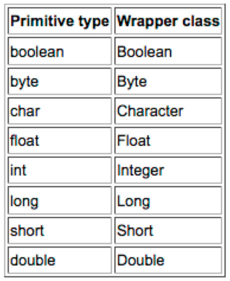
Other Immutable Classes

- Wrapper classes for primitive types



Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

- `java.nio.file.Path`
- `java.util.regex.Pattern`
- Some classes in `java.net`
  - e.g., `URL`, `URI`, `Inet4Address` and `Inet6Address`
- Date and Time API (`java.time`)
  - All the classes are immutable and thread-safe.

`Integer`

- Wrapper class of an `int` value
  - Final class, which cannot be extended (sub-classed)
  - Maintains the initialized `int` data in a private and final data field.

```
–  Integer int = Integer.valueOf(10);
–  Integer int = 10;    // Auto-boxing; Syntactic sugar for
                        // the above code
```

  - Has no setter methods; no methods change the initialized `int` data.
  - All methods are thread-safe.

Note That…

- An immutable class's methods are thread-safe, but…
- Client code of those methods may or may not be thread-safe.
  - The code below is NOT thread-safe; it requires thread synchronization.

```
public class Person {
 private Integer age;              // Shared variables
 ...
 public void setAge(Integer age){
   this.age = age;                 // 2 steps, but thread-safe
                                   // “age” is a local variable. }

 public void getAge(){
    return this.age;               // 2 steps. Not thread-safe  }

 public boolean isKndergartener(){    // Multi-steps. Not thread-safe
   if(this.age < 6){return false;}
   else{return true;}
 }
```

- An immutable class's methods are thread-safe, but…
- Client code of those methods may or may not be thread-safe.
  - The code below is thread-safe.

```
public class ErrorMsgGenerator {
 private final Integer FILE_NOT_FOUND = new Integer.valueOf(404);
 ...

 public String getFileNotFoundErrorMsg(Path path){
   String header = “Error code: “ + FILE_NOT_FOUND;
      // Syntax sugar for
      //      new StringBuilder().append(“...”)
      //                        .append(FILE_NOT_FOUND.toString()).toString()
      // Reads on local and final variables are thread-safe.
      // An instance of StringBuilder is local for each thread.

   String body = “The requested file “ +
                 path.toString() + “ was not found.”
      // “path” is a local variable. Read on it is thread-safe.

   return header + “ “ + body;                // Thread-safe}  }
```

Date and Time API: History

- `java.util.Date` (since JDK 1.0)
  - Poorly designed: Never try to use this class
    - It still exists only for backward compatibility
- `java.util.Calendar` (since JDK 1.1)
  - Deprecated many methods of `java.util.Date`
  - Limited capability: Try not to use this class
- Date and Time API (`java.time`)
  - Since JDK 1.8
  - Always try to use this API.

Date and Time API: `Instant`

- Represents an instantaneous point on the timeline, which starts at 01/01/1970 (on the prime Greenwich meridian).
  - Can be used as a timestamp.
- `Duration`
  - Represents an amount of time in between two `Instant`s

```
– Instant start = Instant.now();
  ...
  Instant end = Instant.now();
  Duration timeElapsed = Duration.between(start, end);
  long timeElapsedMSec = timeElapsed.toMillis();
```

  - This code is thread-safe as far as all the variables are local variables.

Date and Time API: "Local" Classes

- `LocalDate, LocalTime, LocalDateTime`
  - Used to represent date and time without a time zone (time difference)
  - Apply leap-year rules automatically.

```
• LocalDate today = LocalDate.now();
  LocalDate birthday = LocalDate.of(2009, 9, 10);
  LocalDate 18thBirthday = birthday.plusYears(18);
  birthday.getDayOfWeek().getValue();
```

- `Period`
  - Represents an amount of time in between two local date/time.

```
• Period period = today.until( 18thBirthday );
  period.getDays();
```

- All these code are thread-safe as far as all the variables are local variables.

Date and Time API: Other Classes

- `TemporalAdjusters`
  - Utility class that implements various calendaring operations.
    - e.g., Getting the first Sunday of the month.
- `ZonedDateTime`
  - Similar to `LocalDateTime`, but considers time zones (time difference) and time-zone rules such as daylight savings.
- `DateTimeFormatter`
  - Useful to parse and print date-time objects.
- All public methods are thread-safe in these classes.

Implementing User-Defined (Your Own) Immutable Classes

- Immutable class
  - Defined as a `final` class
  - Has `private final` data fields only.
  - Has no setter methods.
  - c.f. A Strategy for Defining Immutable Objects
    - https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html
- Clearly state immutability in program comments, API documents, design documents, etc.
  - Java API documentation does so too.
  - Use {frozen} or {immutable} in UML class diagrams

An Example User-Defined Immutable Class

```
public final class SSN {
  private final int first3Digits, middle2Digits, last4Digits;

  public SSN(int first, int middle, int last){   // Thread-safe
      this.first3Digits = first;
      this.middle2Digits = middle;
      this.last4Digits = last;  }

  public int getLast4Digits(){ return last4Digits; }

  public String toString(){
      return first3Digits + "-" + middle2Digits + "-" + last4Digits;
        // Multiple steps, but thread-safe
        // Those 3 data fields are immutable  }

  public boolean equals( SSN anotherSSN ){
      if( this.toString().equals(anotherSSN.toString()) ){ return true; }
      else{ return false; }
        // Multiple steps, but thread-safe
        // String.toString() and String.equals() are thread-safe
        // "this" and "anotherSSN" are immutable  }  }
```

-

```
public final class SSN {
  private final int first3Digits, middle2Digits, last4Digits;

  public SSN(int first, int middle, int last){ // Thread-safe
      this.first3Digits = first;
      this.middle2Digits = middle;
      this.last4Digits = last;  }
```

- A constructor is always executed as atomic code.
  - Only one thread can run a constructor on a class instance that is being created and initialized.
    - Multiple threads never call a constructor(s) on the same instance concurrently.
  - Until a thread returns/completes a constructor on a class instance, no other threads can call public methods on that instance.

Note That…

- An immutable class's methods are thread-safe, but…
- Client code of those methods may or may not be thread-safe.
  - The code below is thread-safe.

```
public class Person {
  private final SSN ssn;  // Shared final variable

  public Person(SSN ssn){ this.ssn = ssn; }

  public SSN getSSN(){     // 2 Steps, but thread-safe.
    return ssn;            // "ssn" is final.
  } }

Person person = new Person( new SSN(012, 34, 5678) );
person.getSSN();
```

- An immutable class’s methods are thread-safe, but…
- Client code of those methods may or may not be thread-safe.
  - The code below is NOT thread-safe; it requires thread synchronization.

```
public class Person {
  private SSN ssn;        // Shared (non-final) variable

  public Person(SSN ssn){ this.ssn = ssn; }

  public SSN setSSN(SSN ssn){
    this.ssn = ssn;       // 2 steps but thread-safe.
                          // “ssn” is a local variable.

  public SSN getSSN(){    // 2 steps. NOT thread-safe.
    return ssn;
  } }
```

Person requires thread synchronization to guard **ssn**, although **SSN** does not.

```
public class Customer {
  private Address address;    // Shared (non-final) variable

  public Person(Address addr){ address = addr; }

  public Address setAddress(Address addr){
    address = addr;            // Customer needs a setter.
                               // 2 steps, but thread-safe.
                               // “addr” is a local variable. }

  public Address getAddress(){ // 2 steps. NOT thread-safe.
    return address;
  }  }
```

```
Customer customer = new Customer( new Address( ... ) );
customer.getAddress();
customer.setAddress( new Address ( ...) );
customer.setAddress( customer.getAddress().change( ... ) );
```

Customer requires thread synchronization to guard **address**, although **Address** does not.

HW 12

- Implement your own immutable class:

  -
    ```
    public final class Address {
     private final String street, city, state;
     private final int zipcode;
     ... }
    ```
  - Define a constructor that takes 4 parameters and initializes an address.
  - Define getter methods: **equals()** and **toString()**
    - c.f. **SSN**’s **equals()** and **toString()**
  - Define **change()** to change the current address

    -
      ```
      public Address change(String street, String city,
                            String state, int zipcode){
          return new Address(street, city, state, zipcode);  }
      ```
    - It sounds like a setter, but it is NOT. It creates a new instance and returns it.
- Turn in
  - immutable **Address**
  - thread-safe **Customer**
  - **Runnable** class whose **run()** calls **Customer**’s **setAddress()** and **getAddress()**
    - You can replace the **Runnable** class with a lambda expression, if you like
  - Test code to create and run multiple threads
- Deadline: April 11 (Thu) midnight

Performance Implication

- An immutable object makes a bigger difference in performance
  - As more threads read data from the object more often.
- If you are interested, compare the performance of
  - Immutable **`Address`** and
  - Mutable **`Address`** that performs thread synchronization in its setters and getters.
  - Immutable **`Address`** is approx. 25% faster on my machine.

- An immutable object never trigger performance loss in single-threaded apps.
  - If an single-threaded app calls a mutable object's method that performs thread synchronization, the app incurs unnecessary performance loss.
    - The app never need thread synchronization, but the mutable object's method does it for the app.

Well, Not All Classes can be Immutable…

- Immutable classes are good for both API designers and users.
- However, in practice, some/many classes need to be mutable…
- Think of separating a class to mutable and immutable parts
  - if read operations are called very often.

An Example: `String` and `StringBuilder`

- Both represent string data.
- **`String`**
  - Immutable: Its state never change.
  - Thread-safe
  - Faster to run read operations (getters).
  - Slower to run write operations (setters).
- **`StringBuilder`**
  - Mutable: Its state can change through its methods.
  - Not thread-safe; its public methods never perform thread synch.
  - Faster to perform write operations (setters).
  - Slower to perform read operations (getters).

Performance of String Concatenation

-
  ```
  String str = ”UMass”;
  str = str + “ Boston”;          // “UMass Boston”
      // Syntax sugar for:
      // str = new StringBuilder(str).append(“ Boston”).toString();
      // Creates 2 instances: StringBuilder and String
      // Calls 3 methods: StringBuilder’s constructor and 2 methods
  ```

-
  ```
  StringBuilder builder new StringBuilder(str);
  builder.append(“ Boston);
  str = builder.toString();
  ```

- No difference in performance.

-
  ```
  String header = “Error code: “ + FILE_NOT_FOUND;
  String body = “The requested file “ + path.toString()
                                       + “ was not found.”
  return header + “ “ + body;
     // Syntax sugar for:
     // header = new StringBuilder(“...”).append(FILE_NOT_FOUND).toString();
     // body = new StringBuilder(“...”).append(...).append(“...”).toString();
     // return new StringBuilder(header).append(“ ”).append(body).toString();
     //
     // Creates 6 instances and calls 11 methods
  ```

-
  ```
  StringBuilder builder new StringBuilder();
  builder.append(“Error code: “);
  builder.append(FILE_NOT_FOUND);
  builder.append(“The requested file “);
  ...
  builder.append(” was not found.”);
  return builder.toString();
     // Creates 2 instance and calls 5 methods
  ```

- More visible difference in performance, if string concatenation is performed with multiple statements.

-
  ```
  LinkedList<String> emailAddrs = ...;
  ```

-
  ```
  String commaSeparatedEmailAddrs;
  for(String emailAddr: emailAddrs){
      commaSeparatedEmailAddrs += emailAddr + “, ”;   }
  ```

-
  ```
  StringBuilder commaSeparatedEmailAddrs;
  for(String emailAddr: emailAddrs){
      commaSeparatedEmailAddrs.append(emailAddr).append(“, ”); }
  ```

- The latter code can run 20-100% faster depending on the number of collection elements (i.e. email addresses).

- Use `String` (immutable class) for read operations
- Use `StringBuilder` (mutable class) for write operations
  - Note that `StringBuilder`’s methods are NOT thread-safe; e.g., `append()`.

- `String`-to-`StringBuilder` conversion is implemented in a constructor of `StringBuilder`.

- `StringBuilder`-to-`String` conversion is implemented in a constructor of `String`.

`StringBuffer`

- Provides the same set of public methods as **`StringBuilder`** does.

- **`StringBuffer`** (since Java 1.0)
  - All public methods are thread-safe with locking.
  - Client code of **`StringBuffer`** may still require locking.
  - DO NOT use this class.
    - It makes no sense to use it in single-threaded apps.

- **`StringBuilder`** (since Java 5)
  - All public methods are NOT thread-safe.
  - Client code of **`StringBuilder`** require locking.
  - Use this class
    - regardless of single-threaded or multi-threaded apps.