Working With Built-in Java Classes Part II

"Code with Passion!"

Topics

- Type casting
- Primitive type vs Wrapper type
- Comparing objects
- *getClass()* method of Object class and *instanceof* operator

Type Casting

What is Type Casting?

- Mapping type of an object to another type
- To be discussed
  - Casting primitives
  - Casting objects (we will cover this more in “Java inheritance”)

Casting Primitives

- Casting between primitives enables you to convert the value of one data from one primitive type to another primitive type.

- Types of Casting:
  - Implicit Casting
  - Explicit Casting

Implicit Casting of Primitive Types

- Suppose we want to store a value of int data type to a variable of data type double.

```
int      numInt = 10;
double  numDouble = numInt; //implicit cast
```

In this example, since the destination variable's data type (double) holds a larger value than the value's data type (int), the data is implicitly casted to the destination variable's data type double without a problem since there is no loss of information

Implicit Casting of Primitive Types

- Another example:

```
int          numInt1 = 1;
int          numInt2 = 2;

// result is implicitly casted to type double
double       numDouble = numInt1/numInt2;
```

Explicit Casting of Primitive Types

- When we convert a data that has a large type to a smaller type, we must use an explicit cast because there is a possibility of losing information and Java compiler wants to make sure you know what you are doing

- Explicit casts take the following form:

```
(Type)value
```

Type - is the name of the type you're converting to
value - is an expression that results in the value of the source type

Examples: Explicit Casting of Primitive Types

```
double   valDouble = 10.12;
// Without (int) casting, compiler error will occur
int      valInt = (int)valDouble;

// convert valDouble to int type
double x = 10.2;
int y = 2;

// Without (int) casting, compiler error will occur
int result = (int)(x/y);
```

Casting Objects

- Instances of classes also can be cast into instances of other classes, with one restriction: The source and destination classes must be related by inheritance; one class must be a subclass (child class) of the other.
  - We'll cover more about inheritance later

- Casting objects is analogous to converting a primitive value to a larger type, some objects might not need to be cast explicitly.

Casting Objects

- To cast, prepend with (classname)

```
(classname)object
```

classname is the name of the destination class &
object is a reference to the source object

Casting Objects

- The following example casts an instance of the class VicePresident to an instance of the class Employee; VicePresident is a subclass of Employee with more information, which here defines that the VicePresident has executive washroom privileges.

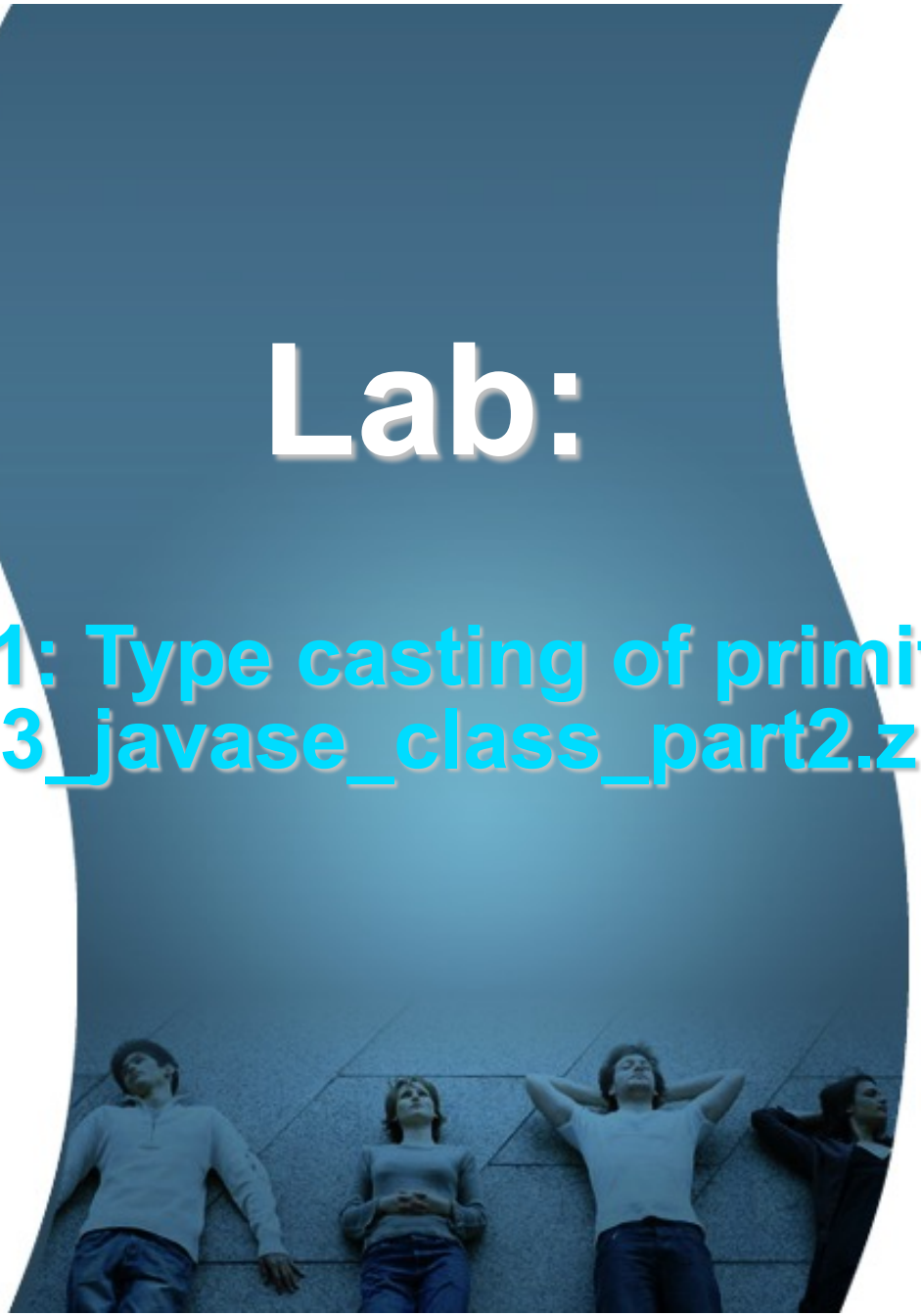
```
Employee emp = new Employee();
VicePresident veep = new VicePresident();

// No cast needed for upward use because VicePresident is
// a subtype of Employee type, meaning any VicePresident
// is also a Employee type
emp = veep;

// Must cast explicitly because not all employee objects
// are VicePresident type
veep = (VicePresident)emp;
```

Lab:

Exercise 1.1: Type casting of primitive types
1013_javase_class_part2.zip



Primitives & Wrapper Types

Primitive types vs Reference types

- In Java, every variable has to be declared with a type
  - Java is called statically typed language for this reason
- There are two kinds of types
  - Primitive types: directly contain values
  - Reference types (Object types): are references to objects

Wrapper Classes

- Every primitive type has corresponding Wrapper type - they are Reference type or Object type version of primitive types
  - Integer is a wrapper type of the primitive int
  - Double is a wrapper type of the primitive double
  - Long is a wrapper type of the primitive long
- Using the Wrapper type, you create an object that holds the same value of corresponding primitive type

Why Wrapper Classes?

- Facts about classes/objects
  - Only objects can access methods of the *Object* class
  - Primitive data types are not objects, thus cannot access methods of the *Object* class
- Why are Wrapper classes needed?
  - Need an object representation for the primitive type variables to use to take advantage of the properties and methods of Object class
  - Collection can take only Object as its element – no primitive type can be added to a Collection

Converting Primitive types to Objects (Wrapper) and vice versa

- Create an instance of the Integer class with the integer value 7801

```
Integer dataCount = new Integer(7801);
```

- Convert an Integer object to its primitive data type int. The result is an int with value 7801

```
int newCount = dataCount.intValue();
```

- Converting a String to a numeric type, such as an int (Object->primitive)

```
String pennsylvania = "65000";
int penn = Integer.parseInt(pennsylvania);
```

Lab:

Exercise 1.2: Primitive type to/from Wrapper type conversion
1011_javase_class.zip

Comparing Objects

Comparing Objects with == and !=

- In our previous discussions, we learned about operators for comparing values of primitive types
  - equal ==, not equal !=, less than <, and so on

- When equality: == (equal) and != (not equal) operators are applied to objects, they determine whether both sides of the operator refer to the same object instance instead of checking whether one object has the same value as the other object

Comparing String Objects

```
1   class EqualsTest {
2       public static void main(String[] arguments) {
3         String str1, str2;

4         str1 = "Free the bound periodicals.";
5         str2 = str1;
6         System.out.println("String1: " + str1);
7         System.out.println("String2: " + str2);
8         System.out.println("Same object? " + (str1 == str2)); //true
9
10        str2 = new String(str1);
11        System.out.println("String1: " + str1);
12        System.out.println("String2: " + str2);
13        System.out.println("Same object? " + (str1 == str2)); //false
14        System.out.println("Same value? " + str1.equals(str2));//true
15      }
16  }
```

Comparing String Objects

- Given the code

```
String str1 = “Hello”;
String str2 = “Hello”;
```

- These two references, str1 and str2, will point to the same object. So, comparing them with == results in true

- String literals are optimized in Java; if you create a string using a literal and then use another literal with the same characters, Java knows enough to give you the same String object instance back from “String constant pool”

Comparing String Objects

- Given the code

```
String str1 = new String("Hello");
String str2 = new String("Hello");
```

- These two references, str1 and str2, will point to different object instances. So comparing them with == results in false.

Lab:

Exercise 2: Comparing Objects
1011_javase_class.zip

getClass() method & instanceof Operator

getClass() method

- The getClass() method returns a Class object instance
  - Every Class loaded in JVM is represented by a Class object
- Class class has a method called getName().
  - getName() returns a string representing the name of the class.

```
String name = myObject.getClass().getName();
```

instanceof operator

- The instanceof checks of an object is a instance of a particular class (type)

```
boolean ex1 = "Texas" instanceof String; // true

Object point = new Point(10, 10);
boolean ex2 = point instanceof String; // false
boolean ex3 = point instanceof Point; // true
```

Lab:

Exercise 3: getClass() method & instanceof operator
1011_javase_class.zip

Code with Passion!