Solution

Handling data types

When passed directly with unsupported type `undefined`, `Symbol`, and `Function`, JSON.stringify outputs `undefined` (not the string `'undefined'`):

```
JSON.stringify(undefined); // undefined
JSON.stringify(Symbol('foo')); // undefined
JSON.stringify(() => {}); // undefined
```

For other built-in object types (except for `Function` and `Date`) such as `Map`, `Set`, `WeakMap`, `WeakSet`, `Regex`, etc., `JSON.stringify` will return a string of an empty object literal, i.e. `{}`:

```
JSON.stringify(/foo/); // '{}'
JSON.stringify(new Map()); // '{}'
JSON.stringify(new Set()); //'{}'
```

`NaN` and `Infinity` are converted into `null`, and `Date` objects are encoded into ISO strings by `JSON.stringify` because of `Date.prototype.toJSON`. And yes, we will have to take care of a custom `toJSON` method present in the input value.

Cyclic references

Finally, `JSON.stringify` can detect a cyclic object i.e. objects with circular references and bail out from the stringification by throwing an error. We will have to account for that as well.

```
const foo = {};
foo.a = foo;

JSON.stringify(foo); // ❌ Uncaught TypeError: Converting circular structure to JSON
```

To detect circular references in an object, we can use a `Set` to keep track of property values we have visited while traversing the object. As soon as we find a value that exists in the set already, we know the object has circular references.

Here is how we would write it:

JavaScript TypeScript

```
function isCyclic(input) {
  const seen = new Set();

  function dfsHelper(value) {
    if (typeof value !== 'object' || value === null) {
      return false;
    }

    seen.add(value);
    return Object.values(value).some(
      (value_) => seen.has(value_) || dfsHelper(value_),
    );
  }

  return dfsHelper(input);
}

const QUOTE_ESCAPE = /"/g;

/**
 * @param {*} value
 * @return {string}
 */
export default function jsonStringify(value) {
  if (isCyclic(value)) {
    throw new TypeError('Converting circular structure to JSON');
  }

  if (typeof value === 'bigint') {
    throw new TypeError('Do not know how to serialize a BigInt');
  }

  if (value === null) {
    // Handle null first because the type of null is 'object'.
    return 'null';
  }
```

```
}

const type = typeof value;

if (type === 'number') {
  if (Number.isNaN(value) || !Number.isFinite(value)) {
    // For NaN and Infinity we return 'null'.
    return 'null';
  }
  return String(value);
}

if (type === 'boolean') {
  return String(value);
}

if (type === 'function' || type === 'undefined' || type === 'symbol') {
  return undefined; // Not the string 'undefined'.
}

if (type === 'string') {
  // Wrap in double quotes/
  return `"${value.replace(QUOTE_ESCAPE, '\\"')}"`;
}

// At this point `value` is either an array, a plain object,
// or other unsupported object types such as `Map` and `Set`.
if (typeof value.toJSON === 'function') {
  // If value has user-provided `toJSON` method, we use that instead.
  return jsonStringify(value.toJSON());
}

if (Array.isArray(value)) {
  const arrayValues = value.map((item) => jsonStringify(item));
  return `[${arrayValues.join(',')}]`;
}

// `value` is a plain object.
const objectEntries = Object.entries(value)
  .map(([key, value]) => {
    const shouldIgnoreEntry =
      typeof key === 'symbol' ||
      value === undefined ||
```

```
      typeof value === 'function' ||
      typeof value === 'symbol';

    if (shouldIgnoreEntry) {
      return;
    }

    return `"${key}":${jsonStringify(value)}`;
  })
  .filter((value) => value !== undefined);

 return `{${objectEntries.join(',')}}`;
}
```

Notes

- There are still uncovered edge cases with the current implementation. Check out **the spec** if you are interested in learning more about it.
  - In particular, special characters like `\n`, `\t` need to be converted into `\\n`, `\\t` respectively.
- One possible follow-up question could be to make it faster. The current implementation involves frequent runtime type checks due to the dynamic typing nature of the JavaScript language. One way we can make the above implementation of `JSON.stringify` faster is to have the user provide a schema of the object (e.g. using **JSON Schema**) so we know the object structure before serialization. This can save us a ton of guesswork. In fact, many `JSON.stringify`-alternative libraries are implemented this way to make serialization faster. One example would be **fast-json-stringify**.