Abstract Data Types

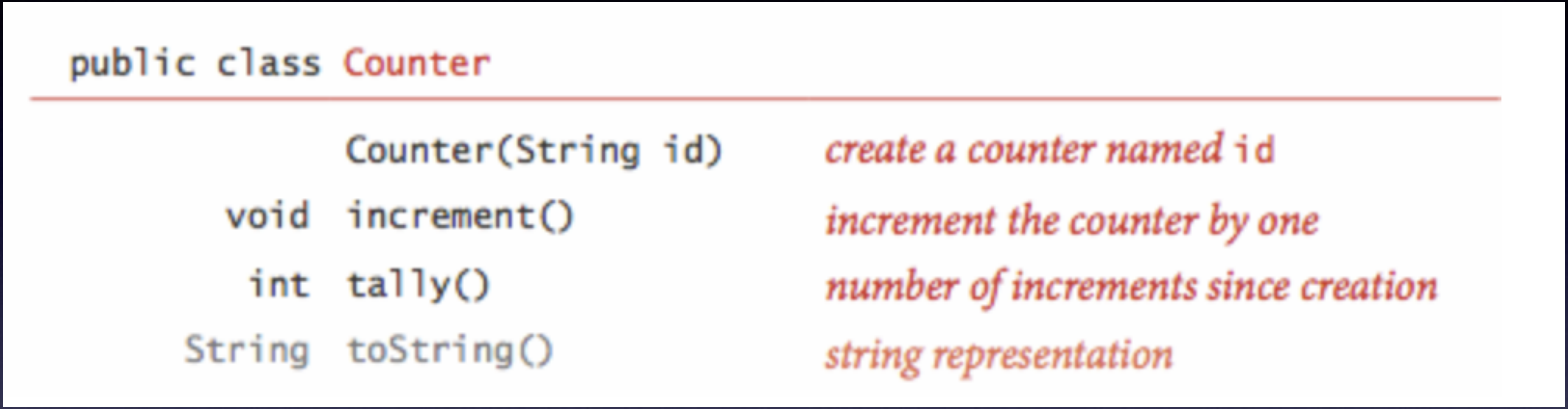
Please note that these slides are based in part on material originally developed by Prof. Kevin Wayne of the CS Dept at Princeton University.

Abstract Data Types

- Last week, I referred to ADTs and APIs without definitions. So…
  - *Data types*: A data type is a set of values and a set of operations on those values.
  - *Abstract data types*: An abstract data type is a data type whose internal representation is hidden from the client (encapsulation).
  - *Objects*: An object is an entity that can take on a data-type value. Objects are characterized by three essential properties: The state of an object is a value from its data type; the identity of an object distinguishes one object from another; the behavior of an object is the effect of data-type operations. In Java, a reference is a mechanism for accessing an object.
  - *Applications programming interface (API)*: To specify the behavior of an abstract data type, we use an application programming interface (API), which is a list of constructors and instance methods (operations), with an informal description of the effect of each.
    - *Client*: A client is a program that uses a data type.
    - *Implementation*: An implementation is the code that implements the data type specified in an API.

Or, no internal representation at all

ADT/API example



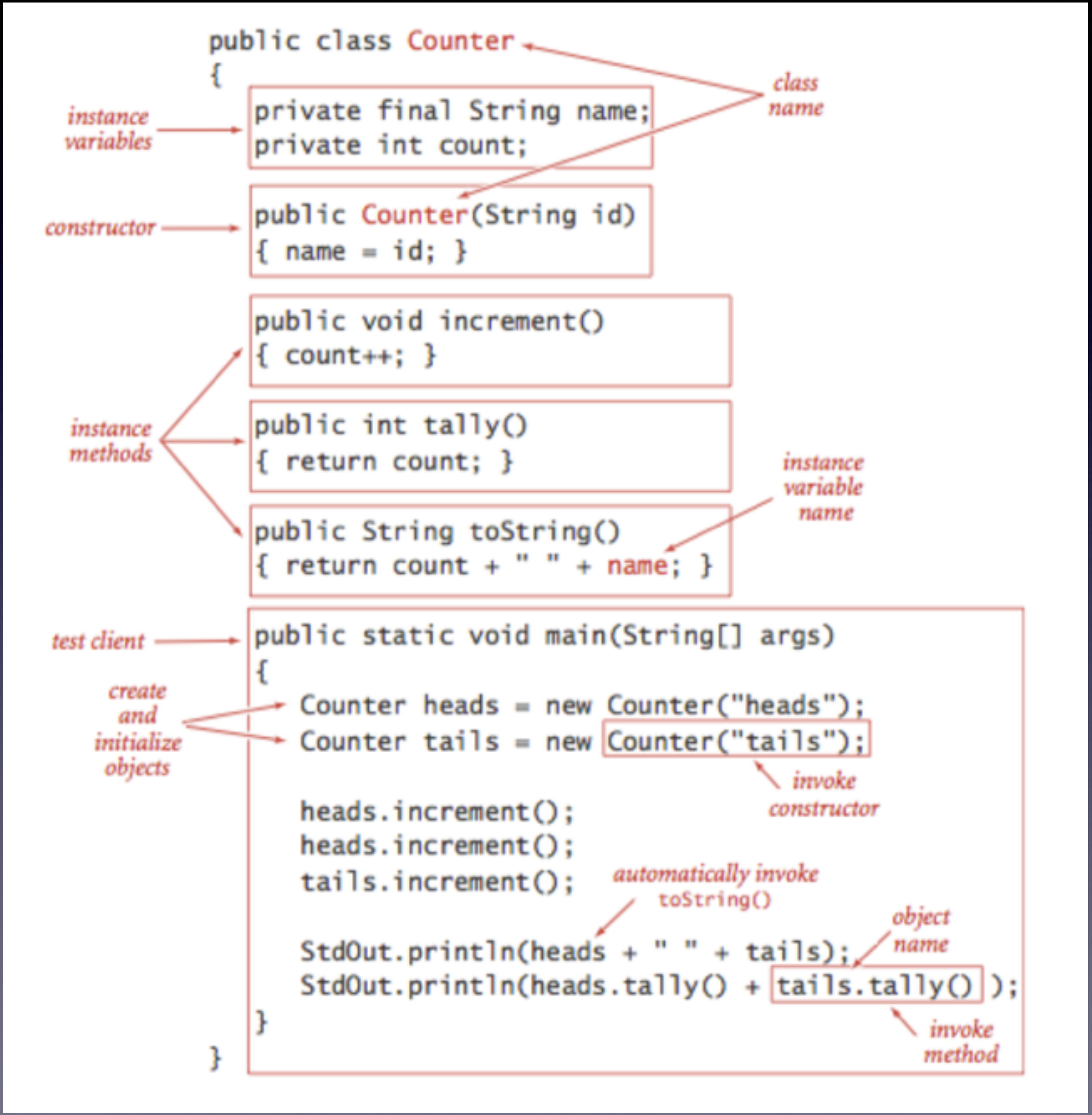
`public class Counter`

	`Counter(String id)`	*create a counter named* `id`
`void`	`increment()`	*increment the counter by one*
`int`	`tally()`	*number of increments since creation*
`String`	`toString()`	*string representation*

- instantiation: `Counter heads = new Counter(“heads”);`
- increment: `heads.increment();`
- tally: `if (heads.tally() > 100) return;`
- show: `System.out.println(“Heads: “+heads);`

Note that we don’t know anything about the internal details. We are just following the API

Implementation



```
public class Counter
{
   private final String name;
   private int count;

   public Counter(String id)
   { name = id; }

   public void increment()
   { count++; }

   public int tally()
   { return count; }

   public String toString()
   { return count + " " + name; }

   public static void main(String[] args)
   {
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");

      heads.increment();
      heads.increment();
      tails.increment();

      StdOut.println(heads + " " + tails);
      StdOut.println(heads.tally() + tails.tally() );
   }
}
```

Implementing ADTs

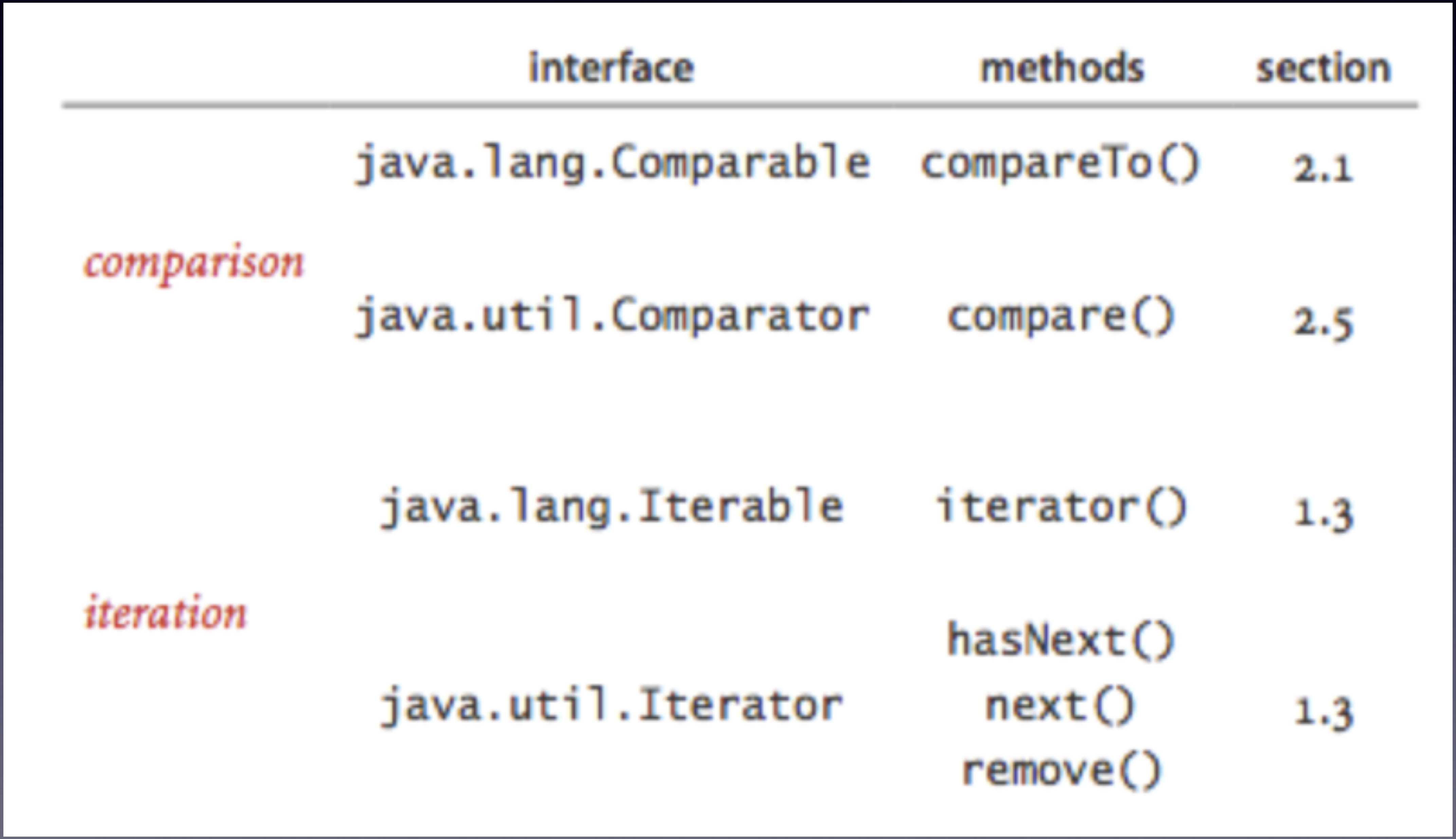
Designing abstract data types. We put important information related to designing data types in one place for reference and to set the stage for implementations throughout this book.

- *Encapsulation.* A hallmark of object-oriented programming is that it enables us to *encapsulate* data types within their implementations, to facilitate separate development of clients and data type implementations. Encapsulation enables modular programming.
- *Designing APIs.* One of the most important and most challenging steps in building modern software is designing APIs. Ideally, an API would clearly articulate behavior for all possible inputs, including side effects, and then we would have software to check that implementations meet the specification. Unfortunately, a fundamental result from theoretical computer science known as the *specification problem* implies that this goal is actually impossible to achieve. There are numerous potential pitfalls when designing an API:
    - Too hard to implement, making it difficult or impossible to develop.
    - Too hard to use, leading to complicated client code.
    - Too narrow, omitting methods that clients need.
    - Too wide, including a large number of methods not needed by any client.
    - Too general, providing no useful abstractions.
    - Too specific, providing an abstraction so diffuse as to be useless.
    - Too dependent on a particular representation, therefore not freeing client code from the details of the representation.
- In summary, provide to clients the methods they need and no others.

Implementing ADTs (contd.)

- *Algorithms and ADTs.* Data abstraction is naturally suited to the study of algorithms, because it helps us provide a framework within which we can precisely specify both what an algorithm needs to accomplish and how a client can make use of an algorithm. For example, our whitelisting example at the beginning of the chapter is naturally cast as an ADT client, based on the following operations:
    - Construct a SET from an array of given values.
    - Determine whether a given value is in the set.
- *Interface inheritance.* Java provides language support for defining relationships among objects, known as *inheritance*. Interfaces are the answer to too much *coupling*. We use interface inheritance for *comparison* and for *iteration*.
- *Inheritance via sub-classing.* Any (non-final) class can be sub-classed. However, people are not that good at recognizing such situations. In Java, every class is a sub-class of Object which defines several important methods, including *toString, equals and hashCode*.

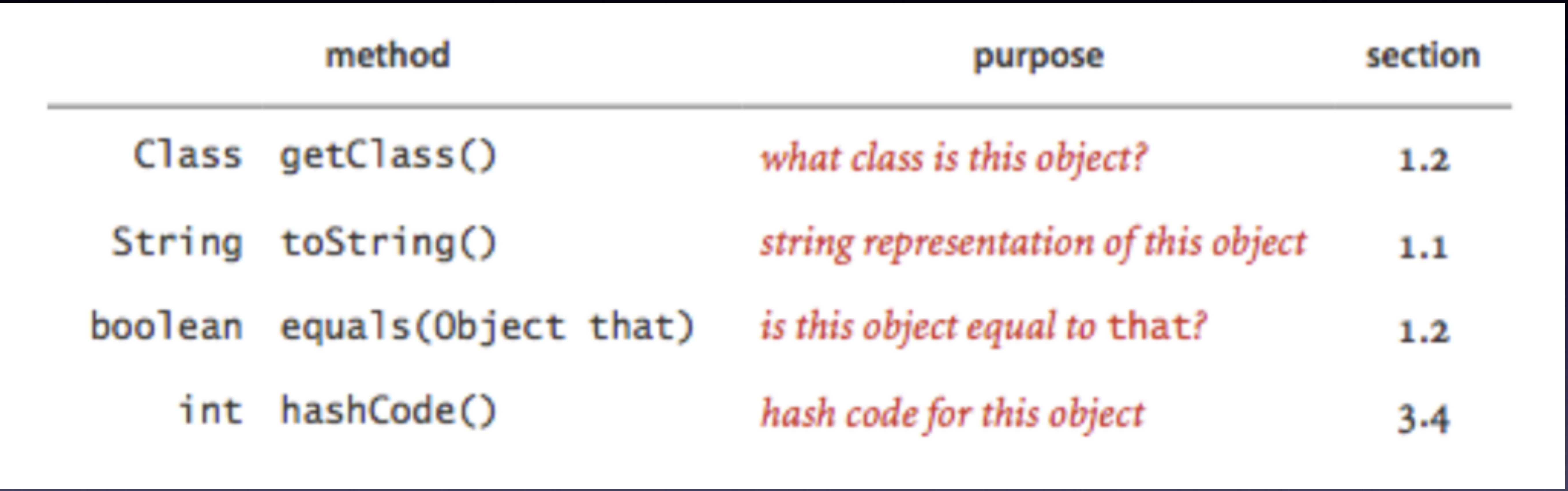
Interfaces



	interface	methods	section
comparison	java.lang.Comparable	compareTo()	2.1
	java.util.Comparator	compare()	2.5
iteration	java.lang.Iterable	iterator()	1.3

	java.util.Iterator	hasNext()
		next()
		remove()
			1.3

Sub-classing Object



	method	purpose	section
Class	getClass()	*what class is this object?*	1.2
String	toString()	*string representation of this object*	1.1
boolean	equals(Object that)	*is this object equal to* that?	1.2
int	hashCode()	*hash code for this object*	3.4

- Please note that these methods aren't just for fun. They are really important (as you'd expect for methods that are defined for *every* object.
- One criticism I have of Java (there are others in my "Software" blog) is that they should have defined *equals* and *hashCode* as part of an interface whereby both (or neither) methods should be defined. That's because it can be really problematic if these two aren't compatible!

Equality

Equality. What does it mean for two objects to be equal? If we test equality with `(a == b)` where `a` and `b` are reference variables of the same type, we are testing whether they have the same identity: whether the *references* are equal. We also need a way to test logical or datatype equality. This is defined in the method *equals()*. When we define our own data types we need to override `equals()`. Java's convention is that `equals()` must be an *equivalence relation*:

- *Reflexive*: `x.equals(x)` is true.
- *Symmetric*: `x.equals(y)` is true if and only if `y.equals(x)` is true.
- *Transitive*: if `x.equals(y)` and `y.equals(z)` are true, then so is `x.equals(z)`.

In addition, it must take an `Object` as argument and satisfy the following properties.

- *Consistent*: multiple invocations of `x.equals(y)` consistently return the same value, provided neither object is modified.
- *Not null*: `x.equals(null)` returns false.

hashCode

- Many datatypes such as *HashMap* or *Set* make much use of *equals* and *hashCode*. The answers must be consistent, otherwise weird behaviors will ensue.
- Also, if something extends *Comparable*, then the result of invoking *compareTo* should also be consistent with *equals*.
- So, for any two objects *x* and *y*:
    - if *x.equals(y)* then *x.hashCode()==y.hashCode()* must be true
    - if *x.equals(y)* then *x.compareTo(y)==0* must be true

Quiz 2