ttexto base normal para probar fuentes;

Elementos HTML a Medida

HTML5 Admite crear nuevos elementos a medida del usuario. Vamos a experimentarlo mediante un ejemplo conductor. Crea una página con el siguiente "body" en el que hemos metido un elemento html inventado llamado "mi-elemento"

```
<body>

<mi-elemento img="./imagen.png" data-text="datos"> Qu&eacute; pasa
con este contenido?
</mi-elemento>

</body>
```

Cuando visualices la página en el navegador, observarás que este elemento ha sido procesado como si de un se tratase. El navegador no sólo no ha indicado un error, sino que ha encajado bien eso que hemos puesto ahí sin sentido aparente

Sin embargo, podemos ir más allá y dotar a ese elemento inventado de un comportamiento y aspecto adaptado. Para ello, se incluirá en el código javascript una descripción de cómo debe tratarse ese componente. Así el navegador, cuando procesa el mismo código html anterior procesará el elemento <mi-elemento> de forma concreta, tal como se indica en el código javascript asociado.

Primeros pasos

El primer paso para utilizar componentes html adaptados es registrarlos en el navegador. Esto debe hacerse desde javascript y comporta dos acciones principalmente

1. Crear una clase que hereda de HTMLElement
2. Registrar una etiqueta ("mi-elemento" enel ej. anterior) con la nueva clase.

Registrar un elemento

Usamos la llamada:

```
customElements.define("mi-elemento",MiElemento);
```

Donde

- "mi-elemento" es el nombre que damos al nuevo elemento html (usado arriba)
- "MiElemento" es una clase hecha en javascript que describe ese elemento

El trabajo importante a la hora de crear un elemento es implementar la clase "MiElemento"

Clase relativa al nuevo elemento

El principal requisito es que la clase debe heredar de la clase `HTMLElement` y como tal, debemos tener un constructor y llamar a super() en él. Este es el esqueleto básico a incluir en el código.

```
class MiElemento extends HTMLElement {
  constructor() {
      // Always call super first in constructor
      super();


      // aquí se pone el código que inicializa el objeto
  }
}
```

ShadowRoot

Un shadowroot es un árbol DOM secundario insertado dentro del DOM principal... ¿Lo qué?... Un shadowroot es similar a un minidocumento html propio que está realmente dentro de otro documento mayor, pero siguen siendo independientes. Por ejemplo, un estilo en un árbol no se aplica al otro.

Nuestro elemento va a tener su propio shadowroot, dentro del cual añadiremos componentes html y contenido Esto necesitamos hacerlo porque la página HTML ha sido cargada, procesada, renderizada, mostrada y ... después de todo eso... tenemos que crear NUEVO contenido a mostrar. La solución es

crearse un shadoRoot nuevo, adicional, que insertaremos dentro de la página principal y que hasta cierto punto es independiente.

Crearemos un shadowroot de la siguiente manera:

```
const shadow = this.attachShadow({mode: 'open'});
```

Así, la variable shadow anterior, admite que insertemos en ella nuevos elementos html como si de cualquier otro elemento similar a div o body se tratase

```
shadow.appendChild(elemento);
```

Estos elementos añadidos pueden ser también elementos de estilo (css)

```
const style = document.createElement('style');
...
style.textContent = `
      * {
      border: 1px solid red;

...

shadow.appendChild(style);
```

El siguiente código simplificado marca la manera en la que construiremos nuestro componente

```
const shadow = this.attachShadow({mode: 'open'});
const wrapper = document.createElement('span');
...
wrapper.appendChild(....
const style = document.createElement('style');
...
style.textContent = `
      * {
      border: 1px solid red;

...

shadow.appendChild(style);

shadow.appendChild(wrapper);
```

Especificando atributos

Los elementos creados pueden ser totalmente especificados en atributos como sus clases , identificador, etc con setAttribute

```
    const icon = document.createElement('span');
    icon.setAttribute('class', 'icon');
    icon.setAttribute('id', 'identificado');
    icon.setAttribute('tabindex', 0);
```

En el caso de asignar clases para establecer un estilo puede usarse classList

```
    icon.classList += "azulito grandecito"
```

Si revisas el código html otra vez, verás que hay un campo llamado data-text en el elemento html

```
<mi-elemento img="./imagen.png" data-text="datos"> Qu&eacute; pasa
con este contenido?
</mi-elemento>
```

El valor asignado a ese atributo puede recuperarse de la siguiente manera

```
  const text = this.getAttribute('data-text');
  info.textContent = text;
```

Es posible preguntar si existe un atributo o no:

```
  let imgUrl;
    if(this.hasAttribute('img')) {
    imgUrl = this.getAttribute('img');
    } else {
    imgUrl = 'img/default.png';
    }
```

En el caso anterior estamos dando valor a la url de una imagen. A continuación se crea totalmente la imagen teniendo el valor de su url lista para asignar en el campo src.

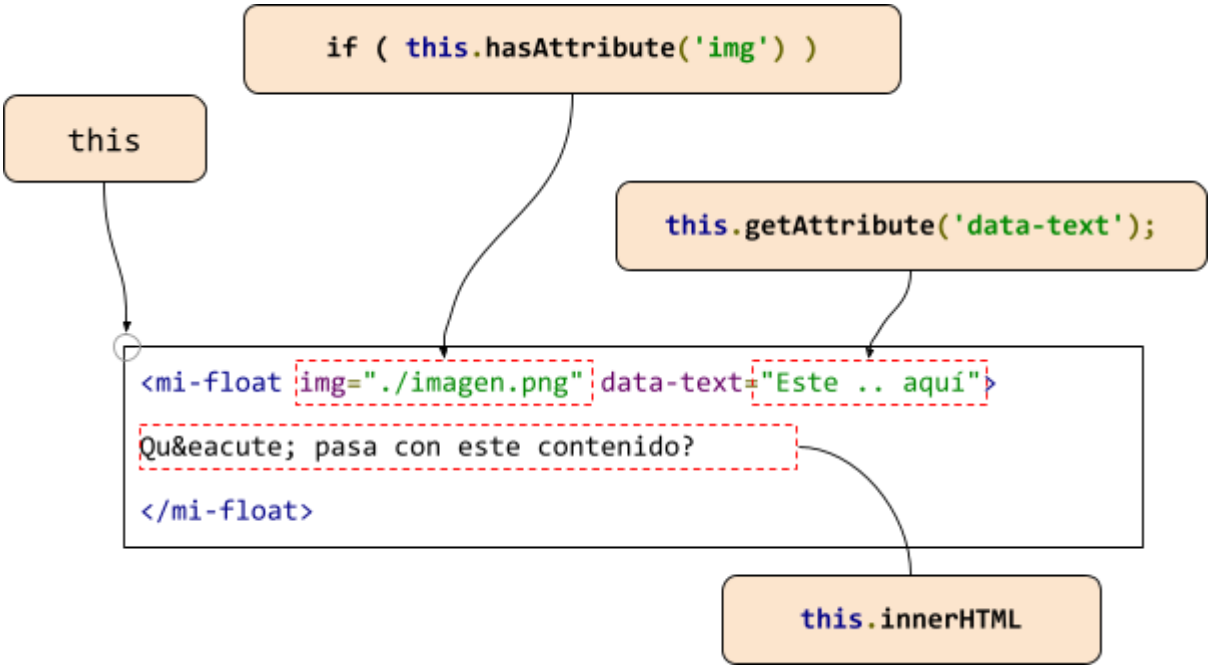
```
const img = document.createElement('img');

    img.src = imgUrl;

    icon.appendChild(img);
```

Los estilos se pueden crear con un elemento de tipo style (primera línea del ejemplo siguiente) y se asigna el código css deseado al campo textContent del elemento . En el ejemplo de a continuación, para mejor legibilidad se ha usado la tilde abierta para delimitar el principio y final del texto css.

```
const style = document.createElement('style');

style.textContent = `
    .wrapper {
        position: relative;
    }
    .info {
        font-size: 0.8rem;
        ...
        opacity: 0;
        ...
    }
    img {
        width: 1.2rem;
    }
    .icon:hover + .info, .icon:focus + .info {
        opacity: 1;
    }
    `;
```

Finalmente el contenido comprendido entre las etiquetas <mi-elemento> y </mi-elemento> se puede acceder (o cambiar) con this.innertHTML;

```
const contenido = document.createElement('div');
contenido.setAttribute('id','identificacion');
contenido.innerHTML = this.innerHTML;
```




```
<mi-float img="./imagen.png" data-text="Este es un texto emergente que aparece aquí"> Qu&eacute; pasa con este contenido?</mi-float>
```

Referenciar atributos y métodos

Una nota importante, a diferencia de una función normal en javascript donde se puede acceder a variables globales o locales libremente. Por ejemplo, en el siguiente código, las variables ctx y posX son globales y son accedidas desde la función "dibujar()":

```
<script type="text/javascript">
      var canvas;
            var ctx;
            var posX;

     function dibujar() {
           console.log("dibujando");
           if (canvas.getContext) {
                ctx.clearRect(0,0,300,300);
                 ctx.fillStyle = 'rgb(200, 0, 0)';
                ctx.fillRect(posX, 10, 30, 80);
```

```
    ...
```

Cuando estamos trabajando en una clase, podemos declararnos igualmente atributos y métodos pertenecientes a la clase. Por ejemplo, supongamos una clase en cuyo principio se declaran unos atributos:

```
class MiFloat extends HTMLElement {
  direccion = "derecha";
  text     = "" ;

  angulo  = 0.0  ;
  iteraciones = 0;

  contenido = "";
  estilo   = "";

  shadow="";
  imagen="";
```

ESta clase también puede declarar métodos (además del constructor que ya conocemos

```
class MiFloat extends HTMLElement {
  direccion = "derecha";
 ...

 crearEstilo(x,y) {
    ...
}

 moverImagen() {
     ...
 }
```

Si estos métodos desean manipular los atributos del objeto declarados arriba, deberán usar el prefijo this obligatoriamente. Si declaramos variables locales a los métodos los usaremos con normalidad. Observa el siguente extracto de ejemplo, donde se accede a atributos de la clase y variables globales.

```
class MiFloat extends HTMLElement {
  direccion = "derecha";
 ...

 crearEstilo(x,y) {
    ...
}

 moverImagen() {

  let geo = this.contenido.getBoundingClientRect();

  let padre = this.contenido.parentElement;
  let geoPadre = padre.getBoundingClientRect();
  let centroX = Math.round( (geo.left + geo.right ) / 2 -
geoPadre.left);
  let centroY = Math.round( (geo.top + geo.bottom ) / 2 -
geoPadre.top);

  let geoImagen = this.imagen.getBoundingClientRect();
  let imgWidth= geoImagen.width;
  centroX -= imgWidth;
  centroY -= imgWidth;

  let radioHorizontal = imgWidth + Math.round( (geo.right -
geo.left )) / 2;
  let radioVertical = imgWidth + Math.round( (geo.bottom - geo.top
))   / 2 ;

  let ang = this.angulo;

  ang = ( ang + 0.1 ) > (2 * Math.PI) ? (0) : ang + 0.1;
  this.angulo = ang; //actualizamos el Ã¡ngulo

 }
```

Igualmente las llamadas a un método deben estar hechas con this

```
class MiFloat extends HTMLElement {
  direccion = "derecha";
 ...

 crearEstilo(x,y) {
```

```
    ...
}

 moverImagen() {

  ...

  const textoEstilo = this.crearEstilo(puntoX,puntoY);


 }
```

Animar un customElement

Si queremos que un elemento ejecute alguna acción periódicamente, se debe "instalar" una llamada **periódica** a algún método de ese objeto. Supón la siguiente clase que contiene algún tipo de animación basada en llamar al método mover() que puedes ver a continuación

```
class MiFloat extends HTMLElement {
 // atributos

 mover() {
     //animar algo
 }

  constructor() {
      super();

} // clase
```

Para que ese método se ejecute, hay que llamar a setInterval y conseguir que cada cierto tiempo se ejecute ese método **sobre ese preciso objeto.** Ojo! con usar this dentro del método mover() porque al llamar a set interval así:

```
class MiFloat extends HTMLElement {
 // atributos
```

```
 mover() {
     //animar este objeto
     this.esto
     this.loOTRO.
 }

  constructor() {
      super();

      ...

       setInterval(mover, 100);

   }// constructor

} // clase
```

Estamos haciendo que se ejecute ese método pero "this" cuando se ejecuta **no es el objeto** que esperamos que sea, sino que **this** ahí es **window.** Lo que hemos de hacer es lograr que se ejecute el método **de ese objeto**. Así lo conseguimos:

```
class MiFloat extends HTMLElement {
 // atributos

 mover() {
     //animar este objeto
     this.esto
     this.loOTRO.
 }

  constructor() {
      super();

      ...


     setInterval( function(elemento) {
       return function(e) {
              elemento.moverImagen(); }
       } (this), 100);
```

```
   }// constructor

} // clase
```

Derivando elementos

Siguiendo con la idea de crear nuestros propios elementos, tenemos una alternativa a crear un nuevo elemento enteramente. Podemos derivar (modificar) uno ya existente.

Tomemos por ejemplo el siguiente código HTML que usaremos de base.

```
   <p is="word-count"></p>
```

Fíjate en el último elemento <p> y concretamente en el atributo "is=" (`is="word-count`). Este atributo es especial en que indica que se trata de un elemento <p> de un cierto tipo (de tipo "word-count"). No es cualquier <p>.

(no debemos confundir esto con el atributo "class=" que simplemente indica que tiene un estilo concreto)

Al igual que se ha hecho antes, en Javascript habremos indicado que deseamos registrar este tipo de elementos "word-cound" en customelements.

En la inicialización de la página (habitualmente):

```
customElements.define('word-count',WordCount,{ extends: 'p' });
```

Arriba estamos ligando los elementos etiquetados como "word-count" a la clase "WordCount" , pero ¡ Atención ! , a diferencia de antes, ahora sólo estamos considerando extender (ampliar, personalizar, modificar) los elementos p con esta etiqueta "word-count"

Por lo demás, nos queda el mismo trabajo que antes: Definir la clase, aunque con una pequeña e importante diferencia: Ahora heredamos o extendemos HTMLParagraphElement (que es la clase correspondiente a <p>=

```
class WordCount extends HTMLParagraphElement {


```

Recuerda que antes , para crear un totalmente nuevo elemento html

```
class MiElemento extends HTMLElement {


```

Debemos completar el constructor y allí hacer todo lo necesario, especialmente crear un nuevo shadowroot donde ubicar nuevos elmentos

```
class WordCount extends HTMLParagraphElement {
  constructor() {
      // Always call super first in constructor
      super();

      // Create a shadow root
      const shadow = this.attachShadow({mode: 'open'});

      // Create text node and add word count to it
      const text = document.createElement('span');
      text.textContent = ...

      // Append it to the shadow root
      shadow.appendChild(text);

  }
}
```

Explorar el DOM

Desde un elemento customelement creado por nosotros, es perfectamente posible explorar y modificar el árbol DOM del documento principal, o el elemento padre donde insertamos este elemento.

Toma por elemplo el siguiente ćodigo. Fíjate que nuestro elemento <p is="word-count"> está dentro de un <article> donde hay más elementos. Es decir, nuestro <p> tiene un padre y también tiene hermanos

```
<article contenteditable="">
   <p>Lorem ipsum dolor sit ... sit amet.                </p>

   <p>Pellentes... luctus dignissim metus.sed diam.     </p>

   <p is="word-count"></p>
</article>
```

La cuestión aquí es ¿Cómo podríamos desde la clase WordCount acceder a los hermanos de <p> o al contenido de <article> ?

La respuesta no tiene mucho misterio, es igual que si estuviésemos fuera de una clase concreta, ejecutando un recorrido del DOM, mediante el atributo "parentNode" sobre un elemento. Sólo que ahora este elemento es "this"

```
class WordCount extends HTMLParagraphElement {
  constructor() {
      // Always call super first in constructor
      super();
      const wcParent = this.parentNode;

      // PODEMOS EXPLORAR wcParent

      // Create a shadow root
      const shadow = this.attachShadow({mode: 'open'});

      // Create text node and add word count to it
      const text = document.createElement('span');
      text.textContent = count;

      // Append it to the shadow root
      shadow.appendChild(text);

  }
}
```

Templates

Antes de ir a por él, métete esto en la cabeza: Un template se hace así:

```
<template id="filaProducto">
  <ELEM>
      <...
  </ELEM>
</template>
```

Donde ELEM es cualquier elemento HTML habitual, por ejemplo un div con contenido interior:

```
<template id="filaProducto">
  <div>
      <p class=">...</p>
      <img src="... ">
  </div>
</template>
```

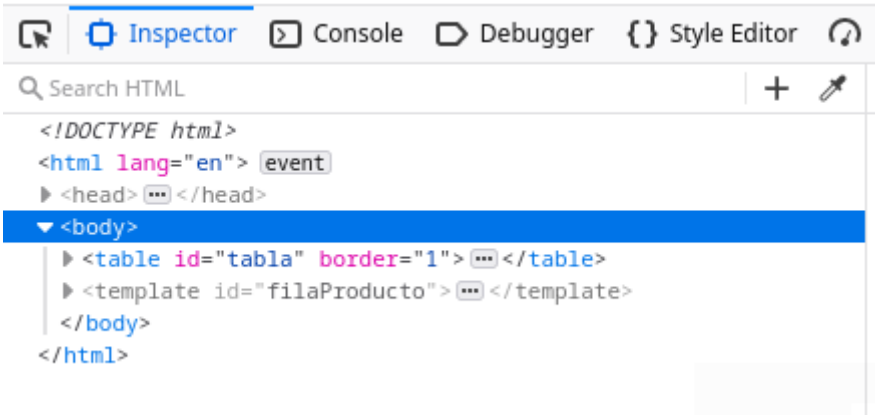
Si incluyes un template en el body de una página html no lo verás en el navegador. No se renderiza, no aparece. Si lo observas con el inspector del firefox, éste aparece en gris, indicando que no es un elemento visible en el viewport.

Por ejemplo. Del siguiente código html

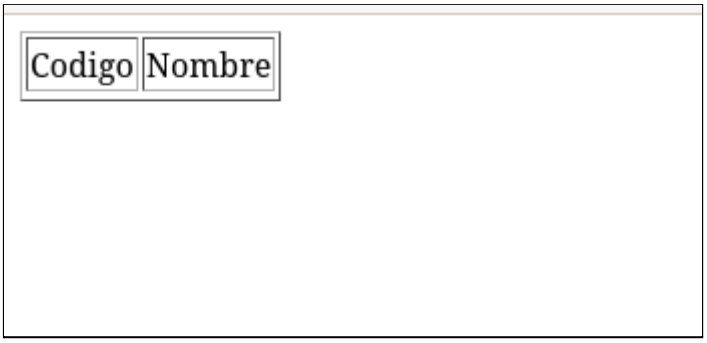
```
  <table id="tabla" border="1">
  <thead >
      <tr>
      <td>Codigo</td>
      <td>Nombre</td>
      </tr>
  </thead>
  <tbody>

  </tbody>
</table>

<template id="filaProducto">
  <tr>
      <td class="record"></td>
      <td></td>
  </tr>
```

```
</template>
```

Lo que el navegador muestra en el inspector es:



¿Ves cóm el template aparece en gris? Esto es porque no se vé nada. Mira la salida de la página



El template es reconocido por el navegador y procesado. El navegador crea un objeto para él pero no muestra nada.

Este objeto puede ser referenciado desde javascript y se le puede insertar contenido o maninpular (pese a que no es visible). En este ejemplo. El template es un elemento <tr> con dos <td> en su interior y por tanto podemos darle valor a cada elemento td de la siguiente forma:

```
var elem = document.getElementById("filaProducto");
var fila = elem.content.querySelectorAll("td");
fila[0].textContent = "0AS-32";
```

```
        fila[1].textContent = "spray Anti Mosquitos";
```

Por mucho que manipulemos el objeto, éste continúa sin aparecer, pero ahora viene lo bueno. Podemos hacernos una copia, un clon de ese objeto

```
        var elem = document.getElementById("filaProducto");
        var fila = elem.content.querySelectorAll("td");
        fila[0].textContent = "0AS-32";
        fila[1].textContent = "spray Anti Mosquitos";

        var clon  = document.importNode(elem.content,true);
```

Y ahora, podemos insertar ese elemento en otro elemento visible ya en el DOM. con ello estamos añadiendo contenido a la página, que hemos pre-fabricado con el template.

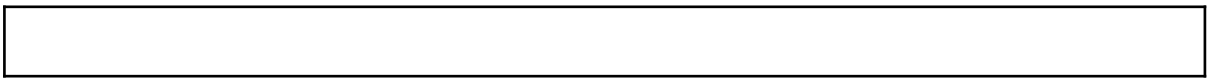
```
      var elem = document.getElementById("filaProducto");
       var fila = elem.content.querySelectorAll("td");
       fila[0].textContent = "0AS-32";
       fila[1].textContent = "spray Anti Mosquitos";

       var clon  = document.importNode(elem.content,true);

       var tabla = document.getElementById("tabla");
       var bodyTabla = tabla.querySelector("tbody");

       bodyTabla.appendChild(clon);
```

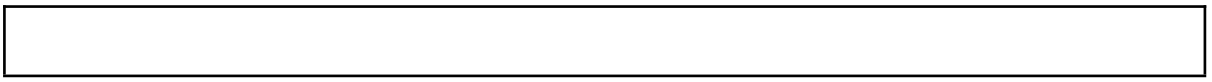
Lo bueno, es que el template no ha sido mostrado, sino una copia suya. Es posible volver a modificar a continuación el template, hacer otra copia e insertarla.

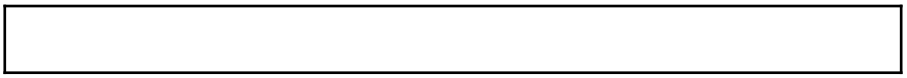
La idea es que podemos crear contenido a partir del template, facilitando una tarea que si tuviésemos que hacerla desde javascript puro con document.createElement(), document,appendChild, document.textContent=, etc sería muy tediosa.

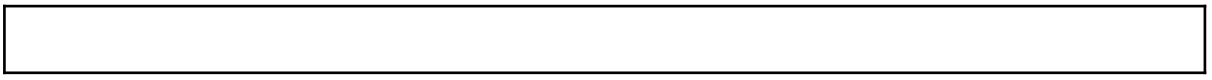
Ciclo de vida

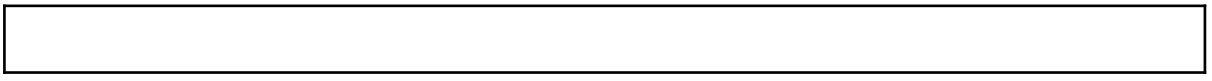
Un elemento html tiene un ciclo de vida,

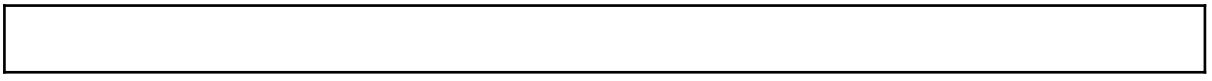


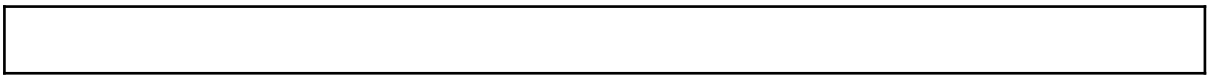


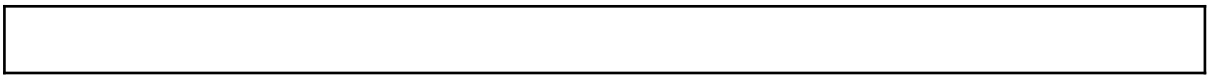


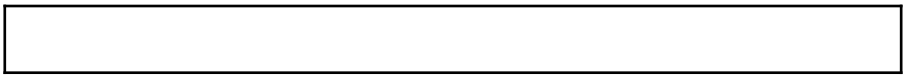


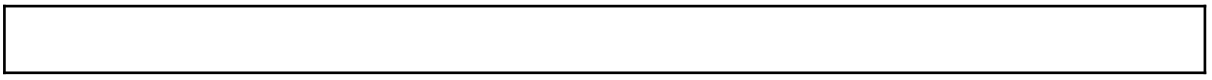


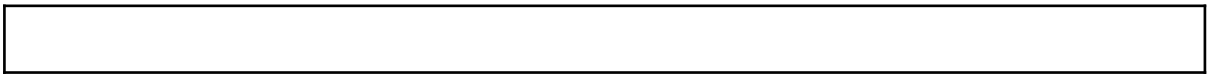


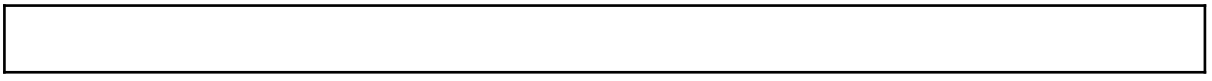


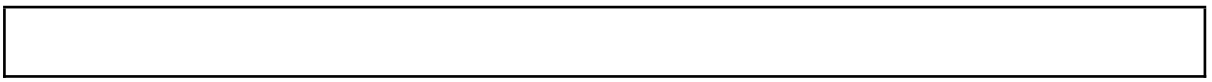












v