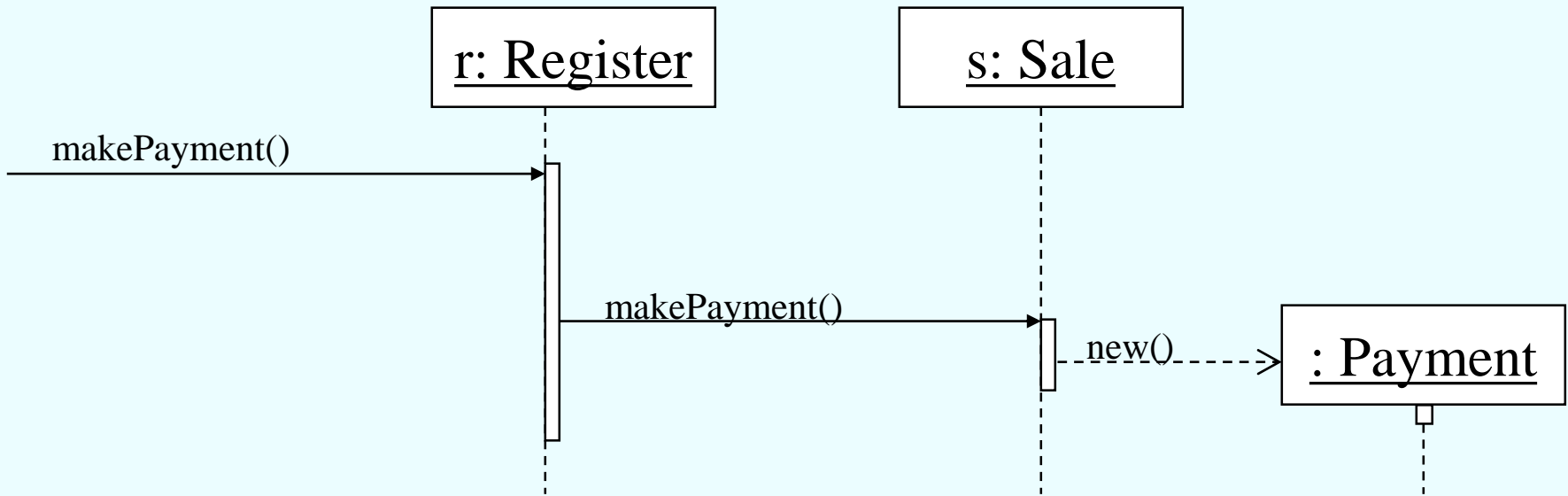


# Sequence Diagram

- A UML diagram used to show how objects interact. Example:



- The above starts with a Register object, *r*, receiving a `makePayment` message.
- *r* then sends `makePayment` to a Sale object, *s*.
- *s* then creates a Payment object (its constructor executes) and then *s* returns control back to *r* which then returns control back to whatever sent the first `makePayment` message.

# Sequence Diagram

- Objects that pre-exist a collaboration (& classes too) are represented horizontally across the top of the diagram
- A *lifeline* is a dashed line extending down from the object/class
- *Time* is represented vertically down the diagram. Time moves forward as you go downwards

# Sequence Diagram

- The time an object is active is indicated by a narrow rectangle
  - Called an *activation bar* or *focus of control*
  - An object is considered active from the point in time it receives a message to the point in time when it returns to its caller or stops.
  - An active object that sends a synchronous message is suspended until control returns.

ACS-3913 is concerned with  
synchronous messages only


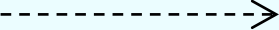
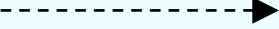
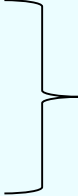
# Sequence Diagram

In ACS-3913 we are concerned with:

- synchronous messages - an object sends a message and waits (i.e. execution is suspended) until there is a response (reply) from the called object
- creation of objects
- replies - sometimes its useful to show a return of control with possibly a returned value
- **always** show the focus of control!

# Sequence Diagram

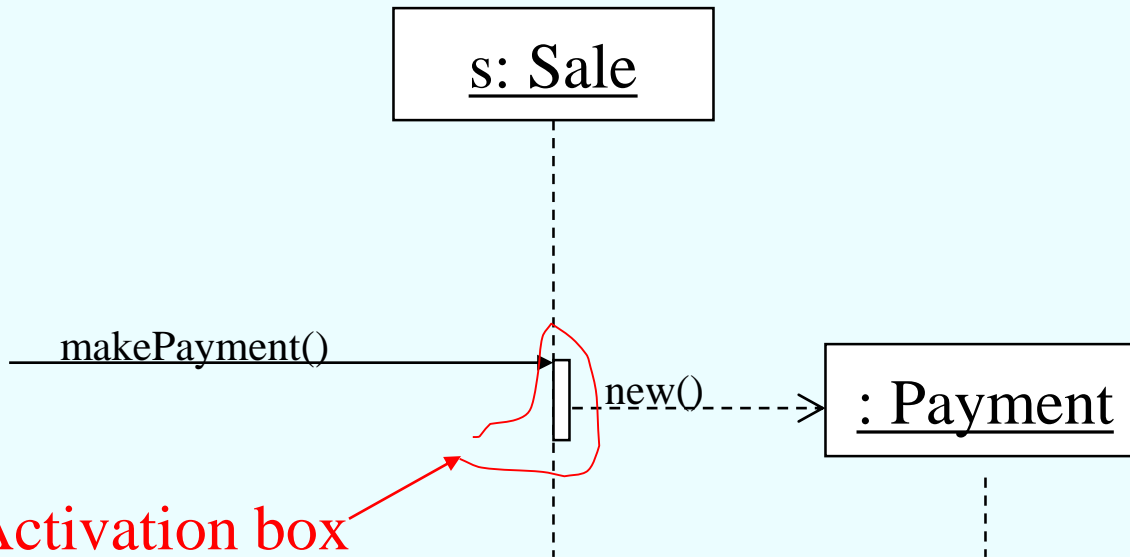
## Message types and their lines

- Synchronous 
  - Creation 
  - Reply 
- 
- Note these are dashed lines

# Sequence Diagram

Below:

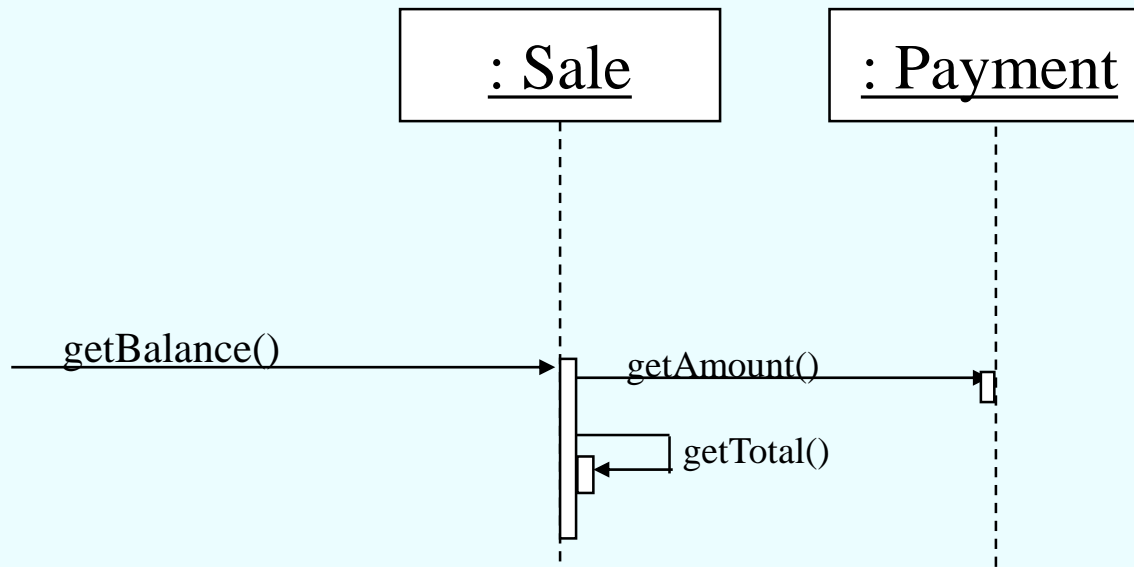
s receives the synchronous message named makePayment. The makePayment method of Sale executes, and creates a Payment object.



Activation box

Represents that makePayment has control (executes).  
makePayment is suspended while a Payment object is created...  
while Payment's constructor executes.

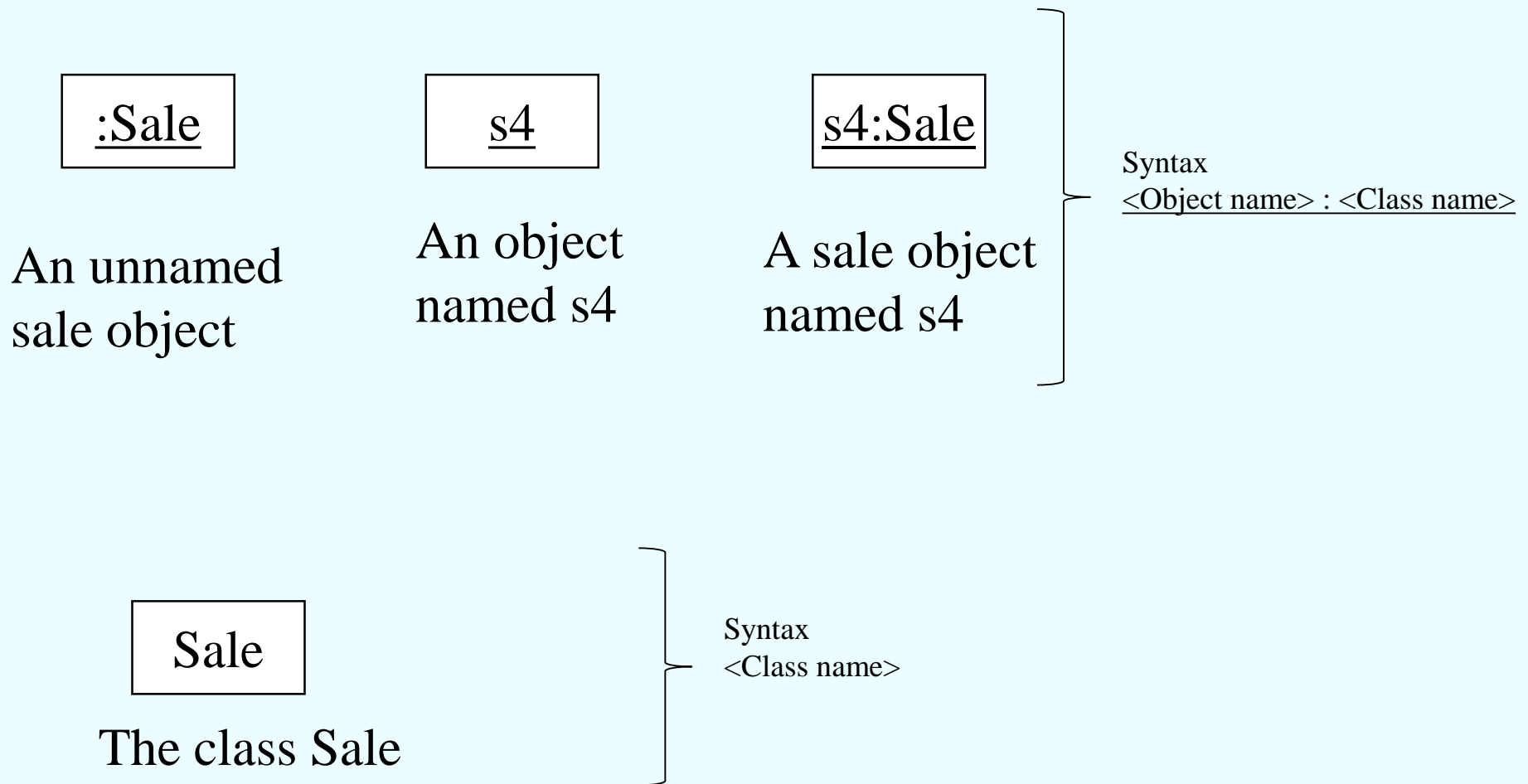
# Reflexive messages



A message where an object sends a message to itself  
i.e. an object calls one of its own methods

Note the additional activation box for `getTotal()`  
overlaid on that for `getBalance()`

# Sequence Diagrams: Objects & Classes





## Example:

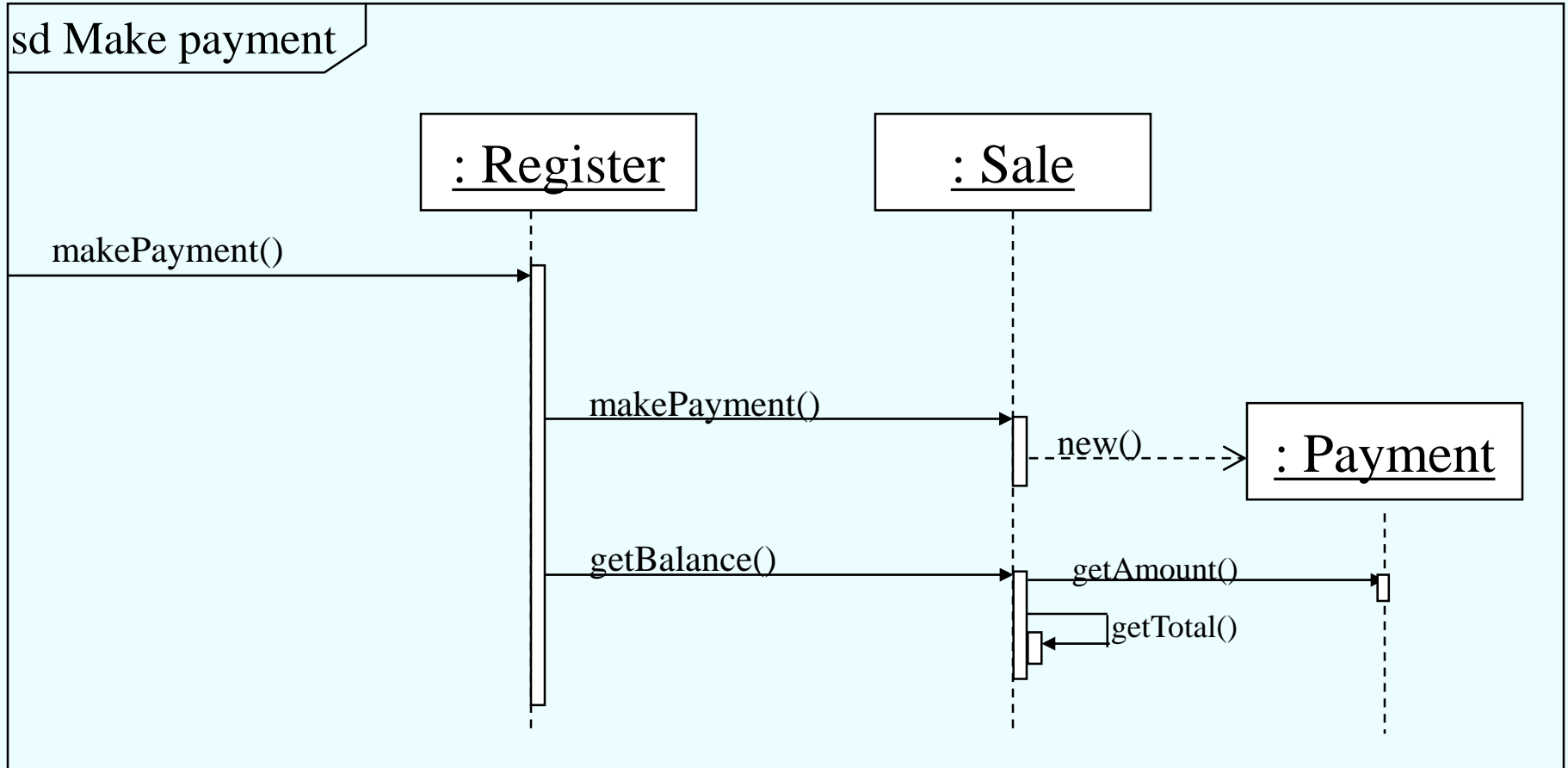
What is the Sequence Diagram for when makePayment received by a register?

```
public class Register
{
    private ProductCatalog catalog;
    private Sale sale;
    ...
    public void makePayment (int cashTendered )
    {sale.makePayment(cashTendered );
        System.out.println(
            "made a payment, change due= "
            + sale.getBalance() );
    }
}

public class Payment {
    private int amount;
    public Payment( int cashTendered )
    { amount = cashTendered ;
    }
    public int getAmount()
    { return amount;
    }
}

public class Sale
...
private Payment payment;
public int getBalance()
{
    return payment.getAmount() - getTotal() ;
}
public int getTotal()
{
    ...
    return total;
}
public void makePayment (int cashTendered )
{
    payment = new Payment(cashTendered );
}
}
```

# Sequence Diagram for makePayment received by a register



Some message starts the collaboration

Objects that pre-exist the collaboration are shown at top of diagram.

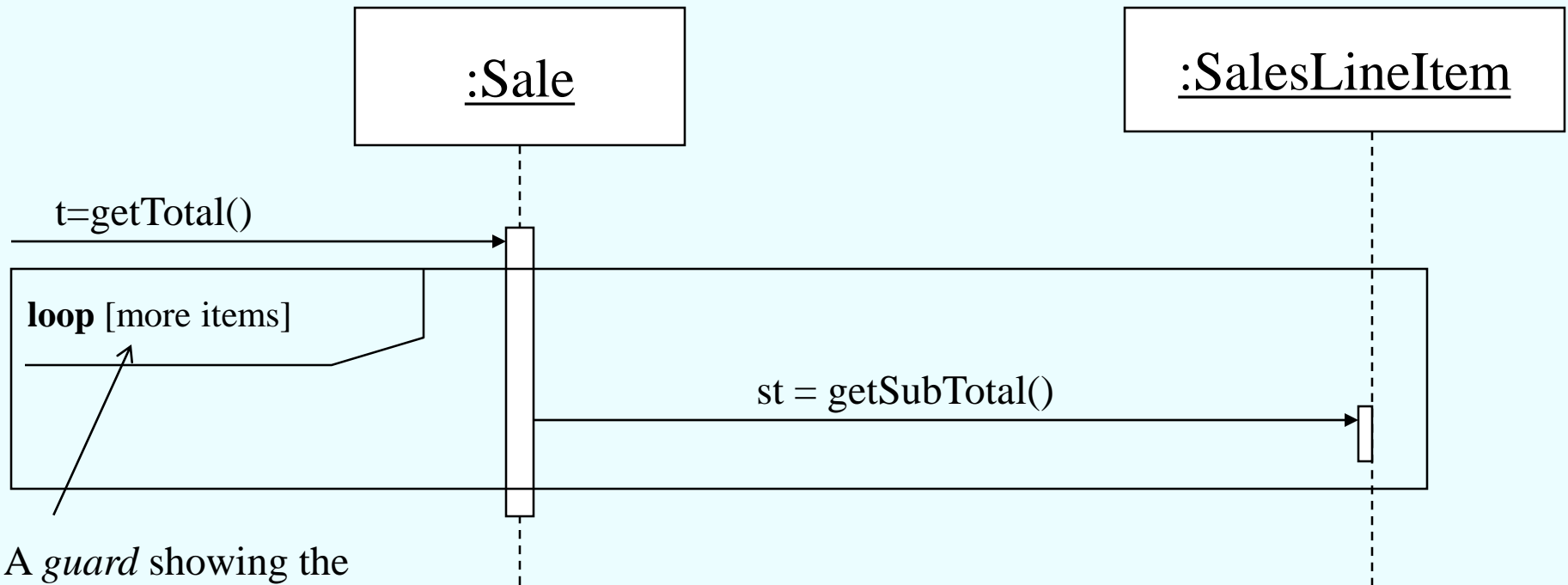
Other objects are shown where/when they are created

# Iteration

Consider the following code segment where a Sale is interacting with its Line Items in order to obtain the total value of the sale.

```
public int getTotal()
{
    ...
    int total = 0;
    while( ...)
    {
        ...
        int lineTotal = sli.getSubtotal() ;
        total = total + lineTotal ;
    }
    return total;
}
```

# Iteration



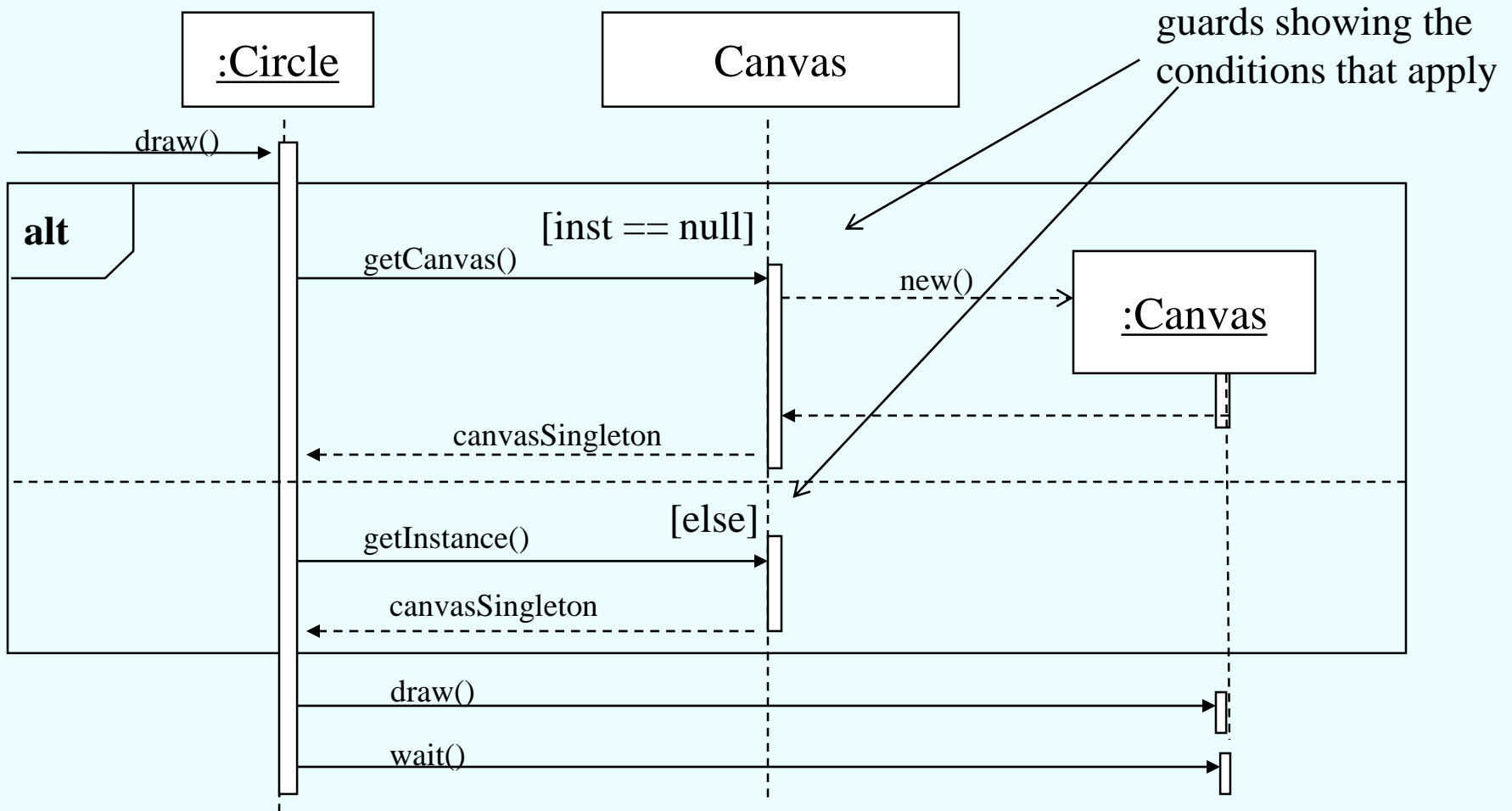
A *guard* showing the condition that applies

***loop*** is a keyword to specify iteration in a sequence diagram

Note the box/frame around the involved messages

# Decision Structures

Consider the BlueJ Shapes example: Below is a sequence diagram for when a circle is asked to draw itself. Canvas is a “singleton” ... when you ask for the instance of Canvas, there are two ways it can complete.



Behaviour in Singleton Pattern shown using a decision structure (i.e. **alt**)