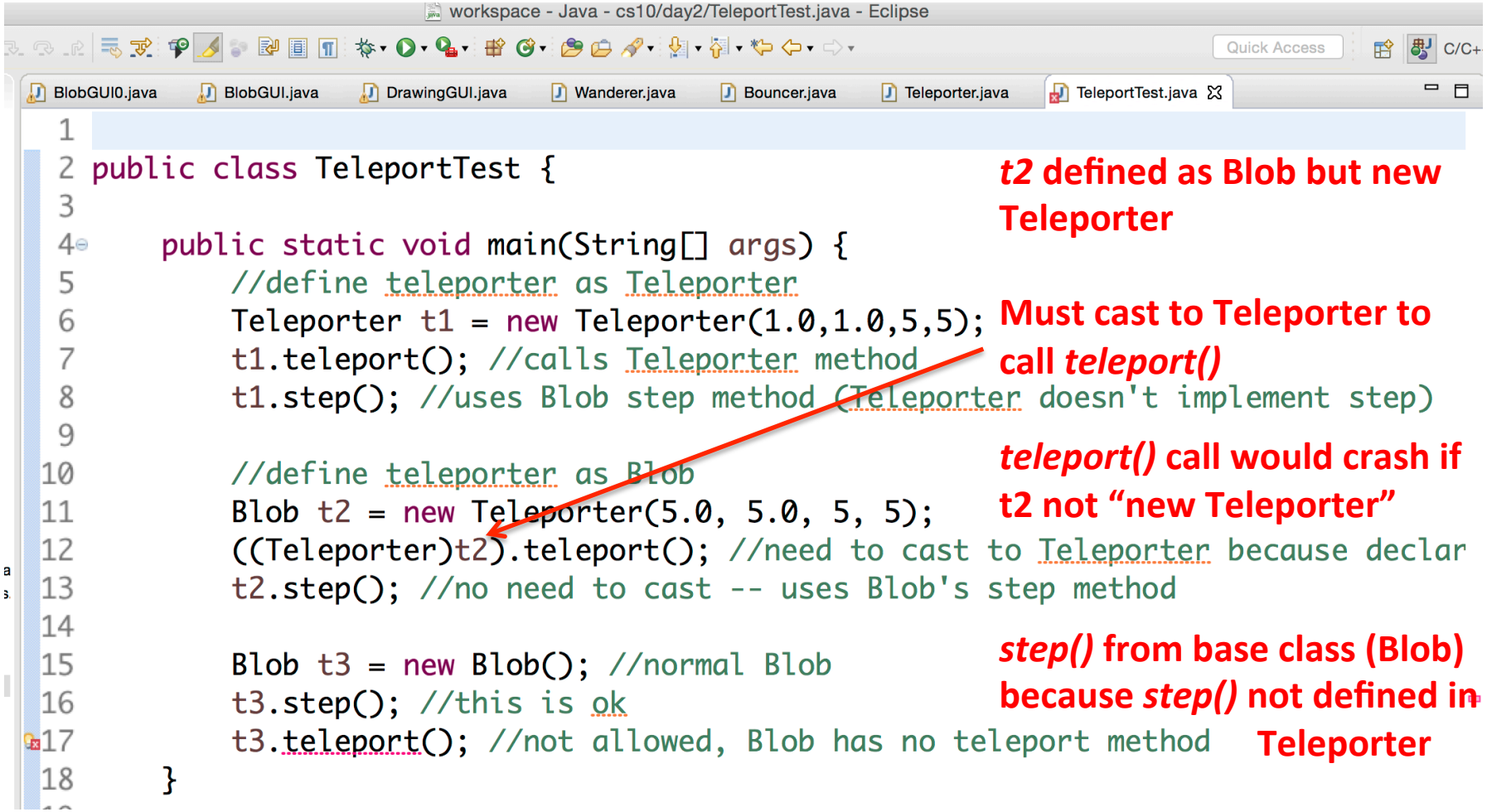


Calling teleport() can be tricky

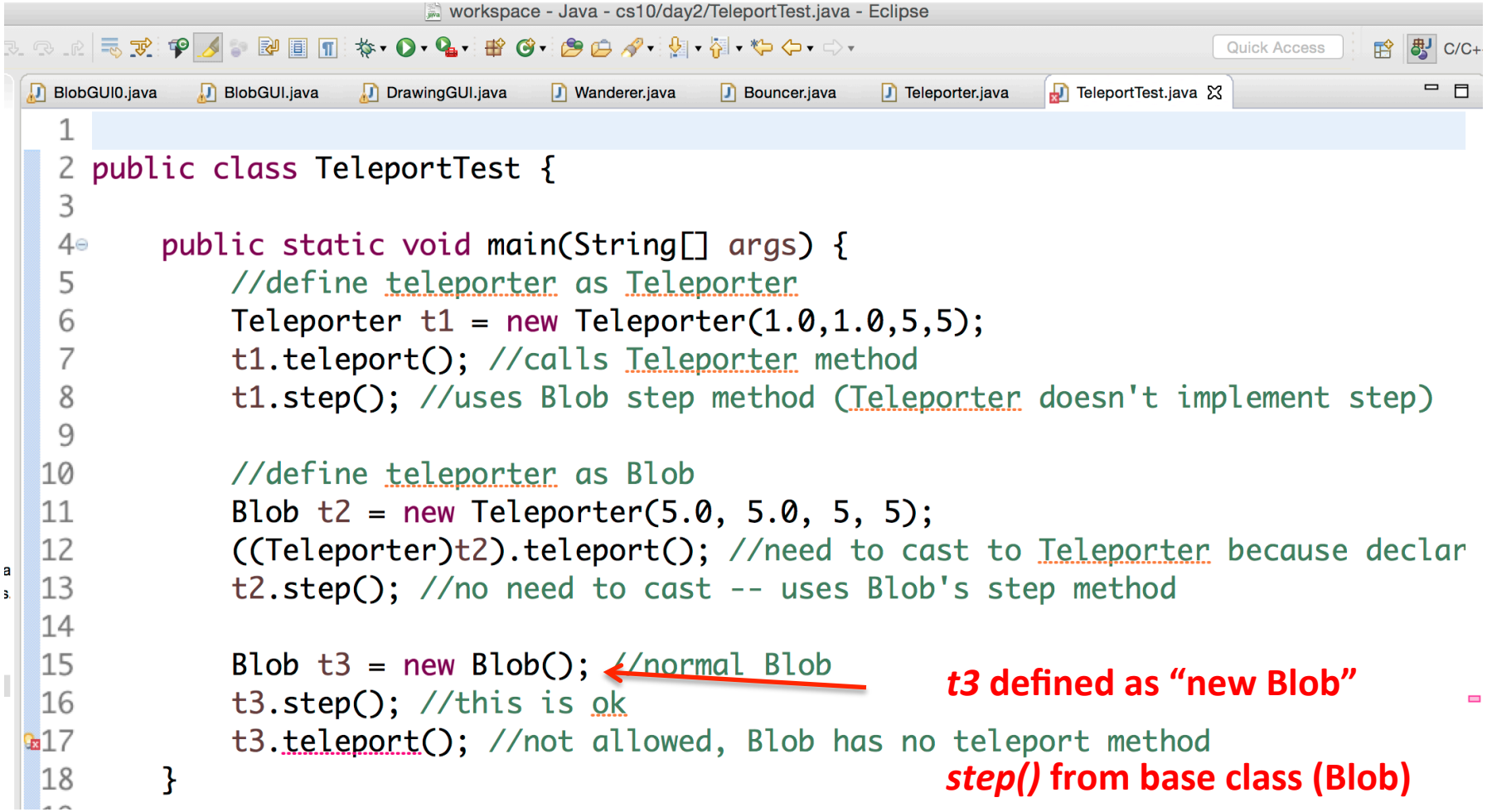



```
1
2  public class TeleportTest {
3
4      public static void main(String[] args) {
5          //define teleporter as Teleporter
6          Teleporter t1 = new Teleporter(1.0,1.0,5,5);
7          t1.teleport(); //calls Teleporter method
8          t1.step(); //uses Blob step method (Teleporter doesn't implement step)
9
10         //define teleporter as Blob
11         Blob t2 = new Teleporter(5.0, 5.0, 5, 5);
12         ((Teleporter)t2).teleport(); //need to cast to Teleporter because declar
13         t2.step(); //no need to cast -- uses Blob's step method
14
15         Blob t3 = new Blob(); //normal Blob
16         t3.step(); //this is ok
17         t3.teleport(); //not allowed, Blob has no teleport method
18     }
```

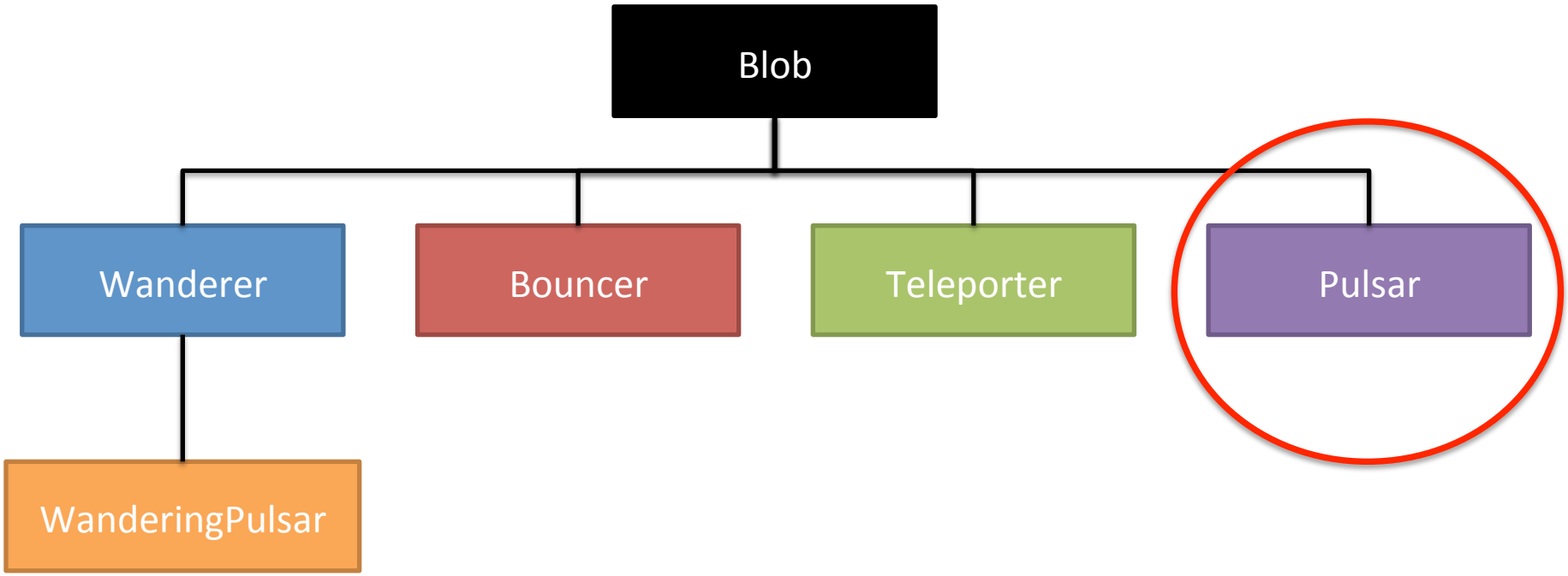
Calling teleport() can be tricky



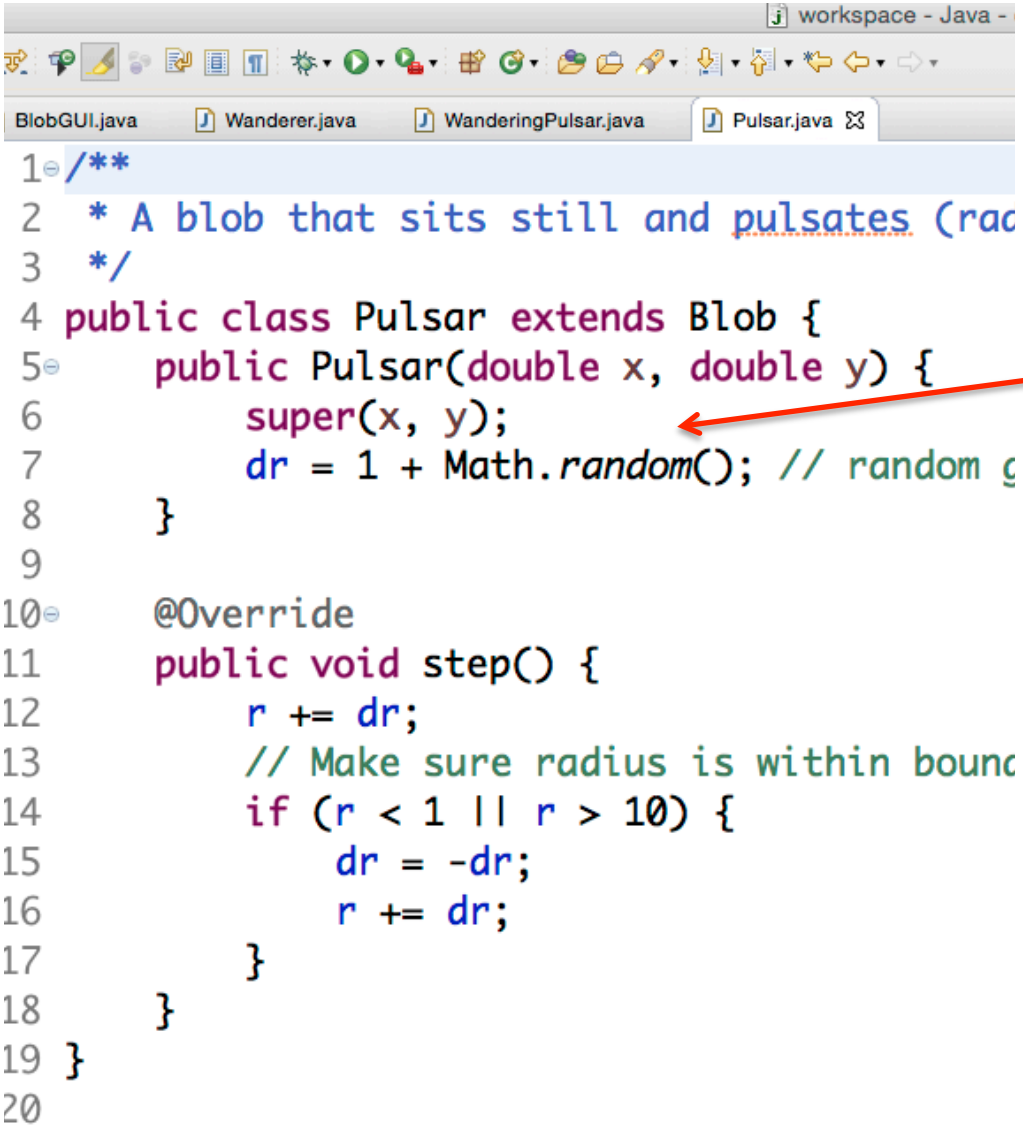
Calling teleport() can be tricky



A Pulsar grows and shrinks, but does not move



A Pulsar grows and shrinks, but does not move

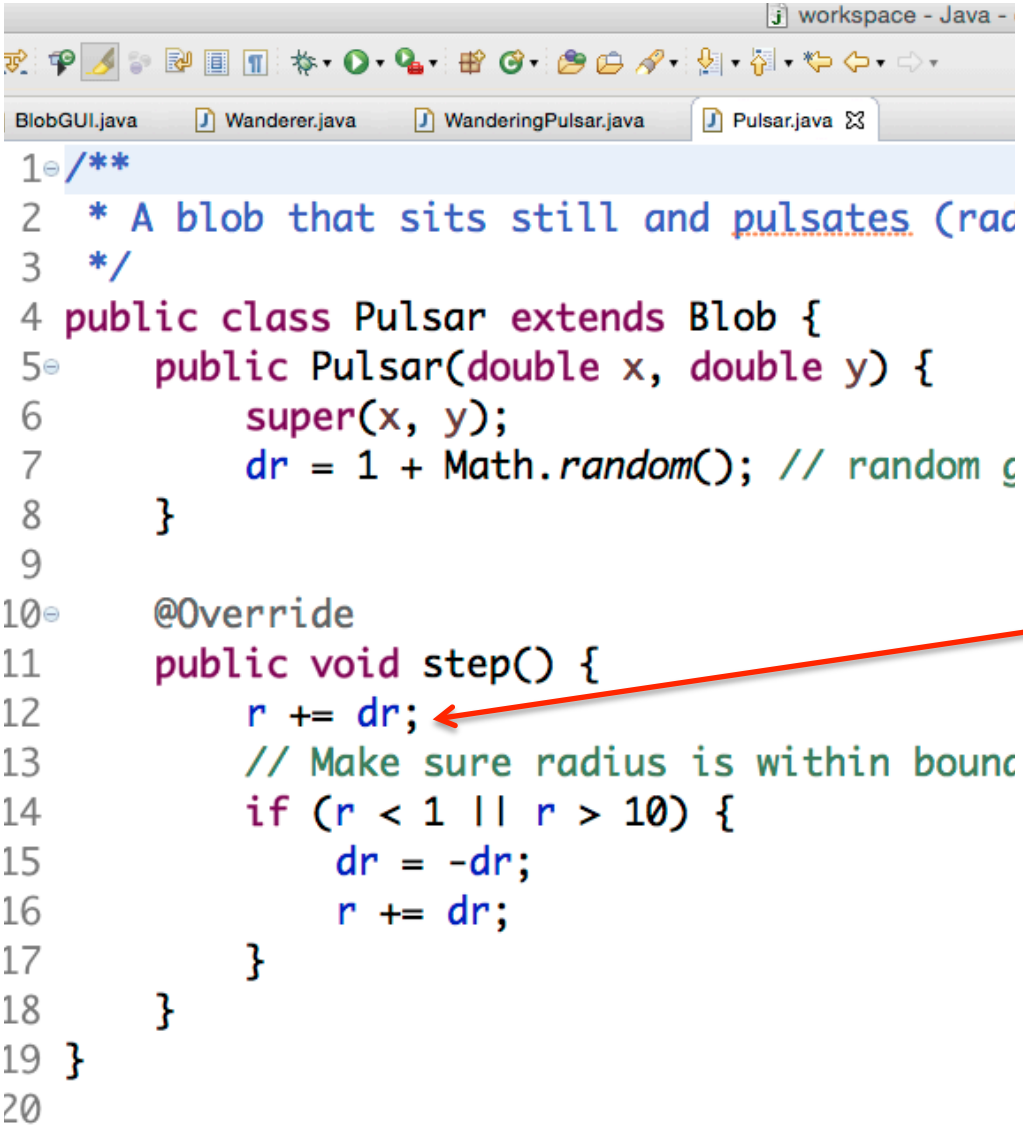



```
1  /**
2   * A blob that sits still and pulsates (rad
3   */
4  public class Pulsar extends Blob {
5      public Pulsar(double x, double y) {
6          super(x, y);
7          dr = 1 + Math.random(); // random g
8      }
9
10     @Override
11     public void step() {
12         r += dr;
13         // Make sure radius is within bound
14         if (r < 1 || r > 10) {
15             dr = -dr;
16             r += dr;
17         }
18     }
19 }
20
```

Call super constructor

Pick a random radius growth rate between 1 and 2

A Pulsar grows and shrinks, but does not move

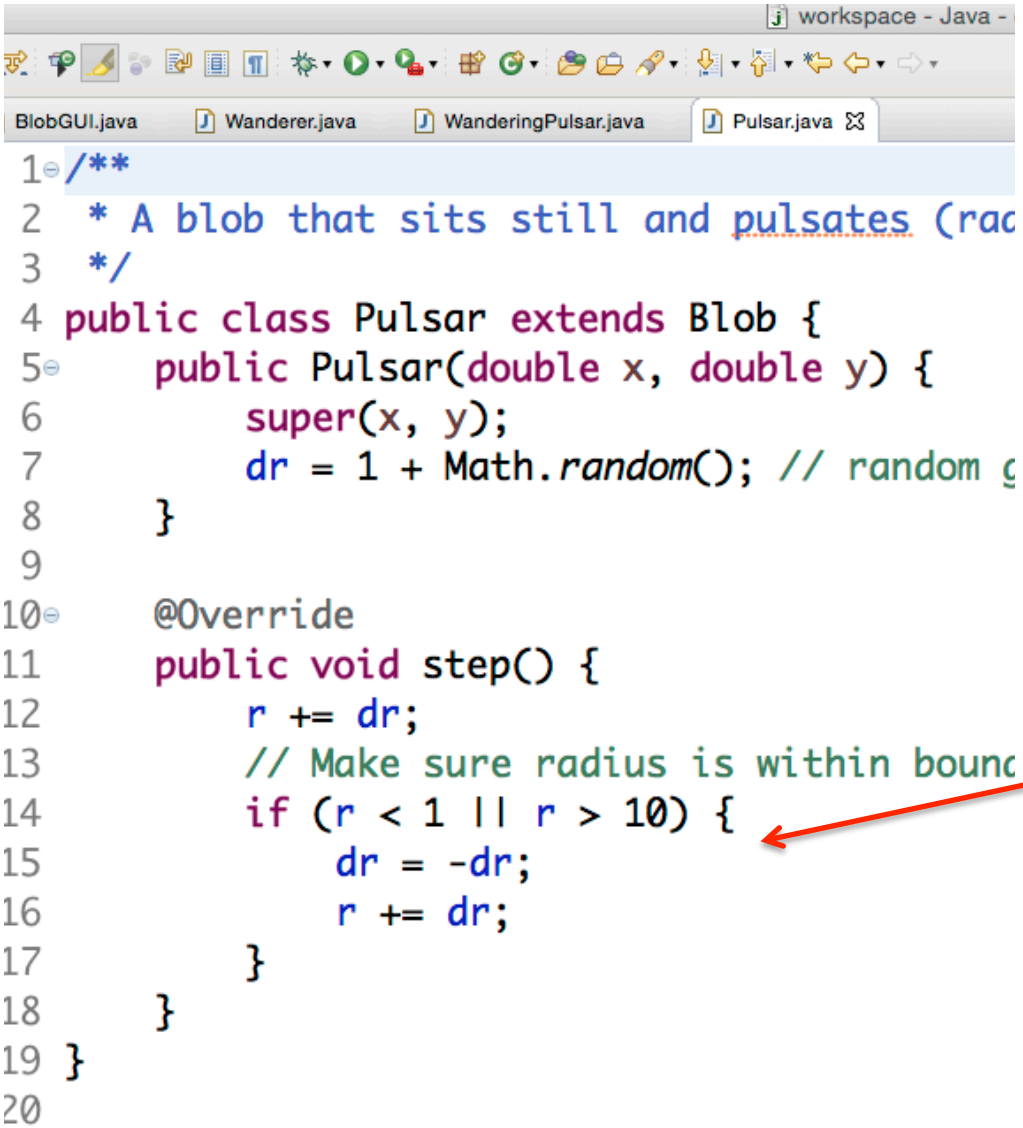


Call super constructor

Pick a random radius growth rate between 1 and 2

Update radius on *step()*

A Pulsar grows and shrinks, but does not move




```
1  /**
2   * A blob that sits still and pulsates (rad
3   */
4  public class Pulsar extends Blob {
5      public Pulsar(double x, double y) {
6          super(x, y);
7          dr = 1 + Math.random(); // random g
8      }
9
10     @Override
11     public void step() {
12         r += dr;
13         // Make sure radius is within bound
14         if (r < 1 || r > 10) {
15             dr = -dr;
16             r += dr;
17         }
18     }
19 }
20
```

Call super constructor

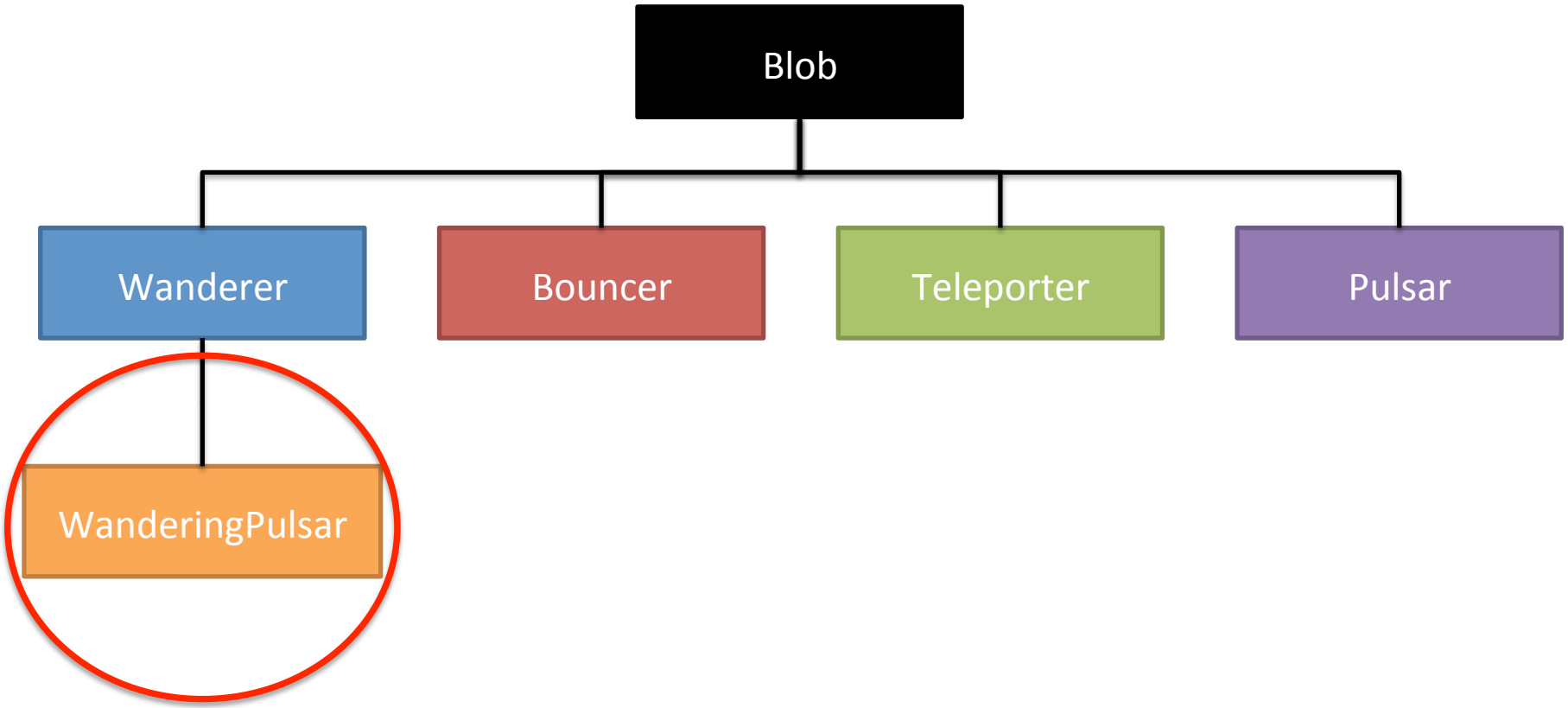
Pick a random radius growth rate between 1 and 2

Update radius on *step()*

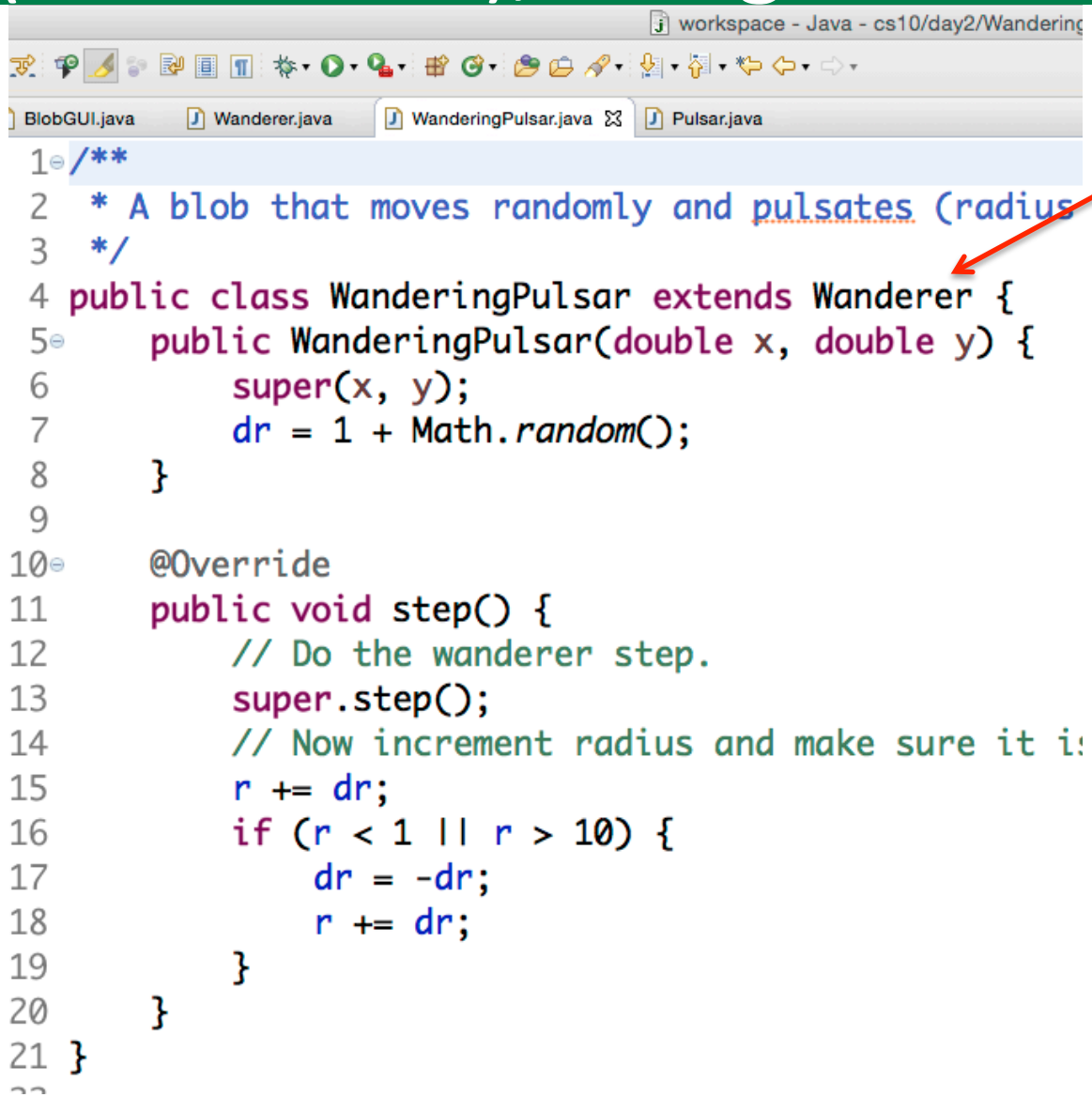
Make sure radius doesn't get too big or become negative

If too big or too small, reverse sign on dr to change growing or shrinking rate

A WanderingPulsar inherits from Wanderer (so it moves), and grows and shrinks



A WanderingPulsar inherits from Wanderer (so it moves), and grows and shrinks

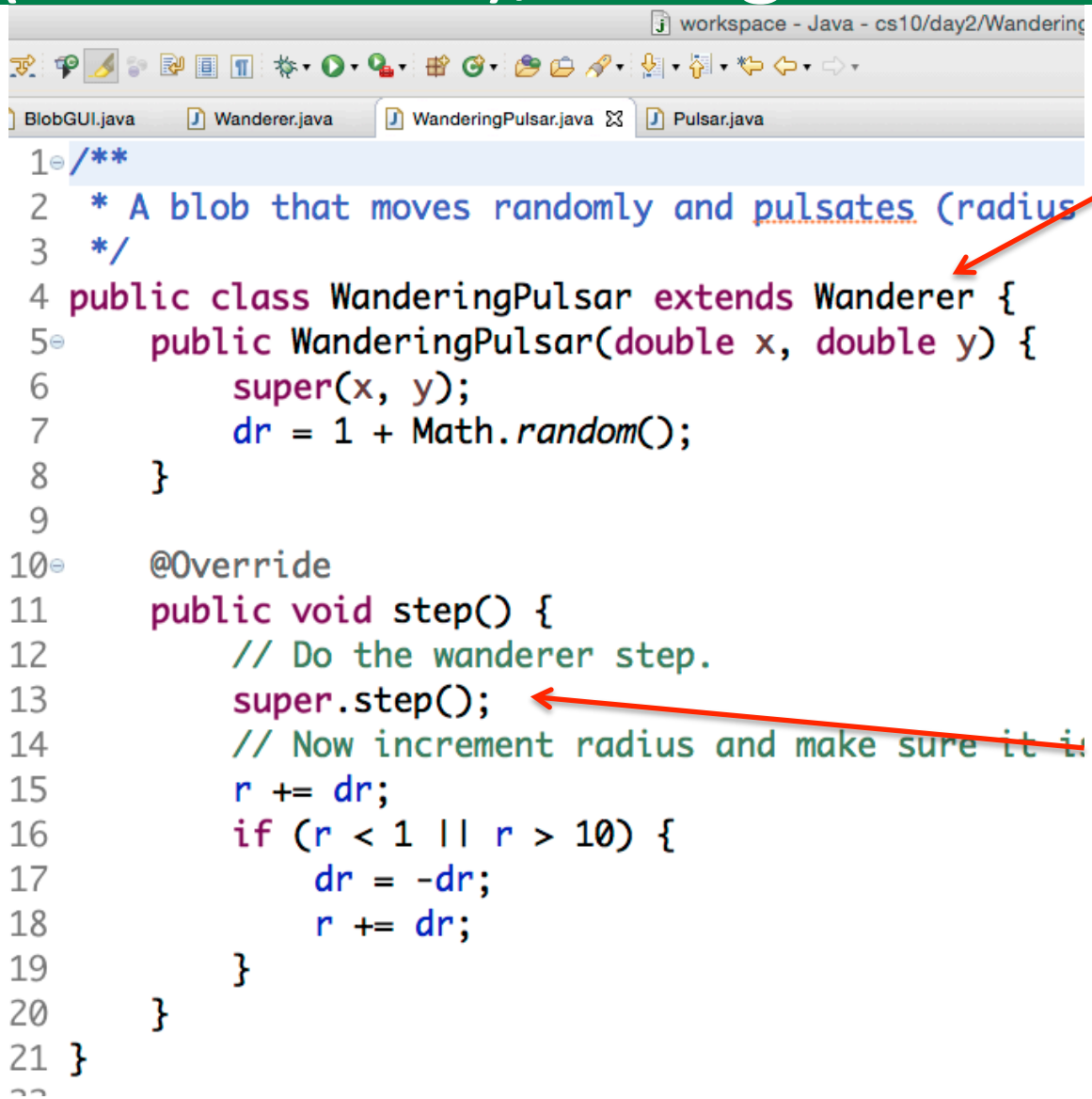


Inherit from Wanderer

In Java, can only inherit from *one* super class

We'll need to duplicate a little code to incorporate growing/shrinking

A WanderingPulsar inherits from Wanderer (so it moves), and grows and shrinks



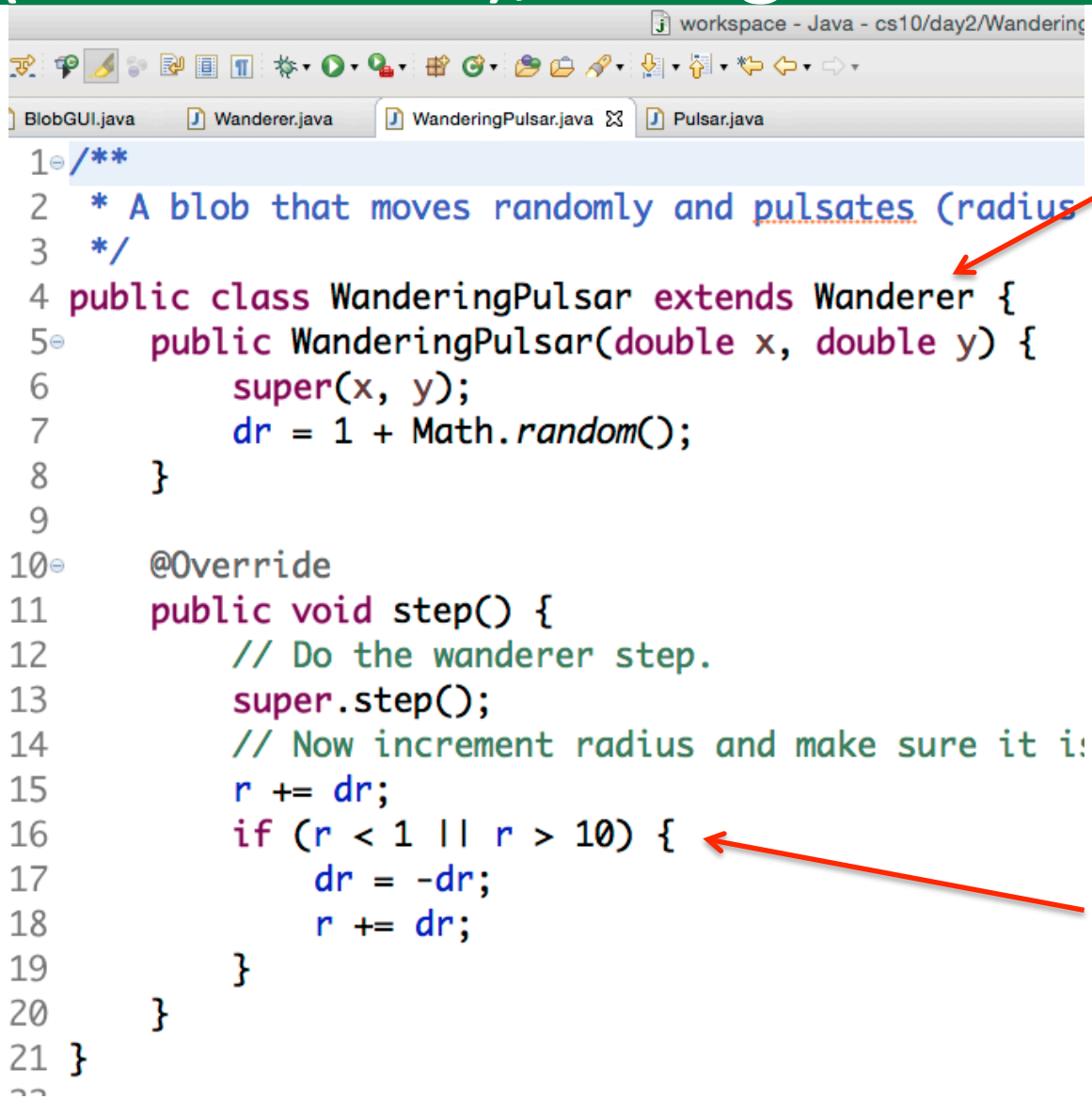
Inherit from Wanderer

In Java, can only inherit from *one* super class

We'll need to duplicate a little code to incorporate growing/shrinking

Call Wanderer's *step()* to move Blob

A WanderingPulsar inherits from Wanderer (so it moves), and grows and shrinks



Inherit from Wanderer

In Java, can only inherit from *one* super class

We'll need to duplicate a little code to incorporate growing/shrinking

Call Wanderer's *step()* to move Blob

Update radius, make sure doesn't get too big or become negative (had to duplicate code because couldn't also inherit from Pulsar)

Agenda

1. Objects vs. primitives
2. Inheritance
3. Quick review of Blobs
4. Using Inheritance to create different types of Blobs
5. Graphical User Interface (GUI) programming

Demo: BlobGUI.java

BlobGUI.java

- Press key to change Blob type
  - B for Bouncer
  - P for Pulsar
  - T for Teleporter
  - W for Wanderer
  - U for WanderingPulsar

- Click mouse to place Blob on screen

- Blob moves each time a timer ticks (every 100ms)

GUI programs do not run straight through, instead they respond to *events*

GUI programs

- Respond to user *events* such as mouse or key presses, rather than running straight through to completion
- Can also respond to non-user initiated events such as timer ticking
- When event occurs, it triggers code responsible for *handling* event, then goes back to waiting for next event
- Java has a lot of functionality, I've provided a simple "wrapper" in `DrawingGUI` to set up GUI display and provide functionality
- Extend `DrawingGUI` in your driver classes to inherit that functionality
- Feel free to look around `DrawingGUI`, but don't feel like you need to "own it"!

DrawingGUI tries to handle a lot of GUI set up messiness for us

Conceptual

- Java provides GUI code
- Somewhat complicated
- Learning the specifics of Java's GUI "machinery" not really the point of this course

- Wrapper that inherits from Java's *JFrame* machinery
- Sets up GUI and provides functionality we can use:
  - *handleMousePress()*
  - *handleKeyPress()*
  - *handleTimer()*
  - *repaint()*
  - *draw()*

- Inherit from *DrawingGUI*
- Override
  - *handleMousePress()*
  - *handleKeyPress()*
  - *handleTimer()*
  - *draw()*
- Get call backs when events (e.g., mouse press) happen
- Can trigger GUI *repaint()*
- *repaint()* calls *draw()*

DrawingGUI.java has a number of useful methods, these are the main ones

Useful DrawingGUI methods

Method	Description
`handleMousePress(x,y)`	• Responds to mouse press • Gets x, y coordinates where mouse was located when pressed

DrawingGUI.java has a number of useful methods, these are the main ones

Useful DrawingGUI methods

Method	Description
`handleMousePress(x,y)`	• Responds to mouse press • Gets x, y coordinates where mouse was located when pressed
`handleKeyPress(key)`	• Responds to keyboard key presses • Gets the character pressed

DrawingGUI.java has a number of useful methods, these are the main ones

Useful DrawingGUI methods

Method	Description
`handleMousePress(x,y)`	• Responds to mouse press • Gets x, y coordinates where mouse was located when pressed
`handleKeyPress(key)`	• Responds to keyboard key presses • Gets the character pressed
`handleTimer()`	• Callback when timer ticks • Ask the Blob to `step()`, then `repaint()`

DrawingGUI.java has a number of useful methods, these are the main ones

Useful DrawingGUI methods

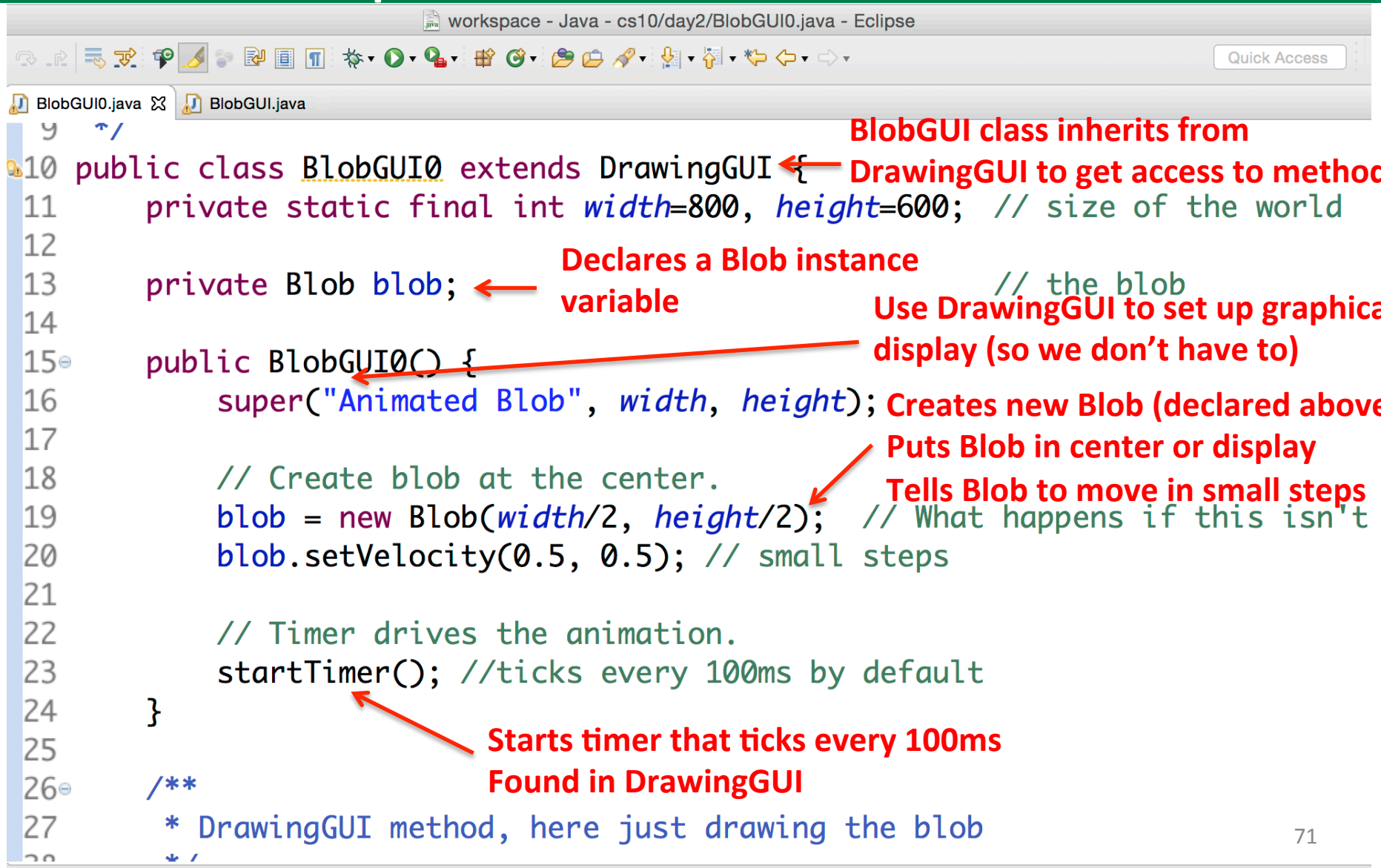
Method	Description
`handleMousePress(x,y)`	• Responds to mouse press • Gets x, y coordinates where mouse was located when pressed
`handleKeyPress(key)`	• Responds to keyboard key presses • Gets the character pressed
`handleTimer()`	• Callback when timer ticks • Ask the Blob to `step()`, then `repaint()`
`repaint()`	• Refreshes the GUI display • We call it when the underlying state changes (e.g., Blob moves) • OS may also call it to manage windows • `repaint()` calls our `draw()` function each time it is called

DrawingGUI.java has a number of useful methods, these are the main ones

Useful DrawingGUI methods

Method	Description
`handleMousePress(x,y)`	• Responds to mouse press • Gets x, y coordinates where mouse was located when pressed
`handleKeyPress(key)`	• Responds to keyboard key presses • Gets the character pressed
`handleTimer()`	• Callback when timer ticks • Ask the Blob to `step()`, then `repaint()`
`repaint()`	• Refreshes the GUI display • We call it when the underlying state changes (e.g., Blob moves) • OS may also call it to manage windows • `repaint()` calls our `draw()` function each time it is called
`draw(Graphics)`	• This is how we draw our portion of the screen (e.g., the Blob) • Inherit from `DrawingGUI` base class • Should call the Blob's `draw()` function to display the Blob (Blob's `draw()` calls `fillOval()` to display the Blob)

BlobGUI0.java: DrawingGUI handles most of the hard parts, inherit from it and use it

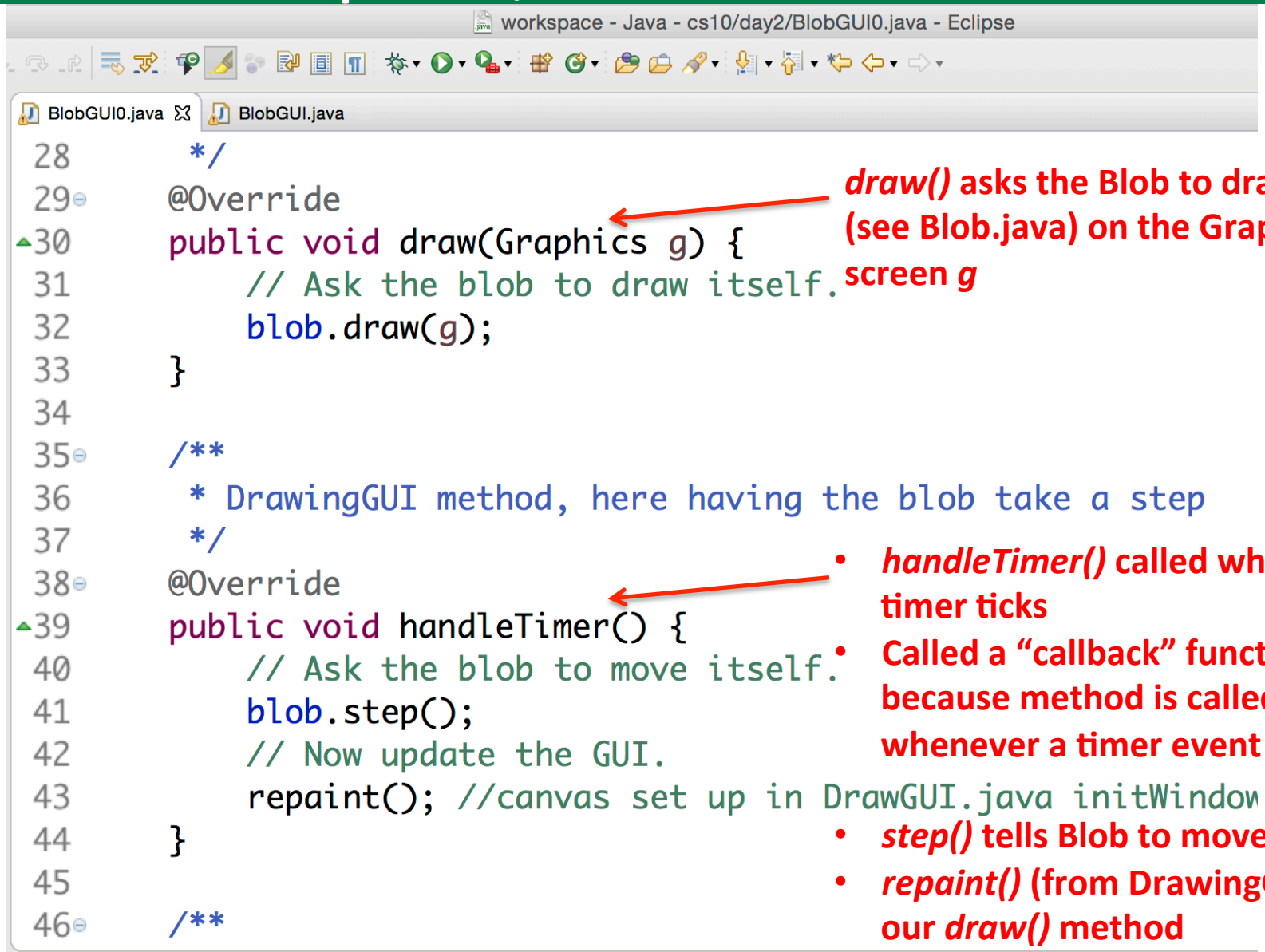


BlobGUI0.java: DrawingGUI handles most of the hard parts, inherit from it and use it

```
64      public static void main(String[] args) {
65          SwingUtilities.invokeLater(new Runnable() {
66              public void run() {
67                  new BlobGUI0();
68              }
69          });
70      }
71  }
```

- ***main()* is kind of ugly, but it's common "boilerplate"**
- **Calls our constructor (from last slide)**
- **Starts the GUI running**
- **Waits for events such as mouse/key presses or timer ticks**

BlobGUI0.java: DrawingGUI handles most of the hard parts, inherit from it and use it

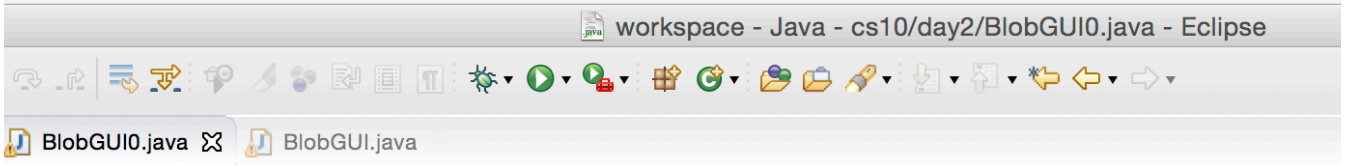


draw() asks the Blob to draw itself (see Blob.java) on the Graphics screen *g*

- *handleTimer()* called whenever timer ticks
- Called a "callback" function, because method is called back whenever a timer event happens

- *step()* tells Blob to move
- *repaint()* (from DrawingGUI) calls our *draw()* method

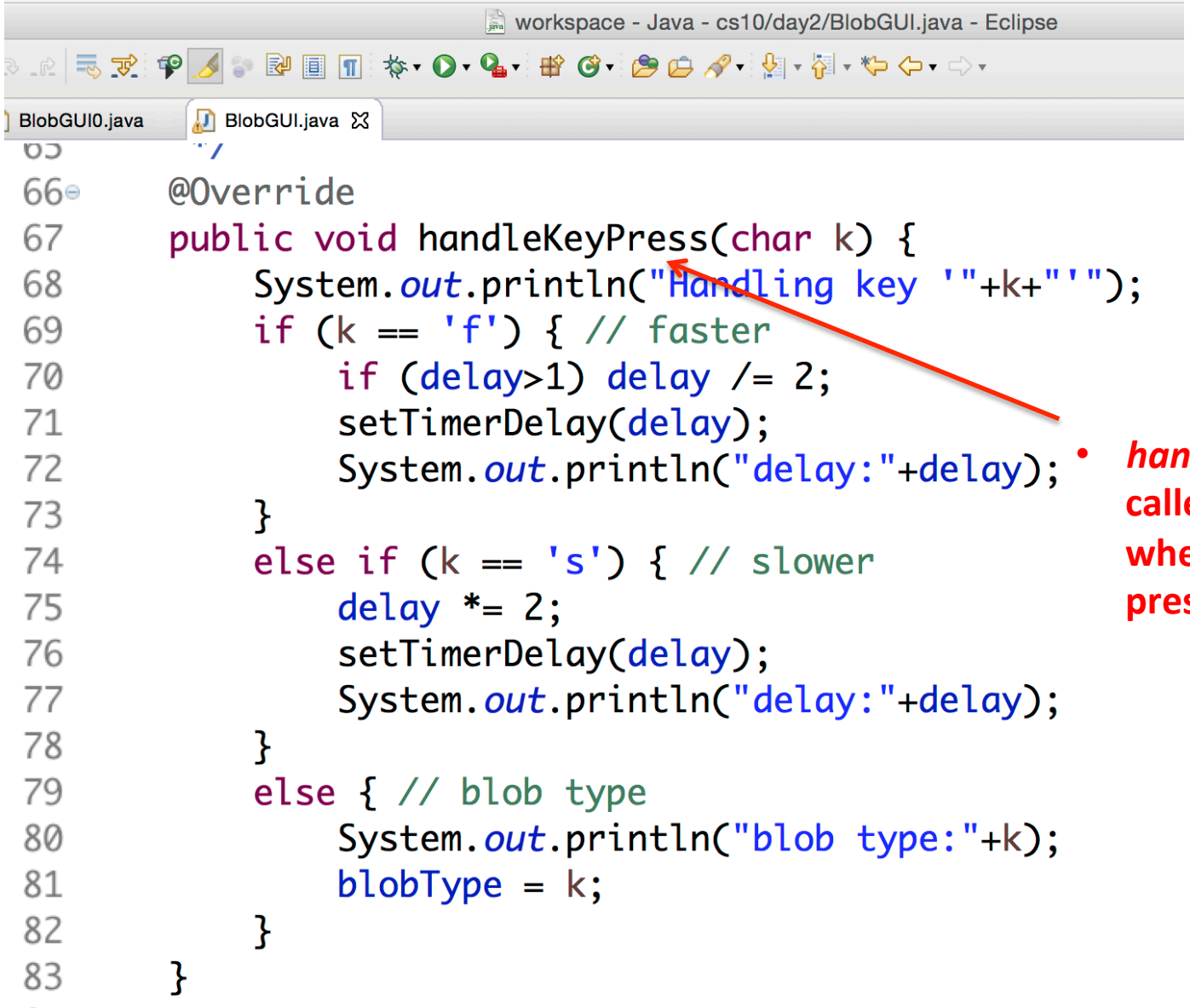
BlobGUI0.java: DrawingGUI handles most of the hard parts, inherit from it and use it




```
45
46      /**
47       * DrawingGUI method, here either detecting if
48       */
49      @Override
50      public void handleMousePress(int x, int y) {
51          if (blob.contains(x, y)) {
52              // Clicked on it
53              System.out.println("back off!");
54          }
55          else {
56              // Moved it
57              blob.setX(x);
58              blob.setY(y);
59              // Redraw with moved blob
60              repaint();
61          }
62      }
```

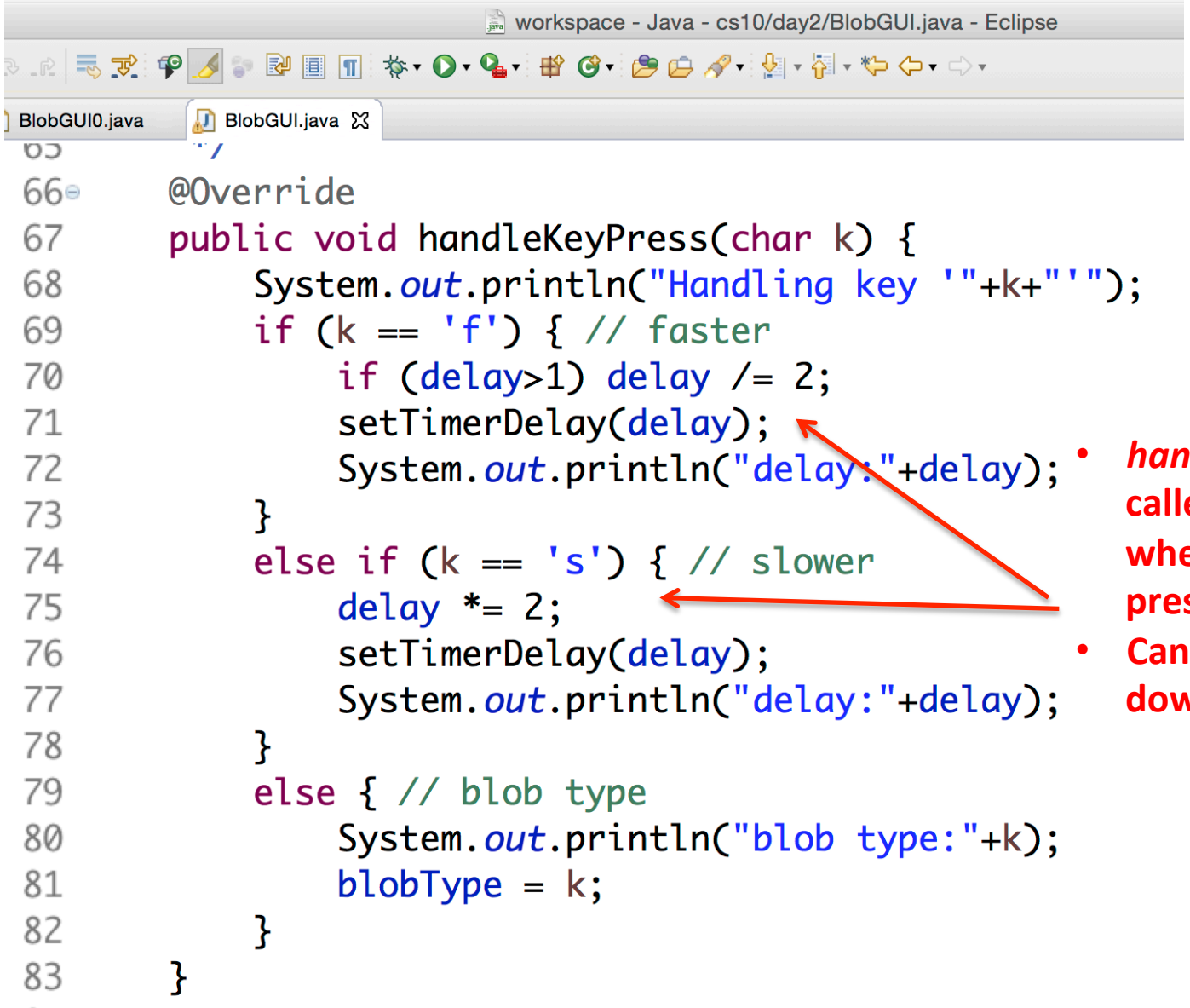
- ***handleMousePress()*** **called whenever mouse is pressed**
- **Checks to see if *x,y* coordinates where mouse pressed is inside blob**
  - **If true, print "back off!"**
  - **Else move Blob to location of mouse press**
- **Call *repaint()* which will in turn call our *draw()***

BlobGUI.java: Adds more functionality, monitors for key press, changes Blob type



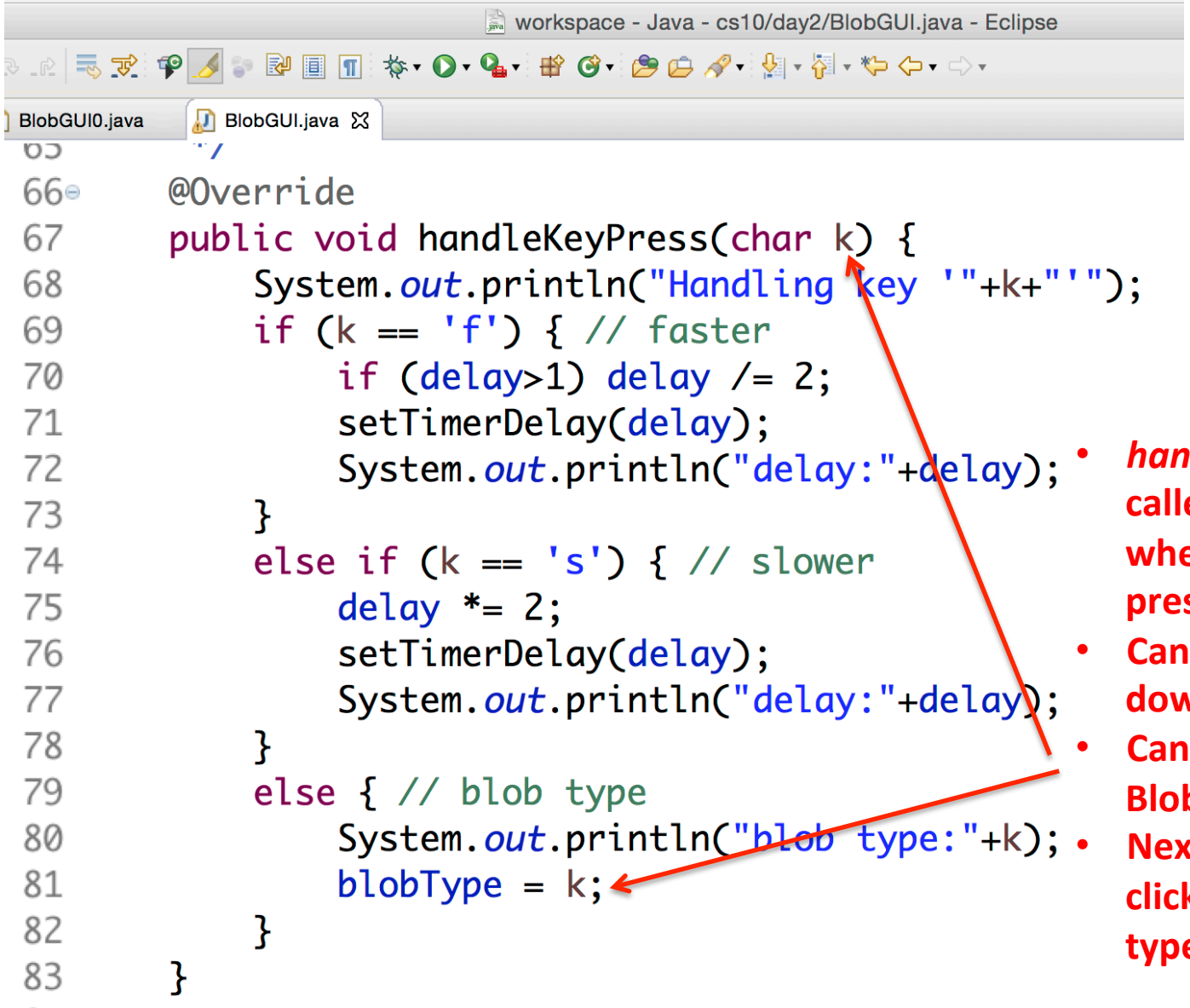
- ***handleKeyPress()*** **called back whenever key is pressed**

BlobGUI.java: Adds more functionality, monitors for key press, changes Blob type



- ***handleKeyPress()*** **called back whenever key is pressed**
- **Can speed up/slow down timer**

BlobGUI.java: Adds more functionality, monitors for key press, changes Blob type



- ***handleKeyPress()*** **called back whenever key is pressed**
- **Can speed up/slow down timer**
- **Can also change Blob type**
- **Next time mouse clicked a blob of type *k* is shown**

BlobGUI.java: Adds more functionality, monitors for key press, changes Blob type

```
31    @Override
32    public void handleMousePress(int x, int y) {
33        if (blob.contains(x, y)) {
34            System.out.println("back off!");
35            return;
36        }
37        if (blobType=='0') {
38            blob = new Blob(x,y);
39        }
40        else if (blobType=='b') {
41            blob = new Bouncer(x,y,width,height);
42        }
43        else if (blobType=='p') {
44            blob = new Pulsar(x,y);
45        }
46        else if (blobType=='t') {
47            blob = new Teleporter(x,y,width,height);
48        }
49        else if (blobType=='w') {
50            blob = new Wanderer(x,y);
51        }
52        else if (blobType=='u') {
53            blob = new WanderingPulsar(x,y);
54        }
55        else {
56            System.err.println("Unknown blob type "+blobType);
57        }
58
59        // Redraw with new blob
60        repaint();
61    }
```

When mouse clicked, set Blob to new Blob type based on last key press, then repaint

Old Blob is garbage collected (memory returned to OS)