# `const` and Pointers

- Using the syntax below, while `obj` is declared by-reference, the compiler will block any attempts to modify its contents:
  - `const Object* obj;`
  - The referenced object `obj` is considered constant.

# `const` and Parameters

- Suppose a method is defined as follows:

```
void someMethod(const Object* obj)
```

- What implications would this have?

# `const` and Parameters

```
void someMethod(const Object* obj)
```

- What implications would this have?
  - As `obj` is defined `const Object*`, we would get a pointer to an unmodifiable instance of `Object`.
  - What are we able to pass in as an argument to `obj`?

# `const` and Parameters

```
void someMethod(const Object* obj)
```

- Which of these would be proper calls?
  - `const Object obj();`
    `someMethod(&obj);`
  - `Object* obj = new Object();`
    `someMethod(obj);`

# `const` and Parameters

```
void someMethod(const Object* obj)
```

- Which of these would be proper calls?
  - `const Obje`
    `someMethoc`
  - `Object* ob`
    `someMethoc(obj);`

Trick Question! Both!

# `const` and Parameters

```
void someMethod(const Object* obj)
```

- While the original argument to methods of this form do not have to be `const`, they become `const` within the method.

# `const` and Parameters

```
void someMethod(Object* obj)
```

- Which of these would be proper calls?
  - `const Object obj();`
    `someMethod(&obj);`
  - `Object* obj = new Object();`
    `someMethod(obj);`

# `const` and Parameters

```
void someMethod(Object* obj)
```

- Which of these would be proper calls?
  - ~~`const Object obj();`~~
    ~~`someMethod(&obj);`~~
  - `Object* obj = new Object();`
    `someMethod(obj);`

# `const` and Parameters

- `const` objects cannot be passed by-reference to non-`const` function parameters.
  - As there is no guarantee that the referenced object will not be modified when passed to a non-`const` parameter, the compiler blocks this.
  - For value types, since a separate value is created, that separate copy is safe for the called function to edit.

# `const` and Parameters

- `const` objects cannot be passed by-reference to non-`const` function parameters.
  - An interesting consequence of this:

```
void someMethod(string &str);

someMethod(string("Hello World"));
// Will be a compile-time error
// due to the compile-time constant.
```

# `const` and Parameters

- `const` objects cannot be passed by-reference to non-`const` function parameters.
  - An interesting consequence of this:

```
void someMethod(const string &str);

someMethod(string("Hello World"));
// Will work without issue!
```

# `const` and Parameters

- A signature of the latter type –

```
void someMethod(const string &str)
```

has one additional benefit.

  - Since `str` is passed by reference here, the system doesn't have to copy its value…
  - And since `str` is declared `const`, its value cannot be changed.

# `const` and Parameters

- A signature of the latter type –

```
void someMethod(const string &str)
```

has one additional benefit.

  - Consider if this were a very large string.
  - Or, just some very large object.
  - This makes the program more efficient in terms of run-time and in terms of memory use.

# `const` and Return Values

- `const` may also be applied to return values!
  - Consider if we were to return a reference to an object's internal field.
  - Rather than copy the internal object, we may wish to return it while blocking write access within the object.

Example: `const Object* gimmeObject();`