Intro

In order to provide quality support for touch-based user interfaces, touch events offer the ability to interpret finger (or stylus) activity on touch screens or trackpads.
The touch events interfaces are relatively low-level APIs that can be used to support application specific multi-touch interactions such as a two-finger gesture. A multi-touch interaction starts when a finger (or stylus) first touches the contact surface. Other fingers may subsequently touch the surface and optionally move across the touch surface. The interaction ends when the fingers are removed from the surface. During this interaction, an application receives touch events during the start, move and end phases.
Touch events are similar to mouse events except they support simultaneous touches and at different locations on the touch surface. The TouchEvent interface encapsulates all of the touch points that are currently active. The Touch interface, which represents a single touch point, includes information such as the position of the touch point relative to the browser viewport.

Definitions

Surface
The touch-sensitive surface. This may be a screen or trackpad.
Touch point
A point of contact with the surface. This may be a finger (or elbow, ear, nose, whatever, but typically a finger) or stylus.

Interfaces

TouchEvent
Represents an event that occurs when the state of touches on the surface changes.
Touch
Represents a single point of contact between the user and the touch surface.
TouchList
Represents a group of touches; this is used when the user has, for example, multiple fingers on the surface at the same time.

Example

This example tracks multiple touch points at a time, allowing the user to draw in a <canvas> with more than one finger at a time. It will only work on a browser that supports touch events.
Note: The text below uses the term "finger" when describing the contact with the surface, but it could, of course, also be a stylus or other contact method.

Create a canvas

```
<canvas id="canvas" width="600" height="600" style="border:solid black 1px;">
  Your browser does not support canvas element.
</canvas>
<br>
<button onclick="startup()">Initialize</button>
<br>
Log: <pre id="log" style="border: 1px solid #ccc;"></pre>
```

Setting up the event handlers

When the page loads, the startup() function shown below should be called by our <body> element's onload attribute (but in the example we use a button to trigger it, due to limitations of the MDN live example system).

```
function startup() {
  var el = document.getElementsByTagName("canvas")[0];
  el.addEventListener("touchstart", handleStart, false);
  el.addEventListener("touchend", handleEnd, false);
  el.addEventListener("touchcancel", handleCancel, false);
  el.addEventListener("touchmove", handleMove, false);
  log("initialized.");
}
```

This simply sets up all the event listeners for our <canvas> element so we can handle the touch events as they occur.

Tracking new touches

We'll keep track of the touches in-progress.

```
var ongoingTouches = [];
```

When a touchstart event occurs, indicating that a new touch on the surface has occurred, the handleStart() function below is called.

```
function handleStart(evt) {
  evt.preventDefault();
  log("touchstart.");
  var el = document.getElementsByTagName("canvas")[0];
  var ctx = el.getContext("2d");
  var touches = evt.changedTouches;

  for (var i = 0; i < touches.length; i++) {
    log("touchstart:" + i + "...");
    ongoingTouches.push(copyTouch(touches[i]));
    var color = colorForTouch(touches[i]);
    ctx.beginPath();
    ctx.arc(touches[i].pageX, touches[i].pageY, 4, 0, 2 * Math.PI, false);  // a
circle at the start
    ctx.fillStyle = color;
    ctx.fill();
    log("touchstart:" + i + ".");
  }
}
```
This calls event.preventDefault() to keep the browser from continuing to process the touch event (this also prevents a mouse event from also being delivered). Then we get the context and pull the list of changed touch points out of the event's TouchEvent.changedTouches property.
After that, we iterate over all the Touch objects in the list, pushing them onto an array of active touch points and drawing the start point for the draw as a small circle; we're using a 4-pixel wide line, so a 4 pixel radius circle will show up neatly.

Drawing as the touches move

Each time one or more fingers moves, a touchmove event is delivered, resulting in our handleMove() function being called. Its responsibility in this example is to update the cached touch information and to draw a line from the previous position to the current position of each touch.
```
function handleMove(evt) {
  evt.preventDefault();
  var el = document.getElementsByTagName("canvas")[0];
  var ctx = el.getContext("2d");
  var touches = evt.changedTouches;

  for (var i = 0; i < touches.length; i++) {
    var color = colorForTouch(touches[i]);
    var idx = ongoingTouchIndexById(touches[i].identifier);

    if (idx >= 0) {
      log("continuing touch "+idx);
      ctx.beginPath();
      log("ctx.moveTo(" + ongoingTouches[idx].pageX + ", " + ongoingTou-
ches[idx].pageY + ");");
      ctx.moveTo(ongoingTouches[idx].pageX, ongoingTouches[idx].pageY);
      log("ctx.lineTo(" + touches[i].pageX + ", " + touches[i].pageY + ");");
      ctx.lineTo(touches[i].pageX, touches[i].pageY);
      ctx.lineWidth = 4;
      ctx.strokeStyle = color;
      ctx.stroke();

      ongoingTouches.splice(idx, 1, copyTouch(touches[i]));  // swap in the new
touch record
      log(".");
    } else {
      log("can't figure out which touch to continue");
    }
  }
}
```
This iterates over the changed touches as well, but it looks in our cached touch information array for the previous information about each touch in order to determine the starting point for each touch's new line segment to be drawn. This is done by looking at each touch's Touch.identifier property. This property is a unique integer for each touch, and remains consistent for each event during the duration of each finger's contact with the surface.
This lets us get the coordinates of the previous position of each touch and use the appropriate context methods to draw a line segment joining the two positions together.
After drawing the line, we call Array.splice() to replace the previous information about the touch point with the current information in the ongoingTouches array.

Handling the end of a touch

When the user lifts a finger off the surface, a touchend event is sent. We handle both of these the same way: by calling the handleEnd() function below. Its job is to draw the last line segment for each touch that ended and remove the touch point from the ongoing touch list.

```
function handleEnd(evt) {
  evt.preventDefault();
  log("touchend");
  var el = document.getElementsByTagName("canvas")[0];
  var ctx = el.getContext("2d");
  var touches = evt.changedTouches;

  for (var i = 0; i < touches.length; i++) {
    var color = colorForTouch(touches[i]);
    var idx = ongoingTouchIndexById(touches[i].identifier);

    if (idx >= 0) {
      ctx.lineWidth = 4;
      ctx.fillStyle = color;
      ctx.beginPath();
      ctx.moveTo(ongoingTouches[idx].pageX, ongoingTouches[idx].pageY);
      ctx.lineTo(touches[i].pageX, touches[i].pageY);
      ctx.fillRect(touches[i].pageX - 4, touches[i].pageY - 4, 8, 8);  // and a
square at the end
      ongoingTouches.splice(idx, 1);  // remove it; we're done
    } else {
      log("can't figure out which touch to end");
    }
  }
}
```
This is very similar to the previous function; the only real differences are that we draw a small square to mark the end and that when we call Array.splice(), we simply remove the old entry from the ongoing touch list, without adding in the updated information. The result is that we stop tracking that touch point.

Handling canceled touches

If the user's finger wanders into browser UI, or the touch otherwise needs to be canceled, the touchcancel event is sent, and we call the handleCancel() function below.
```
function handleCancel(evt) {
  evt.preventDefault();
  log("touchcancel.");
  var touches = evt.changedTouches;

  for (var i = 0; i < touches.length; i++) {
    ongoingTouches.splice(i, 1);  // remove it; we're done
  }
}
```
Since the idea is to immediately abort the touch, we simply remove it from the ongoing touch list without drawing a final line segment.

Convenience functions

This example uses two convenience functions that should be looked at briefly to help make the rest of the code more clear.

Selecting a color for each touch

In order to make each touch's drawing look different, the colorForTouch() function is used to pick a color based on the touch's unique identifier. This identifier is an opaque number, but we can at least rely on it differing between the currently-active touches.
```
function colorForTouch(touch) {
  var r = touch.identifier % 16;
  var g = Math.floor(touch.identifier / 3) % 16;
  var b = Math.floor(touch.identifier / 7) % 16;
  r = r.toString(16); // make it a hex digit
  g = g.toString(16); // make it a hex digit
  b = b.toString(16); // make it a hex digit
  var color = "#" + r + g + b;
  log("color for touch with identifier " + touch.identifier + " = " + color);
  return color;
}
```
The result from this function is a string that can be used when calling <canvas> functions to set drawing colors. For example, for a Touch.identifier value of 10, the resulting string is "#aaa".

Copying a touch object

Some browsers (mobile Safari, for one) re-use touch objects between events, so it's best to copy the bits you care about, rather than referencing the entire object.
```
function copyTouch(touch) {
  return { identifier: touch.identifier, pageX: touch.pageX, pageY: touch.pageY
};
}
```

Finding an ongoing touch

The ongoingTouchIndexById() function below scans through the ongoingTouches array to find the touch matching the given identifier, then returns that touch's index into the array.
```
function ongoingTouchIndexById(idToFind) {
  for (var i = 0; i < ongoingTouches.length; i++) {
    var id = ongoingTouches[i].identifier;

    if (id == idToFind) {
      return i;
    }
  }
  return -1;    // not found
}
```

Showing what's going on

```
function log(msg) {
  var p = document.getElementById('log');
  p.innerHTML = msg + "\n" + p.innerHTML;
}
```
If your browser supports it, you can see it live.
jsFiddle example

Additional tips

This section provides additional tips on how to handle touch events in your web application.

Handling clicks

Since calling preventDefault() on a touchstart or the first touchmove event of a series prevents the corresponding mouse events from firing, it's common to call preventDefault() on touchmove rather than touchstart. That way, mouse events can still fire and things like links will continue to work. Alternatively, some frameworks have taken to refiring touch events as mouse events for this same purpose. (This example is oversimplified and may result in strange behavior. It is only intended as a guide.)
```
function onTouch(evt) {
  evt.preventDefault();
  if (evt.touches.length > 1 || (evt.type == "touchend" && evt.touches.length >
0))
    return;

  var newEvt = document.createEvent("MouseEvents");
  var type = null;
  var touch = null;

  switch (evt.type) {
    case "touchstart":
      type = "mousedown";
      touch = evt.changedTouches[0];
      break;
    case "touchmove":
      type = "mousemove";
      touch = evt.changedTouches[0];
      break;
    case "touchend":
      type = "mouseup";
      touch = evt.changedTouches[0];
      break;
  }

  newEvt.initMouseEvent(type, true, true, evt.originalTarget.ownerDocument.de-
faultView, 0,
    touch.screenX, touch.screenY, touch.clientX, touch.clientY,
    evt.ctrlKey, evt.altKey, evt.shiftKey, evt.metaKey, 0, null);
  evt.originalTarget.dispatchEvent(newEvt);
}
```

Calling preventDefault() only on a second touch

One technique for preventing things like pinchZoom on a page is to call preventDefault() on the second touch in a series. This behavior is not well defined in the touch events spec, and results in different behavior for different browsers (i.e., iOS will prevent zooming but still allow panning with both fingers; Android will allow zooming but not panning; Opera and Firefox currently prevent all panning and zooming.) Currently, it's not recommended to depend on any particular behavior in this case, but rather to depend on meta viewport to prevent zooming.

Specifications

Specification	Status	Comment
Touch Events – Level 2 The definition of 'Touch' in that specification.	Editor's Draft	Added radiusX, radiusY, rotationAngle, force properties
Touch Events The definition of 'Touch' in that specification.	Recommendation	Initial definition.

Browser compatibility

Desktop
Mobile

Note that unfortunately touch events may not fire at all on laptops with touch functionality (test page).
[1] The dom.w3c_touch_events.enabled tri-state preference can be used to disable (0), enable (1), and auto-detect (2) support for standard touch events; by default, they're on auto-detect (2).
Prior to Gecko 12 (Firefox 12.0 / Thunderbird 12.0 / SeaMonkey 2.9), Gecko did not support multi-touch; only one touch at a time was reported.
As of Gecko 24.0 (Firefox 24.0 / Thunderbird 24.0 / SeaMonkey 2.21), the touch events support introduced with Gecko 18.0 (Firefox 18.0 / Thunderbird 18.0 / SeaMonkey 2.15) has been disabled on the desktop version of Firefox (bug 888304), as some popular sites including Google and Twitter were not working properly. Once the bug is fixed, the API will be enabled again. To enable it anyway, open about:config and set the dom.w3c_touch_events.enabled preference to 2. The mobile versions including Firefox for Android and Firefox OS are not affected by this change. Also, the API has been enabled on the Metro-style version of Firefox for Windows 8.
Prior to Gecko 6.0 (Firefox 6.0 / Thunderbird 6.0 / SeaMonkey 2.3), Gecko offered a proprietary touch event API. That API is now deprecated; you should switch to this one.

Example

=======

This example demonstrates using the touchstart, touchmove, touchcancel, and touchend) touch events for the following gestures: single touch, two (simultaneous) touches, more than two simultaneous touches, 1-finger swipe, and 2-finger move/pinch/swipe.

Define touch targets

The application uses <div> for four touch areas.

```
<style>
  div {
    margin: 0em;
    padding: 2em;
  }
  #target1 {
    background: white;
    border: 1px solid black;
  }
  #target2 {
    background: white;
    border: 1px solid black;
  }
  #target3 {
    background: white;
    border: 1px solid black;
  }
  #target4 {
    background: white;
    border: 1px solid black;
  }
</style>
```

Global state

tpCache is used to cache touch points for processing outside of the event where they were fired.

```
// Log events flag
var logEvents = false;

// Touch Point cache
var tpCache = new Array();
```

Register event handlers

Event handlers are registered for all four touch event types. The touchend and touchcancel event types use the same handler.

```
function set_handlers(name) {
 // Install event handlers for the given element
 var el=document.getElementById(name);
 el.ontouchstart = start_handler;
 el.ontouchmove = move_handler;
 // Use same handler for touchcancel and touchend
 el.ontouchcancel = end_handler;
 el.ontouchend = end_handler;
}

function init() {
 set_handlers("target1");
 set_handlers("target2");
 set_handlers("target3");
 set_handlers("target4");
}
```

Move/Pinch/Zoom handler

This function provides very basic support for 2-touch horizontal move/pinch/zoom handling. The code does not include error handling, vertical moving. Note that the *threshold* for pinch and zoom movement detection is application specific (and device dependent).

```
// This is a very basic 2-touch move/pinch/zoom handler that does not include
// error handling, only handles horizontal moves, etc.
function handle_pinch_zoom(ev) {

 if (ev.targetTouches.length == 2 && ev.changedTouches.length == 2) {
   // Check if the two target touches are the same ones that started
   // the 2-touch
   var point1=-1, point2=-1;
   for (var i=0; i < tpCache.length; i++) {
     if (tpCache[i].identifier == ev.targetTouches[0].identifier) point1 = i;
     if (tpCache[i].identifier == ev.targetTouches[1].identifier) point2 = i;
   }
   if (point1 >=0 && point2 >= 0) {
     // Calculate the difference between the start and move coordinates
     var diff1 = Math.abs(tpCache[point1].clientX - ev.targetTou-
ches[0].clientX);
     var diff2 = Math.abs(tpCache[point2].clientX - ev.targetTou-
ches[1].clientX);

     // This threshold is device dependent as well as application specific
     var PINCH_THRESHHOLD = ev.target.clientWidth / 10;
     if (diff1 >= PINCH_THRESHHOLD && diff2 >= PINCH_THRESHHOLD)
         ev.target.style.background = "green";
   }
   else {
     // empty tpCache
     tpCache = new Array();
   }
 }
}
```

Touch start handler

The touchstart event handler caches touch points to support 2-touch gestures. It also calls preventDefault() to keep the browser from applying further event handling (for example, mouse event emulation).

```
function start_handler(ev) {
 // If the user makes simultaneious touches, the browser will fire a
 // separate touchstart event for each touch point. Thus if there are
 // three simultaneous touches, the first touchstart event will have
 // targetTouches length of one, the second event will have a length
 // of two, and so on.
 ev.preventDefault();
 // Cache the touch points for later processing of 2-touch pinch/zoom
 if (ev.targetTouches.length == 2) {
   for (var i=0; i < ev.targetTouches.length; i++) {
     tpCache.push(ev.targetTouches[i]);
   }
 }
 if (logEvents) log("touchStart", ev, true);
 update_background(ev);
}
```

Touch move handler

The touchmove handler calls preventDefault() for the same reason mentioned above, and invokes the pinch/zoom handler.

```
function move_handler(ev) {
 // Note: if the user makes more than one "simultaneous" touches, most browsers
 // fire at least one touchmove event and some will fire several touchmoves.
 // Consequently, an application might want to "ignore" some touchmoves.
 //
 // This function sets the target element's border to "dashed" to visually
 // indicate the target received a move event.
 //
 ev.preventDefault();
 if (logEvents) log("touchMove", ev, false);
 // To avoid too much color flashing many touchmove events are started,
 // don't update the background if two touch points are active
 if (!(ev.touches.length == 2 && ev.targetTouches.length == 2))
   update_background(ev);

 // Set the target element's border to dashed to give a clear visual
 // indication the element received a move event.
 ev.target.style.border = "dashed";

 // Check this event for 2-touch Move/Pinch/Zoom gesture
 handle_pinch_zoom(ev);
}
```

Touch end handler

The touchend handler restores target's background color back to its original color.

```
function end_handler(ev) {
  ev.preventDefault();
  if (logEvents) log(ev.type, ev, false);
  if (ev.targetTouches.length == 0) {
    // Restore background and border to original values
    ev.target.style.background = "white";
    ev.target.style.border = "1px solid black";
  }
}
```

Application UI

The application uses <div> elements for the touch areas and provides buttons to enable logging and to clear the log.

```
<div id="target1"> Tap, Hold or Swipe me 1</div>
<div id="target2"> Tap, Hold or Swipe me 2</div>
<div id="target3"> Tap, Hold or Swipe me 3</div>
<div id="target4"> Tap, Hold or Swipe me 4</div>

<!-- UI for logging/bebugging -->
<button id="log" onclick="enableLog(event);">Start/Stop event logging</button>
<button id="clearlog" onclick="clearLog(event);">Clear the log</button>
<p></p>
<output></output>
```

Miscellaneous functions

These functions support the application but aren't directly involved with the event flow.

Update background color

The background color of the touch areas will change as follows: no touch is white; one touch is yellow; two simultaneous touches is ping, and three or more simultaneous touches is lightblue. See touch move for information about the background color changing when a 2-finger move/pinch/zoom is detected.

```
function update_background(ev) {
 // Change background color based on the number simultaneous touches
 // in the event's targetTouches list:
 //   yellow - one tap (or hold)
 //   pink - two taps
 //   lightblue - more than two taps
 switch (ev.targetTouches.length) {
   case 1:
     // Single tap`
```

```
      ev.target.style.background = "yellow";
      break;
    case 2:
      // Two simultaneous touches
      ev.target.style.background = "pink";
      break;
    default:
      // More than two simultaneous touches
      ev.target.style.background = "lightblue";
 }
}
```

Event logging

The functions are used to log event activity to the application window, to support debugging and learning about the event flow.

```
function enableLog(ev) {
  logEvents = logEvents ? false : true;
}

function log(name, ev, printTargetIds) {
  var o = document.getElementsByTagName('output')[0];
  var s = name + ": touches = " + ev.touches.length +
                " ; targetTouches = " + ev.targetTouches.length +
                " ; changedTouches = " + ev.changedTouches.length;
  o.innerHTML += s + "
";

  if (printTargetIds) {
    s = "";
    for (var i=0; i < ev.targetTouches.length; i++) {
      s += "... id = " + ev.targetTouches[i].identifier + "
";
    }
    o.innerHTML += s;
  }
}

function clearLog(event) {
 var o = document.getElementsByTagName('output')[0];
 o.innerHTML = "";
}
```

Supporting both TouchEvent and MouseEvent

======================================

The touch interfaces enable applications to create enhanced user experiences on touch enabled devices. However, the reality is the vast majority of today's web content is designed only to work with mouse input. Consequently, even if a browser supports touch, the browser must still *emulate* mouse events so content that assumes mouse-only input will work *as is* without direct modification.
Ideally, a touch-based application does not need to explicitly address mouse input. However, because the browser must emulate mouse events, there may be some interaction issues that need to be handled. Below are some details about the interaction and the ramifications for application developers.

Event firing

The touch events standard defines a few browser requirements regarding touch and mouse interaction (see the *Interaction with Mouse Events and click* section for details), noting *the browser may fire both touch events and mouse events in response to the same user input*. This section describes the requirement that may affect an application.
If the browser fires both touch and mouse events because of a single user input, the browser *must* fire a touchstart before any mouse events. Consequently, if an application does not want mouse events fired on a specific touch target element, the element's touch event handlers should call preventDefault() and no additional mouse events will be dispatched.
Here is a code snippet of the touchmove event handler calling preventDefault().

```
// touchmove handler
function process_touchmove(ev) {
  // Call preventDefault() to prevent any further handling
  ev.preventDefault();
}
```

Event order

Although the specific ordering of touch and mouse events is implementation-defined, the standard indicates the following order is *typical:* for single input:

- touchstart
- Zero or more touchmove events, depending on movement of the finger(s)
- touchend
- mousemove
- mousedown
- mouseup
- click

If the touchstart, touchmove or touchend event is canceled during an interaction, no mouse or click events will be fired, and the resulting sequence of events would just be:

- touchstart
- Zero or more touchmove events, depending on movement of the finger(s)
- touchend

Using Touch Events

================

Today, most Web content is designed for keyboard and mouse input. However, devices with touch screens (especially portable devices) are mainstream and Web applications can either directly process touch-based input by using Touch Events or the application can use *interpreted mouse events* for the application input. A disadvantage to using mouse events is that they do not support concurrent user input, whereas touch events support multiple simultaneous inputs (possibly at different locations on the touch surface), thus enhancing user experiences.
The touch events interfaces support application specific single and multi-touch interactions such as a two-finger gesture. A multi-touch interaction starts when a finger (or stylus) first touches the contact surface. Other fingers may subsequently touch the surface and optionally move across the touch surface. The interaction ends when the fingers are removed from the surface. During this interaction, an application receives touch events during the start, move, and end phases. The application may apply its own semantics to the touch inputs.

Interfaces

Touch events consist of three interfaces (Touch, TouchEvent and TouchList) and the following event types:

- touchstart - fired when a touch point is placed on the touch surface.
- touchmove - fired when a touch point is moved along the touch surface.
- touchend - fired when a touch point is removed from the touch surface.
- touchcancel - fired when a touch point has been disrupted in an implementation-specific manner (for example, too many touch points are created).

The Touch interface represents a single contact point on a touch-sensitive device. The contact point is typically referred to as a *touch point* or just a *touch*. A touch is usually generated by a finger or stylus on a touchscreen, pen or trackpad. A touch point's properties include a unique identifier, the touch point's target element as well as the *X* and *Y* coordinates of the touch point's position relative to the viewport, page, and screen.

The TouchList interface represents a *list* of contact points with a touch surface, one touch point per contact. Thus, if the user activated the touch surface with one finger, the list would contain one item, and if the user touched the surface with three fingers, the list length would be three.

The TouchEvent interface represents an event sent when the state of contacts with a touch-sensitive surface changes. The state changes are starting contact with a touch surface, moving a touch point while maintaining contact with the surface, releasing a touch point and canceling a touch event. This interface's attributes include the state of several *modifier keys* (for example the shift key) and the following touch lists:

- touches - a list of all of the touch points currently on the screen.
- targetTouches - a list of the touch points on the *target* DOM element.
- changedTouches - a list of the touch points whose items depends on the associated event type:
- For the touchstart event, it is a list of the touch points that became active with the current event.
- For the touchmove event, it is a list of the touch points that have changed since the last event.
- For the touchend event, it is a list of the touch points that have been removed from the surface (that is, the set of touch points corresponding to fingers no longer touching the surface).

Together, these interfaces define a relatively low-level set of features, yet they support many kinds of touch-based interaction, including the familiar multi-touch gestures such as multi-finger swipe, rotation, pinch and zoom.

From interfaces to gestures

An application may consider different factors when defining the semantics of a gesture. For instance, the distance a touch point traveled from its starting location to its location when the touch ended. Another potential factor is time; for example, the time elapsed between the touch's start and the touch's end, or the time lapse between two *simultaneous* taps intended to create a double-tap gesture. The directionality of a swipe (for example left to right, right to left, etc.) is another factor to consider.

The touch list(s) an application uses depends on the semantics of the application's *gestures*. For example, if an application supports a single touch (tap) on one element, it would use the targetTouches list in the touchstart event handler to process the touch point in an application-specific manner. If an application supports two-finger swipe for any two touch points, it will use the changedTouches list in the touchmove event handler to determine if two touch points had moved and then implement the semantics of that gesture in an application-specific manner.

Browsers typically dispatch *emulated* mouse and click events when there is only a single active touch point. Multi-touch interactions involving two or more active touch points will usually only generate touch events. To prevent the emulated mouse events from being sent, use the preventDefault() method in the touch event handlers. For more information about the interaction between mouse and touch events, see Supporting both TouchEvent and MouseEvent.

Basic steps

This section contains a basic usage of using the above interfaces. See the Touch Events Overview for a more detailed example.

Register an event handler for each touch event type.

```
// Register touch event handlers
someElement.addEventListener('touchstart', process_touchstart, false);
someElement.addEventListener('touchmove', process_touchmove, false);
someElement.addEventListener('touchcancel', process_touchcancel, false);
someElement.addEventListener('touchend', process_touchend, false);
```

Process an event in an event handler, implementing the application's gesture semantics.

```
// touchstart handler
function process_touchstart(ev) {
  // Use the event's data to call out to the appropriate gesture handlers
  switch (ev.touches.length) {
    case 1: handle_one_touch(ev); break;
    case 2: handle_two_touches(ev); break;
    case 3: handle_three_touches(ev); break;
    default: gesture_not_supported(ev); break;
  }
}
```
Access the attributes of a touch point.
```
// Create touchstart handler
someElement.addEventListener('touchstart', function(ev) {
  // Iterate through the touch points that were activiated
  // for this element and process each event 'target'
  for (var i=0; i < ev.targetTouches.length; i++) {
    process_target(ev.targetTouches[i].target);
  }
}, false);
```
Prevent the browser from processing *emulated mouse events*.
```
// touchmove handler
function process_touchmove(ev) {
  // Set call preventDefault()
  ev.preventDefault();
}
```

Best practices

Here are some *best practices* to consider when using touch events:

- Minimize the amount of work done that is done in the touch handlers.
- Add the touch point handlers to the specific target element (rather than the entire document or nodes higher up in the document tree).
- Add touchmove, touchend and touchcancel event handlers within the touchstart.
- The target touch element or node should be large enough to accommodate a finger touch. If the target area is too small, touching it could result in firing other events for adjacent elements.

Implementation and deployment status

The touch events browser compatibility data indicates touch event support among mobile browsers is relatively broad, with desktop browser support lagging although additional implementations are in progress.
Some new features regarding a touch point's touch area - the area of contact between the user and the touch surface - are in the process of being standardized. The new features include the *X* and *Y* radius of the ellipse that most closely circumscribes a touch point's contact area with the touch surface. The touch point's *rotation angle* - the number of degrees of rotation to apply to the described ellipse to align with the contact area - is also be standardized as is the amount of pressure applied to a touch point.

What about Pointer Events?

The introduction of new input mechanisms results in increased application complexity to handle various input events, such as key events, mouse events, pen/stylus events, and touch events. To help address this problem, the Pointer Events standard *defines events and related interfaces for handling hardware agnostic pointer input from devices including a mouse, pen, touchscreen, etc.*. That is, the abstract *pointer* creates a unified input model that can represent a contact point for a finger, pen/stylus or mouse.
The pointer event model can simplify an application's input processing since a pointer represents input from any input device. Additionally, the pointer event types are very similar to mouse event types (for example, pointerdown pointerup) thus code to handle pointer events closely matches mouse handling code.
The implementation status of pointer events in browsers is relatively low with IE11 and Edge having complete implementations. Firefox's implementation has been withdrawn because of bug 1166347.