Programación Orientada a Objetos I - Especificación e implementación de tipos de datos

Algoritmos y Estructuras de Datos II

Año 2025

Dr. Pablo Ponzio

Universidad Nacional de Río Cuarto

CONICET

Horarios de la materia

- Teóricos:
  - Martes de 16 a 18 hs. - Anfiteatro 2 Pabellón 2
  - Miércoles de 14 a 16 hs. - Anfiteatro 2 Pabellón 2
- Prácticos:
  - Comisión 1:
    - Martes de 10 a 12 hs. - Sala 101 Pabellón 2
    - Jueves de 8 a 10 hs. - Sala 101 Pabellón 2
  - Comisión 2:
    - Miércoles de 12 a 14 hs. - Sala 110 Pabellón 2
    - Jueves de 14 a 16 hs. - Sala 110 Pabellón 2
  - Comisión 3:
    - Martes de 8 a 10 hs. - Sala 101 Pabellón 2
    - Jueves de 12 a 14 hs. - Sala 101 Pabellón 2

Requisitos para la aprobación

- Para la regularidad:
  - Aprobación de dos exámenes parciales escritos
  - Aprobación de un trabajo práctico grupal
    - Con una instancia de defensa oral individual
    - Una instancia de recuperación en cada evaluación
- Para la promoción:
  - Aprobar todos los exámenes parciales y la defensa del trabajo práctico con nota mayor o igual a 7
  - Aprobar un coloquio que se tomará a los estudiantes que cumplan los requisitos anteriores

Motivación: Calidad de software

- La Ingeniería de Software define cuales son las cualidades deseables de un sistema de software
  - Estas se denominan atributos de calidad del software
- Los atributos de calidad se dividen en atributos internos y externos
  - Los atributos de calidad externos son aquellos visibles a los usuarios del sistema
    - Algunos de los más importantes son:
      - La confiabilidad: Los usuarios pueden depender del programa para realizar alguna tarea
      - El rendimiento: Los programas deben ejecutarse rápidamente
      - La facilidad de uso
      - etc..

Motivación: Calidad de software

- Los desarrolladores del sistema usualmente se centran en los atributos de calidad internos
  - Las cualidades internas de un programa usualmente determinan sus cualidades externas, por lo que son fundamentales
- Algunos de los atributos de calidad internos más importantes son:
  - Corrección funcional: El programa se comporta de acuerdo a lo establecido en sus especificaciones
    - La corrección funcional es crucial para obtener programas confiables
  - Eficiencia: Los programas deben economizar tanto como sea posible en el uso de los recursos computacionales (CPU, memoria, etc.)
    - Usualmente, la eficiencia determina en gran medida el rendimiento del programa

Motivación: Calidad de software

- Algunos de los atributos de calidad internos más importantes son:
  - Mantenibilidad: El ciclo de vida de un software puede durar años o décadas. Por lo tanto, el mantenimiento suele ser la tarea más costosa (más del 60% del costo total)
  - Se suele dividir en otros dos atributos:
    - Evolucionabilidad: Los programas deben ser fáciles de extender con nuevas funcionalidades
      - Es importante porque los requisitos de los programas actuales cambian frecuentemente
      - Usualmente la tarea más costosa, pudiendo insumir más de un 50% del costo del mantenimiento
    - Reparabilidad: Debe ser fácil modificar el programa y/o corregir errores en el mismo

Motivación: Calidad de software

- Algunos de los atributos de calidad internos más importantes son:
  - Reusabilidad: El programa (o partes del mismo) deberían ser reutilizables en otros contextos
  - Comprensibilidad: El código debe ser fácil de entender para cualquier desarrollador
    - Durante el mantenimiento se gasta más tiempo en entender el código que en modificarlo
    - Muchas veces el desarrollador que modifica el código no es el mismo que lo escribió originalmente
    - Es muy difícil lograr las restantes cualidades internas (y las externas) si el código no es comprensible

¿Sobre qué trata este curso?

- Estudiaremos algunas de las metodologías y técnicas que favorecen el desarrollo de software de calidad
- Vamos a estudiar la programación orientada a objetos
  - Es el paradigma de programación más usado en la actualidad
  - Una de las razones principales es que favorece el desarrollo de programas modulares
  - Usaremos el lenguaje Java, uno de los más relevantes en la actualidad
- Pondremos énfasis en el desarrollo de programas modulares
  - La modularización se basa en descomponer problemas grandes de programación en problemas más pequeños, más fáciles de resolver que el problema original
  - Los subproblemas son resueltos por programas más pequeños, independientes entre si, denominados módulos
  - Finalmente, los módulos se combinan para dar lugar a la solución del problema original

¿Sobre qué trata este curso?

- La modularización es crucial para el desarrollo de software de calidad:
  - A medida que los programas crecen en tamaño el código monolítico se vuelve imposible de entender
  - Permite que distintos programadores trabajen en distintas partes del sistema al mismo tiempo
    - Facilita el mantenimiento: las modificaciones y/o extensiones se mantienen acotadas a unos pocos módulos, dejando intactos a los restantes
    - Facilita la comprensión del código ya que permite enfocarse en las partes relevantes a una tarea, ignorando el resto
- Es fundamental diseñar bien los módulos para que luego puedan interactuar correctamente
  - Contar con especificaciones es fundamental para esta tarea
- Estudiaremos como especificar programas orientados a objetos y la teoría de tipos abstractos de datos

¿Sobre qué trata este curso?

- Estudiaremos conceptos básicos de testing para verificar que los módulos respetan su especificación
- Estudiaremos técnicas para el desarrollo de software eficiente
  - Estudiaremos como analizar y comparar la eficiencia de los algoritmos
    - Nos permitirá elegir algoritmos eficientes para implementarlos en nuestros programas
- Estudiaremos implementaciones eficientes de los tipos abstractos de datos más usados: Listas, conjuntos, diccionarios, grafos, etc.
    - Útiles para almacenar datos y acceder a ellos eficientemente, en todo tipo de aplicaciones
    - Compararemos las características de eficiencia de las distintas implementaciones
    - Nos permitirá elegir la más adecuada para cada aplicación

Objetos y Clases

- Al programar en un lenguaje orientado a objetos estamos creando un modelo de una parte del mundo
- El modelo está conformado por objetos que ocurren en el dominio del problema
  - Ejemplos de objetos en diferentes tipos de aplicaciones:
    - Palabras y párrafos en un procesador de textos
    - Usuarios y mensajes en una red social
    - Jugadores y monstruos en un juego
- Una clase define el conjunto de objetos de un tipo particular

Definición de clases

- Las convenciones que usaremos para la definición de clases son las siguientes:

```
/**
 * Especificación informal breve del propósito de la clase
 */
public class ClassName
{
    atributos
    constructores
    métodos
}
```

- Primero, damos una especificación breve de la responsabilidad de la clase en un comentario
  - Para el desarrollador: Sirve para definir de manera precisa las funcionalidades que debe proveer la clase, y que se debe implementar
  - Para el cliente: Sirve para entender cuál es la utilidad de la clase
- Luego, viene el encabezado de la clase: public class ClassName
  - Una clase pública (modificador public) es visible desde otras clases
  - Por convención, los nombres de las clases comienzan con mayúsculas

Definición de clases

- Las convenciones que usaremos para la definición de clases son las siguientes:

```
/**
 * Especificación informal breve del propósito de la clase
 */
public class ClassName
{
   atributos
   constructores
   métodos
}
```

- Es muy importante elegir nombres para las clases que sean representativos de su funcionalidad
- Si bien seguir algunas de estas convenciones no es obligatorio, usar un estilo consistente durante todo el proyecto hace que el código sea más fácil de leer y de entender

Ejemplo: TicketMachine

- Problema: Implementar el software para una máquina de tickets de una estación de trenes
- Requisitos:
  - Todos los tickets tienen un costo único, y es fijo
  - El usuario ingresa dinero a la máquina, y puede requerir que esta imprima un ticket
    - La máquina sólo imprimirá un ticket en caso de que el dinero ingresado sea mayor o igual al costo del ticket
  - El usuario puede solicitar que la máquina le de vuelto si el dinero ingresado excede el costo del ticket

Atributos

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 */
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;
```

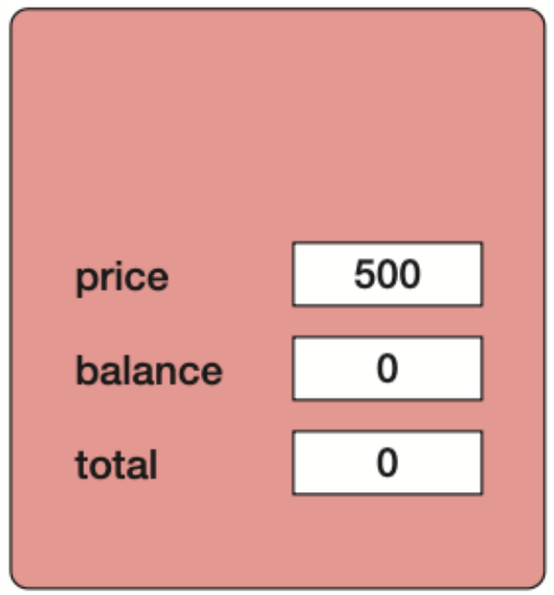
- Los atributos sirven para almacenar información relevante para los objetos de la clase
- Para las máquinas de tickets necesitamos guardar tres enteros:
  - `price` para guardar el costo (fijo) de los tickets
  - `balance` para tener la cantidad de dinero ingresada por el usuario
  - `total` para llevar el dinero recaudado por la máquina

Atributos

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 */
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;
```

- Los objetos tienen espacio de memoria reservado para guardar valores por cada uno de sus atributos
  - 3 `int`s para `TicketMachine`
- Los atributos pueden cambiar durante el ciclo de vida de un objeto, por ejemplo:
  - Cuando el usuario ingresa dinero el monto se suma a `balance`
  - Cuando se imprime un ticket, el precio del ticket se suma a `total`
- Los atributos definen el estado de los objetos

Objeto en memoria



Atributos

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 */
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;
```

- La sintaxis para la declaración de atributos es: `private tipo nombre`
  - `private` indica que los atributos sólo son accesibles en la clase que los define
    - No son accesibles en otras clases que la usan
    - Sirve para implementar el ocultamiento de información: Los clientes de una clase no deben depender de su implementación, sino de sus funcionalidades (definidas mediante métodos)
  - Convención: los nombres de los atributos comienzan con minúsculas
  - Es muy importante elegir nombres para los atributos que describan su propósito

Atributos

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 */
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;
```

- La sintaxis para la declaración de atributos es: `private tipo nombre`
  - `private` indica que los atributos sólo son accesibles en la clase que los define
    - No son accesibles en otras clases que la usan
    - Sirve para implementar el ocultamiento de información: Los clientes de una clase no deben depender de su implementación, sino de sus funcionalidades (definidas mediante métodos)

Prestar especial atención a la indentación del código en los ejemplos, y procurar mantener el mismo estilo en sus programas

Tipos de datos en Java

- Java es un lenguaje fuertemente tipado: toda variable tiene un tipo asociado
  - La ventaja es que el compilador puede detectar errores de programación sin necesidad de ejecutar el programa
  - La desventaja es que el programador debe definir los tipos de las variables
- El tipo de una variable determina el conjunto de valores que puede tomar la misma
  - Ejemplo: `int distance` indica que el valor de `distance` debe ser un número entero
- Los tipos en Java se dividen en dos categorías: tipos primitivos y objetos

Tipos Primitivos

- Tipos primitivos: tipos predefinidos en el lenguaje que definen conjuntos de valores

  - `int`: números enteros
    - Valores: `3`, `123`, `-35`,...
    - Operaciones: `+`, `-`, `*`, `/`, `==`, `!=`, ...
  - `float`: números de punto flotante
    - Valores: `0.32f`, `123.1f`, `-758.032f`, ...
    - Operaciones: `+`, `-`, `*`, `/`, `==`, `!=`, ...
  - `boolean`: valores de verdad
    - Valores: `true`, `false`
    - Operaciones: `!` (not), `&&` (and), `||` (or), `==`, `!=`, ...
  - `char`: caracteres
    - Valores: 'a', 'b', '/', '&', ...
    - Operaciones: `==`, `!=`, ...
  - y otros [1]...

[1] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Tipos de Objetos

- A excepción de los tipos primitivos, los tipos restantes definen conjuntos de objetos
  - `String`: Define el conjunto de cadenas de caracteres
    - Tipo predefinido en el lenguaje
    - Valores: `"hola mundo"`, `"red"`, `"blue"`, `"yellow"`, ...
      - Java provee la notación anterior para definir fácilmente `String`s
        - Pero estos son representados internamente como objetos
  - Tipos definidos por el usuario (ej. `TicketMachine`)
  - Tipos definidos en bibliotecas de Java y por otros desarrolladores, etc...
- Los tipos primitivos y los objetos se comportan de maneras distintas:
  - El programador debe reservar memoria para los objetos (con `new`)
  - Para los objetos, la asignación y el pasaje de parámetros es por referencia; para los tipos primitivos es por valor

Constructores

```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    /**
     * @pre 'cost' > 0.
     * @post Create a machine that issues tickets
     *   with a price of 'cost'.
    */
    public TicketMachine(int cost)
    {
      // TODO: Implementar
    }
```

- Los constructores son métodos responsables de crear los objetos e inicializarlos
- Llevan el mismo nombre de la clase, son públicos, y pueden tomar parámetros
- No tienen tipo de retorno
- Cómo cualquier procedimiento, su especificación consiste de una pre y una postcondición (ver más adelante)

Constructores

```
/**
 * @pre 'cost' > 0.
 * @post Create a machine that issues tickets
 *   with a price of 'cost'.
*/
public TicketMachine(int cost)
{
    price = cost;
    balance = 0;
    total = 0;
}
```

- En Java, = es el operador de asignación: asigna el valor resultante de evaluar una expresión a una variable
  - En realidad, se asigna una copia del valor resultante de evaluar la expresión
  - La asignación debe respetar los tipos o el programa será rechazado por el compilador
- Notar que el costo del ticket es externo a la máquina
  - El valor concreto para el costo se pasa como parámetro durante la creación de la máquina
  - Parametrizar el costo permite crear máquinas que venden tickets con distintos costos (distintos objetos)
  - Uno de los roles principales de los atributos es permitirles a los objetos almacenar información externa pasada como parámetro

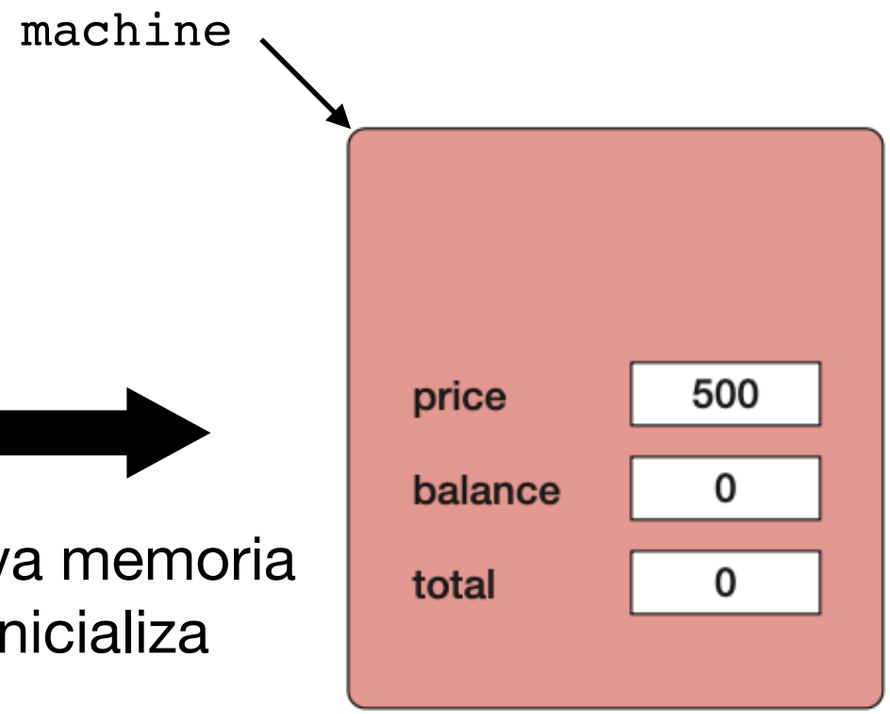
Creación de objetos

```
public TicketMachine(int cost)
{
    price = cost;
    balance = 0;
    total = 0;
}
```

Invocación al constructor:

```
TicketMachine machine = new TicketMachine(500);
```

reserva memoria
e inicializa



objeto en memoria

- El pasaje de parámetros para tipos primitivos en Java es por valor: el parámetro formal toma como su valor una copia del parámetro actual
- Para objetos el pasaje de parámetros es por referencia
  - Lo veremos más adelante...

Métodos

- Los métodos definen las funcionalidades que proveen los objetos de la clase. Vamos a implementar los siguientes métodos en `TicketMachine`

```java
/**
 * @pre 'cost' > 0.
 * @post Create a machine that issues tickets with a price of 'cost'.
*/
public TicketMachine(int cost)
{
    // TODO: Implementar
}
```

```java
/**
 * @post Returns the price of a ticket.
 */
public int getPrice()
{
    // TODO: Implementar
}
```

```java
/**
 * @post Returns the amount of money already inserted
 * for the next ticket.
 */
public int getBalance()
{
    // TODO: Implementar
}
```

```java
/**
 * @post If enough money has been inserted, print a
 * ticket to the console and reduce the current balance
 * by the ticket price. Returns 'true' if successful;
 * otherwise, it does nothing and returns 'false'.
 */
public boolean printTicket()
{
    // TODO: Implementar
}
```

```java
/**
 * @post Returns the money in the balance and clears
 * the balance.
 */
public int refundBalance()
{
    // TODO: Implementar
}
```

Repaso: Abstracción procedural

- Uno de los mecanismos de abstracción más importantes de los lenguajes imperativos es la abstracción procedural

```
float sqrt (float coef) {
   // REQUIRES: coef > 0
   // EFFECTS: Returns an approximation to the square root of coef
   float ans = coef/2.0;
   int i = 1;
   while (i < 7) {
      ans = ans - ((ans * ans - coef)/(2.0*ans));
      i = i + 1;
   }
   return ans;
}
```

- Hay dos tipos de abstracciones en juego en los procedimientos
- Abstracción por parametrización: Nos abstraemos de la identidad de los datos y los reemplazamos por parámetros
  - Generaliza el código y lo hace reusable en distintas situaciones
- Abstracción por especificación: Nos basamos en la especificación para conocer el funcionamiento y usar el procedimiento
  - Nos permite abstraernos de la implementación

Repaso: Especificación de procedimientos

```
float sqrt (float coef) {
    // REQUIRES: coef > 0
    // EFFECTS: Returns an approximation to the square root of coef
```

- Típicamente, el comportamiento de un procedimiento se especifica en términos de una precondición y una postcondición:
  - La precondición describe las condiciones bajo las cuales el procedimiento funciona correctamente
    - Si se invoca el procedimiento en un estado que viola su precondición, no podemos garantizar nada sobre su ejecución
      - Puede fallar, devolver un valor inválido, terminar el programa, etc.
  - La postcondición especifica las propiedades que se cumplen al finalizar la ejecución del procedimiento, si se invoca en un estado que satisface la precondición
- En este curso, vamos especificar los programas orientados a objetos en lenguaje natural, tomando una notación similar a la propuesta por Liskov (ver Bibliografía)

Repaso: Especificación de procedimientos

```
float sqrt (float coef) {
    // REQUIRES: coef > 0
    // EFFECTS: Returns an approximation to the square root of coef
```

- Otra forma de entender la especificación es pensar que define un contrato entre quien implementa un procedimiento `P` y los clientes (usuarios) de `P`

	Desarrollador	Cliente
`pre`	Puede asumir que `pre` vale antes de invocar a `P`	Debe garantizar que `pre` vale antes de invocar a `P`
`post`	Debe garantizar que `post` se satisface cuando finaliza la ejecución de `P`	Puede asumir que `post` se cumple al finalizar la ejecución de `P`

Especificación de métodos

```
/**
 * @pre 'amount' > 0.
 * @post Receives an amount of money from a customer.
 */
public void insertMoney(int amount)
```

- Antes del perfil de cada método, especificaremos su comportamiento con un texto breve, siguiendo una notación similar a la propuesta por Liskov (ver Bibliografía)
- `@pre` describe la precondición: Condiciones que se deben satisfacer para poder ejecutar el método
  - Por ejemplo, `insertMoney` sólo puede ejecutarse si se le pasa una cantidad de dinero mayor a cero
  - Usualmente se omite `@pre` si la precondición es true
- `@post` describe la postcondición: Condiciones que podemos asumir como válidas luego de la ejecución del método, si lo ejecutamos en un estado que satisface `@pre`
  - Por ejemplo, `insertMoney` cambiará el estado de la máquina de tickets para registrar que recibió la cantidad de dinero dada (`amount`)

Especificación de métodos

```
/**
 * @pre 'amount' > 0.
 * @post Receives an amount of money from a customer.
 */
public void insertMoney(int amount)
```

- La especificación de los métodos es fundamental, ya que describe con un buen grado de precisión el comportamiento deseado del método
  - Utilidad para el programador:
    - Evita errores por una mala comprensión de la especificación (si no la escribimos)
  - Utilidad para los clientes:
    - Ayuda a entender rápidamente la funcionalidad del método
    - Evita tener que estudiar minuciosamente la implementación para saber qué hace el método
- Usualmente, unas pocas oraciones deberían ser suficientes para describir el comportamiento de un método
  - Clase muy complejas pueden requerir especificaciones más detalladas, aunque estos casos suelen ser raros (una pocas clases de un programa grande)

Especificación de métodos

```
/**
 * @pre 'amount' > 0.
 * @post Receives an amount of money from a customer.
 */
public void insertMoney(int amount)
```

- La especificación de los métodos es fundamental, ya que describe con un buen grado de precisión el comportamiento deseado del método
  - Utilidad para el programador:
    - Evita errores por una mala comprensión de la especificación (si no la escribimos)
  - Utilidad para los clientes:
    - Ayuda a entender rápidamente la funcionalidad del método
    - Evita tener que estudiar minuciosamente la implementación para saber qué hace el método
- Usualmente, unas pocas oraciones deberían ser suficientes para describir el

Vamos a poner especial énfasis en escribir especificaciones para todas las clases que implementemos en la materia

Perfil de un método

- Después de la especificación, los métodos tienen dos partes: el perfil y el cuerpo
  - Por ejemplo, el perfil de `insertMoney` es:

    ```
    public void insertMoney(int amount)
    ```

- La sintaxis de los perfiles es la siguiente:

    ```
    public tipo nombre(parámetro 1, ..., parámetro n)
    ```

  - Los métodos públicos (`public`) son accesibles desde cualquier clase
    - Podemos declarar métodos privados (`private`) que sólo pueden ejecutarse desde dentro de la clase
  - `tipo` es el tipo de retorno del método (`void` si no retorna un valor)
  - Convención: Los nombres de los métodos comienzan con minúsculas
  - Los parámetros son opcionales, y se declaran con su tipo

    ```
    tipo1 nombre1, tipo2 nombre2, ...
    ```

Cuerpo de un método

- El cuerpo de un método define su implementación
- Está delimitado por { }

```
public class TicketMachine
{
    // The price of a ticket from this machine.
private int price;
// The amount of money entered by a customer so far.
private int balance;
// The total amount of money collected by this machine.
private int total;

/**
 * @pre 'amount' > 0.
 * @post Receives an amount of money from a customer.
 */
public void insertMoney(int amount)
{
    // TODO: Implementar
}
```

Ejemplo: `insertMoney`

```
/**
 * @pre 'amount' > 0.
 * @post Receives an amount of money from a customer.
 */
public void insertMoney(int amount)
{
    balance = balance + amount;
}
```

- Notar que `insertMoney` modifica el estado de la `TicketMachine`, ya sirve para registrar en la máquina el dinero que ingresó el usuario

Ejemplo: `getPrice`

- Aquí tenemos otro método de la clase:

```
/**
 * @post Returns the price of a ticket.
 */
public int getPrice()
{
        // TODO: Implementar
}
```

- Notar que `getPrice` tiene precondición `true` (siempre es posible ejecutar el método) y podemos omitir `@pre` en la especificación

Ejemplo: `getPrice`

```
/**
 * @post Returns the price of a ticket.
 */
public int getPrice()
{
    return price;
}
```

- La palabra reservada `return` se usa para indicar el valor de retorno de un método
  - `return` también devuelve el control al invocante, por lo que siempre tiene que ser la última sentencia del método
- Notar que `getPrice` permite hacer una consulta sobre el estado de la `TicketMachine`
  - En particular, permite consultar la parte del estado que representa el precio del ticket
  - `getPrice` no modifica el estado de la `TicketMachine`

Métodos modificadores vs. observadores

- Los métodos se pueden clasificar en dos categorías:
- Modificadores: sirven para producir un cambio de estado en el objeto al que se aplican (o indirectamente en algún otro objeto)

```
/**
 * @pre 'amount' > 0.
 * @post Receives an amount of money from a customer.
 */
public void insertMoney(int amount)
{
  balance = balance + amount;
}
```

- Observadores: sirven para obtener información sobre el estado de un objeto
  - No modifican el estado del objeto al que se aplican (ni ningún otro)

```
/**
 * @post Returns the price of a ticket.
 */
public int getPrice()
{
    return price;
}
```

Condicionales en Java

- La sintaxis de la sentencia condicional en Java es la siguiente:

```
if(condition) {
   // sentencias if
}
else {
   // sentencias else
}
```

- `condition` es una condición booleana
  - Java provee los operadores booleanos tradicionales: `&&` (and), `||` (or), `!` (not), `==` (equals), `!=` (not equals)
- Podemos omitir la sentencia `else` en caso de que no sea necesaria
- Se pueden omitir las llaves sólo si luego del `if` (o del `else`) hay una única sentencia

Imprimir por salida estándar

- `System.out.println` imprime un `String` por la salida estándar
  - Típicamente, la salida estándar es la terminal
  - Por ejemplo, en `printTicket` usaremos las siguientes sentencias para simular la impresión de un ticket:

```
// Simulate the printing of a ticket.
System.out.println("##################");
System.out.println("# The BlueJ Line");
System.out.println("# Ticket");
System.out.println("# " + price + " cents.");
System.out.println("##################");
System.out.println();
```

- Notar que usamos la concatenación de `String`s implementada por el operador `+`

Ejemplo: `printTicket`

- Implementemos el método más importante de la clase
- Para mantener el ejemplo simple usamos la salida estándar para imprimir el ticket
  - Una alternativa más apropiada sería definir una clase para modelar los tickets
  - Lo veremos más adelante...

```
/**
 * @post If enough money has been inserted,
 * print a ticket to the console and reduce
 * the current balance by the ticket price.
 * Returns 'true' if successful; otherwise,
 * it does nothing and returns 'false'.
 */
public boolean printTicket()
{
    if(// TODO: Implementar.) {
        // Simulate the printing of a ticket.
        System.out.println("##################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("##################");
        System.out.println();

        // Update the total collected with the price.
        // Reduce the balance by the price.
        // TODO: Implementar.
    }
    else {
        // TODO: Implementar.
    }
}
```

Ejemplo: `printTicket`

- Notar que `printTicket` es un método modificador
  - Cambia el estado de la máquina y el de la terminal (imprime un ticket)
- El valor de retorno se usa para informar sobre la posibilidad o no de emitir un ticket
  - Otra opción sería tratar esto como un error y agregarlo a la precondición del método

```
/**
 * @post If enough money has been inserted,
 * print a ticket to the console and reduce
 * the current balance by the ticket price.
 * Returns 'true' if successful; otherwise,
 * it does nothing and returns 'false'.
 */
public boolean printTicket()
{
    if(balance >= price) {
        // Simulate the printing of a ticket.
        System.out.println("####################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("####################");
        System.out.println();

        // Update the total collected with the price.
        total = total + price;
        // Reduce the balance by the price.
        balance = balance - price;
        return true;
    }
    else {
        return false;
    }
}
```

Variables locales

- A veces es necesario almacenar temporalmente algunos datos durante la ejecución de un método
- En estos casos definimos variables locales como `amountToRefund`
- Las variables locales se definen en el cuerpo de un método, y sólo pueden usarse dentro del método

```
/**
 * @post Returns the money in the balance
 * and clears the balance.
 */
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

- Notar que `amountToRefund` no almacena datos que nos interese persistir en las `TicketMachine`s
  - Es por esto que la declaramos como una variable local y no como atributo

Alcance

- El alcance de una variable determina la sección del código en la se puede acceder a ella
- Notar que los atributos, los parámetros formales y las variables locales son distintos tipos de variables

```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }
```

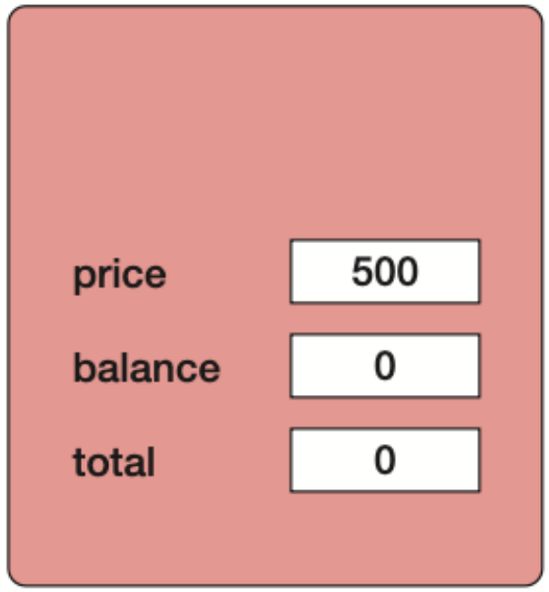
```
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

- El alcance de un atributo privado es la clase a la que pertenece el atributo
  - Pueden accederlo todos los constructores y métodos de la clase
- El alcance de los parámetros formales y de las variables locales se limita al método que los define

Tiempo de vida

- Un concepto relacionado al alcance es el tiempo de vida: cuanto tiempo "vive" una variable en memoria
- El tiempo de vida de un atributo es el mismo que el tiempo de vida del objeto al que pertenece
  - Cuando se crea un objeto se reserva espacio para todos sus atributos
  - Los atributos persisten durante el tiempo de vida del objeto
- El tiempo de vida de un parámetro se limita a una única ejecución de un método
  - Cuando un método se invoca, se reserva espacio de memoria para los parámetros formales, y se copian los parámetros actuales en dicho espacio
  - Se eliminan al terminar la ejecución
- El tiempo de vida de una variable local se limita a una única ejecución de un método

Objeto en memoria:




```
public TicketMachine(int cost)
{
    price = cost;
    balance = 0;
    total = 0;
}
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

Atributos vs. variables locales

- ¿Definimos una variable como atributo o cómo variable local?
- Podemos usar las siguientes pautas como guía:
  - Analizar si tiene sentido que la variable sea parte del estado del objeto
    - Si la respuesta es positiva definirla como atributo
    - Si no estamos seguros, definir una variable local
  - Evitar definir un atributo cuando con una variable local es suficiente

Abstracciones de datos

- En esta teoría, especificamos e implementamos nuestra primera abstracción de datos: `TicketMachine`
- Una abstracción de datos define un conjunto de objetos, y un conjunto de operaciones aplicables a los objetos
  - Abstracción de datos = (objetos, operaciones)
- Notar que una abstracción de datos es un tipo de datos
- Pero nos abstraemos de la representación interna de los objetos
- Requerimos que los clientes usen las operaciones de los objetos, basándose en sus especificaciones (abstracción por especificación)
  - En lugar de modificar directamente la representación interna de los objetos

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 */
public class TicketMachine
{
    /**
     * @pre 'cost' > 0.
     * @post Create a machine that issues tickets
     *   with a price of 'cost'.
     */
    public TicketMachine(int cost)
    /**
     * @post Returns the price of a ticket.
     */
    public int getPrice()
    /**
     * @post Returns the amount of money already inserted
     * next ticket.
     */
    public int getBalance()
    /**
     * @pre 'amount' > 0.
     * @post Receives an amount of money from a customer.
     */
    public void insertMoney(int amount)
    /**
     * @post If enough money has been inserted, print a ti
     * and reduce the current balance by the ticket price.
     * successful; otherwise, it does nothing and returns
     */
    public boolean printTicket()
    /**
     * @post Returns the money in the balance and clears
     * the balance.
     */
    public int refundBalance()
}
```

Abstracciones de datos

- De esta manera, los programas que utilizan las abstracciones se vuelven independientes de la representación interna de los objetos
  - Podemos cambiar la implementación sin afectar a las clases que la utilizan
- Diseñar y usar buenas abstracciones de datos resulta en programas bien modularizados
  - Lo que facilita la modificación, la reparación, y la extensión de los programas, y los hace más fáciles de entender
- Las abstracciones de datos son la base de la programación orientada a objetos, y uno de los conceptos centrales de la materia

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 */
public class TicketMachine
{
    /**
     * @pre 'cost' > 0.
     * @post Create a machine that issues tickets
     *   with a price of 'cost'.
     */
    public TicketMachine(int cost)
    /**
     * @post Returns the price of a ticket.
     */
    public int getPrice()
    /**
     * @post Returns the amount of money already inserted
     * next ticket.
     */
    public int getBalance()
    /**
     * @pre 'amount' > 0.
     * @post Receives an amount of money from a customer.
     */
    public void insertMoney(int amount)
    /**
     * @post If enough money has been inserted, print a ti
     * and reduce the current balance by the ticket price.
     * successful; otherwise, it does nothing and returns
     */
    public boolean printTicket()
    /**
     * @post Returns the money in the balance and clears
     * the balance.
     */
    public int refundBalance()
}
```

El método `main`

- El método `main` es el punto de inicio para la ejecución de las aplicaciones Java
- Debe tener el siguiente perfil:

```
public static void main(String[] args)
```

  - El arreglo `args` permite pasar valores como parámetro en la invocación del programa
    - Desde la terminal, o usando gradle

- Usualmente se define una clase aparte que contiene este método, para separar el punto de inicio del resto de las clases de la aplicación

El método `main`

- Como ejemplo, veamos un `main` que crea una `TicketMachine` y realiza algunas operaciones sobre la misma

```
public class TicketMachineMain {

    public static void main(String[] args) {
        TicketMachine machine = new TicketMachine(10);
        machine.insertMoney(10);
        boolean res = machine.printTicket();
        System.out.println("Result: " + res);
        System.out.println("Balance: " + machine.getBalance());
    }

}
```

Actividades

- Leer los capítulos 1 y 2 del libro "*Objects First with Java A Practical Introduction using BlueJ*". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016

- Leer los capítulos 1 y 2 del libro "*Program Development in Java - Abstraction, Specification, and Object-Oriented Design*". B. Liskov & J. Guttag. Addison-Wesley. 2001

Bibliografía

- "*Objects First with Java A Practical Introduction using BlueJ*". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016

- "*Program Development in Java - Abstraction, Specification, and Object-Oriented Design*". B. Liskov & J. Guttag. Addison-Wesley. 2001